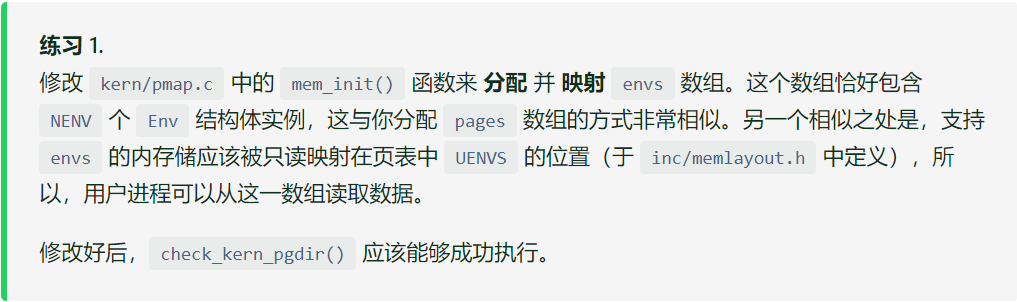
LAB3

练习1：



Mem\_init:

为envs分配内存：

1. envs = (**struct** Env\*)boot\_alloc(NENV\***sizeof**(**struct** Env));
2. memset(envs, 0, NENV \* **sizeof**(**struct** Env));

在页表中建立映射

boot\_map\_region(kern\_pgdir,UENVS,PTSIZE,PADDR(envs),PTE\_U|PTE\_P);

练习2：



env\_init() 初始化全部 envs 数组中的 Env 结构体，并将它们加入到 env\_free\_list 中。因为要保证Env 在 env\_free\_list 中的顺序要和它在 envs 中的顺序一致，所以采用这种遍历顺序。

env\_init\_percpu ，这个函数会通过配置段硬件，将其分隔为特权等级 0 (内核) 和特权等级 3（用户）两个不同的段。

1. **void**
2. env\_init(**void**)
3. {
4. **int** i;
5. **for** (i = NENV-1;i >= 0; --i) {
6. //initialize backwards to maintain the order
7. envs[i].env\_id = 0;
8. //normal link-list routine
9. envs[i].env\_link = env\_free\_list;
10. env\_free\_list = envs+i;
11. }
12. env\_init\_percpu();
13. }

env\_setup\_vm() 为新的进程分配一个页目录，并初始化新进程的地址空间对应的内核部分。

1. **static** **int**
2. env\_setup\_vm(**struct** Env \*e)
3. {
4. **int** i;
5. **struct** PageInfo \*p = NULL;
6. **if** (!(p = page\_alloc(ALLOC\_ZERO)))
7. **return** -E\_NO\_MEM;
8. p->pp\_ref++; //reference count
9. e->env\_pgdir = (pde\_t \*) page2kva(p);
10. memcpy(e->env\_pgdir, kern\_pgdir, PGSIZE);
12. e->env\_pgdir[PDX(UVPT)] = PADDR(e->env\_pgdir) | PTE\_P | PTE\_U;
13. **return** 0;
14. }

region\_alloc 为用户环境分配物理空间

1. **static** **void**
2. region\_alloc(**struct** Env \*e, **void** \*va, **size\_t** len)
3. {
4. **void** \*begin = ROUNDDOWN(va, PGSIZE), \*end = ROUNDUP(va+len, PGSIZE);
5. **void** \*i;
6. **for** (i = begin;i < end; i += PGSIZE) {
7. **struct** PageInfo \*pg = page\_alloc(0);    //not initialized
8. **if** (pg == NULL)
9. panic("region\_alloc failed!");
10. page\_insert(e->env\_pgdir, pg, i, PTE\_W | PTE\_U);
11. }
12. }

load\_icode 功能是为每一个用户进程设置它的初始代码区，堆栈以及处理器标识位。每个用户程序都是ELF文件，所以我们要解析该ELF文件。

1. **static** **void**
2. load\_icode(**struct** Env \*e, uint8\_t \*binary, **size\_t** size)
3. {
4. **struct** Elf \*ELFHDR = (**struct** Elf \*) binary;
5. **struct** Proghdr \*ph, \*eph;
7. **if** (ELFHDR->e\_magic != ELF\_MAGIC)
8. panic("Not executable!");
10. ph = (**struct** Proghdr \*) ((uint8\_t \*) ELFHDR + ELFHDR->e\_phoff);
11. eph = ph + ELFHDR->e\_phnum;
13. //here above is just as same as main.c
15. lcr3(PADDR(e->env\_pgdir));
16. //it's silly to use kern\_pgdir here.
18. **for** (; ph < eph; ph++)
19. **if** (ph->p\_type == ELF\_PROG\_LOAD) {
20. region\_alloc(e, (**void** \*)ph->p\_va, ph->p\_memsz);
21. memset((**void** \*)ph->p\_va, 0, ph->p\_memsz);
22. memcpy((**void** \*)ph->p\_va, binary+ph->p\_offset, ph->p\_filesz);
23. }
25. //we can use this because kern\_pgdir is a subset of e->env\_pgdir
26. lcr3(PADDR(kern\_pgdir));
28. e->env\_tf.tf\_eip = ELFHDR->e\_entry;
29. //we should set eip to make sure env\_pop\_tf runs correctly
31. region\_alloc(e, (**void** \*) (USTACKTOP - PGSIZE), PGSIZE);
32. }

env\_create通过调用 env\_alloc 分配一个新进程，并调用 load\_icode 读入 ELF 二进制映像。

1. **void**
2. env\_create(uint8\_t \*binary, **enum** EnvType type)
3. {
4. // LAB 3: Your code here.
5. **struct** Env \*e;
6. **int** rc;
7. **if** ((rc = env\_alloc(&e, 0)) != 0)
8. panic("env\_create failed: env\_alloc failed.\n");
10. load\_icode(e, binary);
11. e->env\_type = type;
12. }

env\_run() 启动给定的在用户模式运行的进程

1. env\_run(**struct** Env \*e)
2. {
4. **if**(curenv != NULL && curenv->env\_status == ENV\_RUNNING)
5. curenv->env\_status = ENV\_RUNNABLE;
7. curenv = e;
8. curenv->env\_status = ENV\_RUNNING;
9. curenv->env\_runs++;
10. lcr3(PADDR(curenv->env\_pgdir));
12. env\_pop\_tf(&curenv->env\_tf);
14. panic("env\_run not yet implemented");
15. }

练习3：没有

练习4：



trapentry.S

利用上面的两个宏定义设置中断 处理的函数。

1. TRAPHANDLER\_NOEC(divide\_error, T\_DIVIDE)
2. TRAPHANDLER\_NOEC(debug\_exception, T\_DEBUG)
3. TRAPHANDLER\_NOEC(non\_maskable\_interrupt, T\_NMI)
4. TRAPHANDLER\_NOEC(break\_point, T\_BRKPT)
5. TRAPHANDLER\_NOEC(overflow, T\_OFLOW)
6. TRAPHANDLER\_NOEC(bounds\_check, T\_BOUND)
7. TRAPHANDLER\_NOEC(illegal\_opcode, T\_ILLOP)
8. TRAPHANDLER\_NOEC(device\_not\_available, T\_DEVICE)
9. TRAPHANDLER(double\_fault, T\_DBLFLT)
10. TRAPHANDLER(invalid\_task\_switch\_segment, T\_TSS)
11. TRAPHANDLER(segment\_not\_present, T\_SEGNP)
12. TRAPHANDLER(stack\_fault, T\_STACK)
13. TRAPHANDLER(general\_protection\_fault, T\_GPFLT)
14. TRAPHANDLER(page\_fault, T\_PGFLT)
15. TRAPHANDLER\_NOEC(floating\_point\_error, T\_FPERR)
16. TRAPHANDLER(alignment\_check, T\_ALIGN)
17. TRAPHANDLER\_NOEC(machine\_check, T\_MCHK)
18. TRAPHANDLER\_NOEC(SIMD\_floating\_point\_exception, T\_SIMDERR)

Alltraps，将剩下的参数压入栈，即 Trapframe 指针类型的输入参数 tf

1. \_alltraps:
2. pushl %ds
3. pushl %es
4. pushal
5. movl $GD\_KD , %eax
6. movl %eax, %ds
7. movl %eax, %es
8. push %esp
9. call trap

Trap\_init():

使用SETGATE 宏在 idt 表中插入中断向量描述符。

#define SETGATE(gate, istrap, sel, off, dpl)

其中gate是idt表的index入口，istrap判断是异常还是中断，sel为代码段选择符，off表示对应的处理函数地址，dpl表示触发该异常或中断的用户权限。

1. **void** divide\_error();
2. **void** debug\_exception();
3. **void** non\_maskable\_interrupt();
4. **void** break\_point();
5. **void** overflow();
6. **void** bounds\_check();
7. **void** illegal\_opcode();
8. **void** device\_not\_available();
9. **void** double\_fault();
10. **void** invalid\_task\_switch\_segment();
11. **void** segment\_not\_present();
12. **void** stack\_fault();
13. **void** general\_protection\_fault();
14. **void** page\_fault();
15. **void** floating\_point\_error();
16. **void** alignment\_check();
17. **void** machine\_check();
18. **void** SIMD\_floating\_point\_exception();
19. SETGATE(idt[T\_DIVIDE], 0, GD\_KT, divide\_error, 0);
20. SETGATE(idt[T\_DEBUG], 0, GD\_KT, debug\_exception, 0);
21. SETGATE(idt[T\_NMI], 0, GD\_KT, non\_maskable\_interrupt, 0);
22. SETGATE(idt[T\_BRKPT], 0, GD\_KT, break\_point, 3);
23. SETGATE(idt[T\_OFLOW], 0, GD\_KT, overflow, 0);
24. SETGATE(idt[T\_BOUND], 0, GD\_KT, bounds\_check, 0);
25. SETGATE(idt[T\_ILLOP], 0, GD\_KT, illegal\_opcode, 0);
26. SETGATE(idt[T\_DEVICE], 0, GD\_KT, device\_not\_available, 0);
27. SETGATE(idt[T\_DBLFLT], 0, GD\_KT, double\_fault, 0);
28. SETGATE(idt[T\_TSS], 0, GD\_KT, invalid\_task\_switch\_segment, 0);
29. SETGATE(idt[T\_SEGNP], 0, GD\_KT, segment\_not\_present, 0);
30. SETGATE(idt[T\_STACK], 0, GD\_KT, stack\_fault, 0);
31. SETGATE(idt[T\_GPFLT], 0, GD\_KT, general\_protection\_fault, 0);
32. SETGATE(idt[T\_PGFLT], 0, GD\_KT, page\_fault, 0);
33. SETGATE(idt[T\_FPERR], 0, GD\_KT, floating\_point\_error, 0);
34. SETGATE(idt[T\_ALIGN], 0, GD\_KT, alignment\_check, 0);
35. SETGATE(idt[T\_MCHK], 0, GD\_KT, machine\_check, 0);
36. SETGATE(idt[T\_SIMDERR], 0, GD\_KT, SIMD\_floating\_point\_exception, 0);

问题：

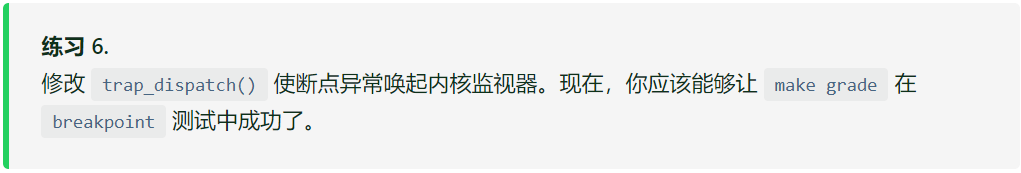
1. 对每一个中断/异常都分别给出中断处理函数的目的是什么？换句话说，如果所有的中断都交给同一个中断处理函数处理，现在我们实现的哪些功能就没办法实现了？

因为不同的异常/中断可能需要不同的处理方式，比如有些异常是代表指令有错误，则不会返回被中断的命令。而有些中断可能只是为了处理外部IO事件，此时执行完中断函数还要返回到被中断的程序中继续运行。

2.你有没有额外做什么事情让 user/softint 这个程序按预期运行？打分脚本希望它产生一个一般保护错(陷阱 13)，可是 softint 的代码却发送的是 int $14。为什么 这个产生了中断向量 13 ？如果内核允许 softint 的 int $14 指令去调用内核中断向量 14 所对应的的缺页处理函数，会发生什么？ 因为当前的系统正在运行在用户态下，特权级为3，而INT指令为系统指令，特权级为0。特权级为3的程序不能直接调用特权级为0的程序，会引发一个General Protection Exception，即trap 13。

练习5，6，7：







trap\_dispatch()处理缺页异常，断点异常，中断向量 T\_SYSCALL

1. **struct** PushRegs regs = tf->tf\_regs;
2. **switch** (tf->tf\_trapno) {
3. **case** T\_PGFLT:
4. page\_fault\_handler(tf);
5. **break**;
6. **case** T\_BRKPT:
7. monitor(tf);
8. **break**;
9. **case** T\_SYSCALL:
10. tf->tf\_regs.reg\_eax = syscall(regs.reg\_eax, regs.reg\_edx, regs.reg\_ecx, regs.reg\_ebx, regs.reg\_edi, regs.reg\_esi);
11. **break**;
12. **default**:
13. print\_trapframe(tf);
14. **if** (tf->tf\_cs == GD\_KT)
15. panic("unhandled trap in kernel");
16. **else** {
17. env\_destroy(curenv);
18. **return**;
19. }
20. }

其中T\_SYSCALL要在*trapentry.S和// trap.c* *// trap\_init()中定义*

1. // trapentry.S
2. TRAPHANDLER\_NOEC(system\_call, T\_SYSCALL)
4. // trap.c
5. // trap\_init()
6. **extern** **void** system\_call();
7. SETGATE(idt[T\_SYSCALL], 0, GD\_KT, system\_call, 3);

修改kern/syscall.c中的syscall()函数，使能处理定义在inc/syscall.h中的所有系统调用。

1. // syscall.c
2. // syscall()
3. **switch** (syscallno) {
4. **case** SYS\_cputs:
5. sys\_cputs((**const** **char** \*) a1, a2);
6. **return** 0;
7. **case** SYS\_cgetc:
8. **return** sys\_cgetc();
9. **case** SYS\_getenvid:
10. **return** sys\_getenvid();
11. **case** SYS\_env\_destroy:
12. **return** sys\_env\_destroy(a1);
13. **default**:
14. **return** -E\_NO\_SYS;
15. }

问题：

断点那个测试样例可能会生成一个断点异常，或者生成一个一般保护错，这取决你是怎样在 IDT 中初始化它的入口的（换句话说，你是怎样在 trap\_init 中调用 SETGATE 方法的）。为什么？你应该做什么才能让断点异常像上面所说的那样工作？怎样的错误配置会导致一般保护错？

如果在 SETGATE 中设置的优先级为 3 时，就会触发 break point exception；

当优先级设置为 0 时，则会触发 general protection exception。这是因为优先级

低的代码试图访问优先级高的部分时，就会触发 general protection exception。

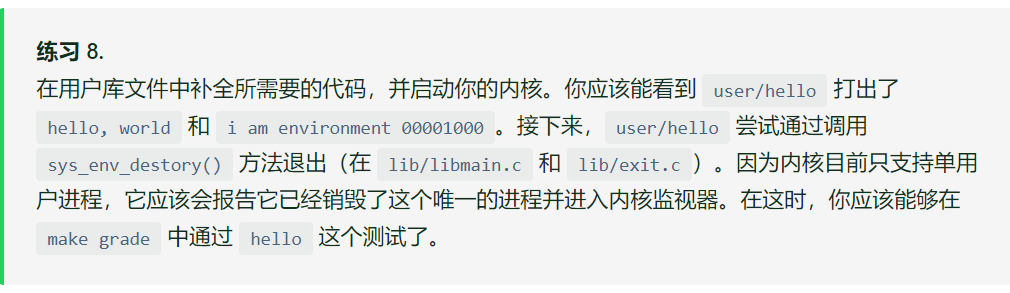
因为在系统中 break point 测试程序的优先级是 3，所以如果优先级为 3 则会触发

break point exception，优先级为 0 则是 general protection exception

你认为这样的机制意义是什么？尤其要想想测试程序 user/softint 的所作所为 / 尤其要考虑一下 user/softint 测试程序的行为。

在 user/softint.asm 文件中 asm volatile("int $14");本应调用的是 page fault（即 页面错误处理），但是由于优先级不够将触发 general protection exception。所以这个机制的重点就是在 SETGATE 中对每个异常设置的优先级。

练习8：



用户程序执行后都会走到lib/libmain.c中的libmain()，需要修改该函数初始化其中的thisenv;变量。

thisenv = &envs[ENVX(sys\_getenvid())];

练习 9：



由于 cs 寄存器的低 2 位的值与 CPL 相等，所以可以根据 cs 寄存器判断是否在内核态

1. // trap.c
2. // page\_fault\_handler()
3. **if** ((tf->tf\_cs & 3) == 0) {
4. panic("page fault in kern");
5. }

通过页表找到相应的 pte ，然后判断是否具有权限，这里需要记录第一个出错的虚拟地址，所以一开始不能将 va 对齐

1. // pmap.c
2. // user\_mem\_check()
3. **uintptr\_t** high = ROUNDUP((**uintptr\_t**) va + len, PGSIZE);
4. **for** (**uintptr\_t** low = (**uintptr\_t**) va; low < high; low = ROUNDUP(low + 1, PGSIZE)) {
5. pte\_t \*pte = pgdir\_walk(env->env\_pgdir, (**void** \*) low, **false**);
6. **if** (!pte || (~(\*pte) & perm) || low >= ULIM) {
7. user\_mem\_check\_addr = low;
8. **return** -E\_FAULT;
9. }
10. }
11. **return** 0;

利用刚才的函数检查这一段地址

1. // syscall.c
2. // sys\_cputs()
3. user\_mem\_assert(curenv, s, len, 0);