Lab3 文档

lab设计

Exercise 1. Modify `mem_init()` in `kern/pmap.c` to allocate and map the `envs` array. This array consists of exactly `NENV` instances of the `Env` structure allocated much like how you allocated the `pages` array. Also like the `pages` array, the memory backing `envs` should also be mapped user read-only at `UENVS` (defined in `inc/memlayout.h`) so user processes can read from this array.

在`mem_init()`中调用`boot_alloc()`分配一段大小为`NENV * sizeof(struct Env)`的空间给envs，并将这段内存初始化为0。`page_init()`之后，调用`boot_map_region()`将UENVS开始的一段PTSIZE空间映射到envs的物理地址上。

Exercise 2. In the file `env.c`, finish coding the following functions:

- `env_init()`

  从后往前遍历envs数组，将每一个元素的env_id置为0，env_status置为ENV_FREE。使用头插法将envs数组加入到env_free_list中。调用`env_init_percpu()`将不同的段分别设置优先级别0和优先级别3。

- `env_setup_vm()`

  调用`page_alloc(ALLOC_ZERO)`获得1个页作为环境的页目录。该页的`pp_ref`自增1，该环境的`env_pgdir`置为该页的虚拟地址。将UTOP以上的所有大页的PTE置为kern_pgdir上对应的PTE。将UVPT所对应的PTE置为环境的`env_pgdir`的物理地址，权限为user只读、kernel只读。

- `region_alloc()`

  将`va`与页大小向下对齐，`va+len`与页大小向上对齐。对应虚拟地址范围内的每一个页大小的空间，通过`page_alloc()`分配1个物理页，并通过调用`page_insert()`将该页插入环境的页表中，权限为kernel可读写，user可读写。

- `load_icode()`

  调用`lcr3(PADDR(e->env_pgdir))`，将环境的页表基址加载到CR3中，便于将ELF的内容加载到对应的虚拟地址上。通过检查elf->e_magic是否等于ELF_MAGIC判断elf是否有效。遍历ELF中的program header，对于ELF_PROG_LOAD类型的段，调用`region_alloc()`分配一段从`ph->p_va`开始的大小为`ph->p_memsz`的空间；调用`memmove`将`binary + ph->p_offset`开始的一段`ph->p_filesz`大小的内存复制到`ph->p_va`上；调用`memset`将这段空间的剩余部分初始化为0，作为bss段的内存空间。调用`region_alloc()`分配一段从`USTACKTOP - PGSIZE`开始的一段大小为 PGSIZE的空间，作为用户栈；调用`memset`将这段空间初始化为0。`e->env_tf.tf_esp = USTACKTOP`和`e->env_tf.tf_eip = elf->e_entry`设置了环境启动时的用户栈和程序入口点。

- `env_create()`

  调用`env_alloc`获得新环境，调用`load_icode`将binary加载到新环境中。将环境的`env_type`置为type，`env_parent_id`置为0

- `env_run()`

  检查curenv是否等于e。若等于，则不需要上下文切换，直接调用`env_pop_tf(&e->env_tf)`恢复当前环境的状态；若不等于，将curenv的环境状态设为ENV_RUNNABLE，将curenv置为e，并将e的状态置为ENV_RUNNING、`e->env_runs += 1`，通过调用`lcr3(PADDR(e->env_pgdir))`将页表切换为新环境的页表，最后调用`env_pop_tf(&e->env_tf)`恢复e的状态。

Exercise 4. Edit `trapentry.S` and `trap.c` and implement the features described above. The macros `TRAPHANDLER` and `TRAPHANDLER_NOEC` in `trapentry.S` should help you, as well as the T_* defines in `inc/trap.h`. You will need to add an entry point in `trapentry.S` (using those macros) for each trap defined in `inc/trap.h`, and you'll have to provide _alltraps which the `TRAPHANDLER` macros refer to. You will also need to modify `trap_init()` to initialize the `idt` to point to each of these entry points defined in `trapentry.S`; the `SETGATE` macro will be helpful here.

trapentry.S中，调用`TRAPHANDLER_NOEC`和`TRAPHANDLER`生成异常和中断的程序入口点。查阅80386手册可知，`T_DBLFLT`、`T_TSS`、`T_SEGNP`、`T_STACK`、`T_GPFLT`、`T_PGFLT`等trap需要由处理器将错误码入栈，由`TRAPHANDLER`生成入口点；其他trap不需要错误码入栈，由`TRAPHANDLER_NOEC`生成入口点。由Trapframe的结构体可知，在栈上从高地址到低地址依次需要保存`%ds`、`%es`、8个通用寄存器的值。`%ds`和`%es`之前各有2字节的padding。因此通过如下指令可以在栈上构造出一个Trapframe结构体用于保存user环境状态

```
pushl %ds
pushl %es
pushal
```

通过`MOV`指令，将`GD_KD`存入`%es`、`%ds`。最后调用`trap`。

在trap.c中，对trapentry.S中所有trap处理程序的函数名进行声明，用于填IDT。在`trap_init()`中填IDT。除了

`T_NMI`需要interrupt gate外，其他trap传istrap参数为1。`Code segment selector`参数为`GD_KT`，`offset`参数为各个handler的函数名，即函数入口点的位置，`Descriptor Privilege Level`参数为0，即只有ring0级别可以通过`int`调用该处理程序。

Exercise 5. Modify `trap_dispatch()` to dispatch page fault exceptions to `page_fault_handler()`.

在`trap_dispatch()`中，若传入的trapframe的`tf_trapno`为`T_PGFLT`，调用`page_fault_handler()`，然后返回。

Exercise 6. Modify `trap_dispatch()` to make breakpoint exceptions invoke the kernel monitor. You should now be able to get make grade to succeed on the `breakpoint` test.

在`trap_dispatch()`中，若传入的trapframe的`tf_trapno`为`T_BRKPT`，调用`monitor()`并传入trapframe，然后返回。

Exercise 7. Add a handler in the kernel for interrupt vector `T_SYSCALL`. You will have to edit `kern/trapentry.S` and `kern/trap.c`'s `trap_init()`. You also need to change `trap_dispatch()` to handle the system call interrupt by calling `syscall()` (defined in `kern/syscall.c`) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in `%eax`. Finally, you need to implement `syscall()` in `kern/syscall.c`. Make sure `syscall()` returns `-E_INVAL` if the system call number is invalid. You should read and understand `lib/syscall.c` (especially the inline assembly routine) in order to confirm your understanding of the system call interface. Handle all the system calls listed in `inc/syscall.h` by invoking the corresponding kernel function for each call.

在trapentry.S中，调用`TRAPENTRY_NOEC()`生成syscall的处理程序入口点。在trap.c中，声明syscall handler，并在`trap_init()`中将处理程序填入IDT，权限为ring3。在`trap_dispatch()`中，若trapframe的trapno为`T_SYSCALL`，则从trapframe中读出user传入`%eax`,`%edx`,`%ecx`,`%ebx`,`%edi`,`%esi`的值，作为`syscall()`的参数传入。通过以下指令将返回值存入`%eax`

```
asm volatile("movl %0,%%eax" : : "r" (ret));
```

Exercise 8. Implement system calls using the `sysenter` and `sysexit` instructions instead of using `int 0x30` and `iret`.

在inc/x86.h添加了以下函数和声明

```
#define IA32_SYSENTER_CS  0x174
#define IA32_SYSENTER_ESP 0x175
#define IA32_SYSENTER_EIP 0x176
static inline void
wrmsr(uint32_t msr, uint32_t val1, uint32_t val2)
{
  asm volatile("wrmsr"
       :
       : "c" (msr), "a" (val1), "d" (val2));
}
```

在kern/init.c的`i386_init()`中，调用`wrmsr()`将`IA32_SYSENTER_CS`设置为`GD_KT`，将`IA32_SYSENTER_CS + 8`设置为`GD_KD`，将`IA32_SYSENTER_EIP`设置为`sysenter_handler`的地址，将`IA32_SYSENTER_ESP`设置为`KSTACKTOP`。由Intel手册可知，user态调用sysenter之后，会在MSR中读取对应的值到`esp`、`eip`等寄存器中，跳转到sysenter_handler的位置开始执行。

在kern/trapentry.S中，添加sysenter_handler的实现如下：

- 将ds寄存器和es寄存器压到栈上上，用于保护用户数据段
- 将user态传入的ebp寄存器压到栈上，用于保护返回时的esp值
- 将user态传入的esi寄存器压到栈上，用于保护返回时的eip值
- 按照`%edi`,`%ebx`,`%ecx`,`%edx`,`%eax`的顺序依次压栈，作为syscall的五个参数值
- 将ds寄存器和es寄存器置为内核数据段，即`GD_KD`
- 调用syscall（syscall的参数修改为1个syscallno和4个user传入的参数）
- 将esp自增20，使得参数出栈
- 栈顶值出栈，赋给edx寄存器，用于sysexit恢复eip值
- 栈顶值出栈，赋给ecx寄存器，用于sysexit恢复esp值
- 栈顶值出栈，依次赋给es寄存器和ds寄存器，用于恢复用户数据段
- 调用`sysexit`

lib/syscall.c中，user态调用sysenter的方法如下：

- 将ebp压栈，保护ebp的值
- 将esp移入ebp
- `leal after_sysenter_label, %%esi`
- 调用sysenter
- 恢复ebp的值

Exercise 9. Add the required code to the user library, then boot your kernel. You should see `user/hello` print "`hello, world`" and then print "`i am environment 00001000`".

在`libmain()`中调用`sys_getenvid()`获得当前环境的id。以`ENVX(id)`为索引，在envs数组中获得当前环境的`struct Env`，存于thisenv中。

Exercise 10. You need to write syscall *sbrk*.

在`struct Env`中添加变量env_brk用于存储环境的break。在环境load_icode时，检查程序代码和数据的最高地址，取出大于该地址并与PGSIZE对其的地址作为环境的初始env_brk。

在`sys_sbrk(inc)`中，对`[curenv->env_brk, ROUNDUP(curenv->env_brk + inc))`范围里的所有页进行`page_alloc()`和`page_insert()`完成内存空间的分配。以`ROUNDUP(curenv->env_brk + inc))`作为增加后的`env_brk`，存入curenv结构体中，并返回该值。

Exercise 11. Change `kern/trap.c` to panic if a page fault happens in kernel mode.

Read `user_mem_assert` in `kern/pmap.c` and implement `user_mem_check` in that same file.

Change `kern/syscall.c` to sanity check arguments to system calls.

Finally, change `debuginfo_eip` in `kern/kdebug.c` to call `user_mem_check` on `usd`, `stabs`, and `stabstr`. If you now run `user/breakpoint`, you should be able to run backtrace from the kernel monitor and see the backtrace traverse into `lib/libmain.c` before the kernel panics with a page fault. What causes this page fault? You don't need to fix it, but you should understand why it

happens.

在kern/trap.c的`page_fault_handler()`中，检查`tf->tf_cs & 3`，即privilege level。若该值为3，则为user态；若该值为0，则为内核态，调用`panic()`。

在`user_mem_check()`中，对`[ROUNDDOWN((uintptr_t)va, PGSIZE), ROUNDUP((uintptr_t)va + len, PGSIZE))`范围内的所有页，检查虚拟地址是否大于`ULIM`，并调用`pgdir_walk()`获得PTE，检查权限以及页是否存在。

在syscall.c的`sys_cputs()`中，调用`user_mem_assert(curenv, s, len, 0)`检查user是否能够访问该段内存。

在kdebug.c中，调用`user_mem_check()`对usd、stabs、stabstr三段内存进行检查。

backtrace时，最终会触发kernel panic的原因是，user环境最终能回溯到程序的_start标签处。此时在esp+8处获得的ebp值是无效的，因此通过该ebp访存会导致缺页异常。由于backtrace最终检查到kernel的入口处`ebp==0`才会停止，而user环境进入前不会保存kernel的指令地址，因此无法从user的libmain回溯到kernel的entry处。

Exercise 12. Boot your kernel, running `user/evilhello`. The environment should be destroyed, and the kernel should not panic.

在syscall.c的`sys_cputs()`中，调用`user_mem_assert(curenv, s, len, 0)`检查到user没有访问该段内存的权限，报错并将该环境破坏。

Exercise 13. `evilhello2.c` want to perform some privileged operations in function `evil()`. Function `ring0_call()` takes an function pointer as argument. It calls the provided function pointer with ring0 privilege and then return to ring3. There's few ways to achieve it. You should follow the instructions in the comments to enter ring0.

1. Store the GDT descripter to memory (sgdt instruction)
2. Map GDT in user space (sys_map_kernel_page)
3. Setup a CALLGATE in GDT (SETCALLGATE macro)
4. Enter ring0 (lcall instruction)
5. Call the function pointer
6. Recover GDT entry modified in step 3 (if any)
7. Leave ring0 (lret instruction): 将`call_fun_ptr()`中`asm volatile("popl %ebp")`改为`asm volatile("leave")`，用于恢复ebp值和esp值，否则lret后esp值会导致程序处罚general protection