OSLab3 文档

陈俊 5140379064

一、envs 相关

首先，修改 mem_init，在其中将 envs 物理地址映射到 NENVS 上。之后，就是实现 envs 的相关函数。

在 envs_init 函数中，逆序往 free list 头上添加 env，保证 envs 和 free_list 的顺序一致，直至 envs 中的所有项都加入 free_list 中。

之后，实现 env_setup_vm，主要是新建一个 page，作为 env 专属的页表，将 kern_pgdir 的内容复制过来，并且将 UVPT 换成自身，这样，env 就有了自己独立的地址空间。

Region_alloc 用于虚拟地址的分配，可以根据 lens alloc page，再通过 page_insert 插入页表中，实现地址的分配。

Load_icode 是将可执行文件进行导入，代码部分主要参考了 boot 中 main.c 的文件的导入，将文件写入内存。在写入前要切换页表，保证写入的内容按照当前 env 的页表进行地址翻译，并且再导入后切回 kern_pgdir。同时，给用户分配初始的栈。

Env_create 主要就是调用了 env_alloc，并且 load_icode，以完成完整的 env 的创建。

Env_run 为进程的运行。按照上方注释的说明，先将原先的 env 的状态设为 RUNNABLE，并将传入的 e 设为当前的 curenv，状态设为 RUNNING，runs 次数加 1，换成当前 env 的页表，并 pop_tf 保存当前的 trapframe。

二、trap 相关

按照 Intel 的相关规范，针对每个中断和异常，创建相应的 handler，并根据中断和异常的类型，判断是否要压入 error code，并压入相应的 trapno。由于提供了相应的宏，因此针对每个中断异常分别调用不同的宏即可。之后，都会进入同一个入口，alltraps。按照说明，先构造相应的 trapframe。由于 processor 会帮我们压入一系列参数，我们在 alltraps 里只需压入 ds、es 和其他 regs。通过 pushal，可以压入通用寄存器。之后，按照提示，将 ds 和 es 改为 GD_KD，之后，push %esp,作为 trapframe 的指针，当作参数压入栈中，之后调用 trap。在 trap.c 的 trap_init 中，extern 所有的 handler，并且调用 SETGATE，在 idt 中注册，这样就可以实现 trap 的调用。

trap 函数会调用 trap_dispatch，由于传入了 tf 参数，可以根据 tf 参数中的 trapno，判断如果是 page fault，调用相应的 page_fault_handler，就可以处理 page fault。

三、systemCall 相关

在最开始，我先按照 lab 文档中的提示，在 init.c 中添加了 wrmsr 语句，并且在 x86.h 中添加了 wrmsr 的实现函数。当用户调用 syscall 时，首先调用 lib/syscall.c 中的相关函数,然后从 inline 的 syscall 中调用 sysenter。调用之前，先按照格式将参数存储到相应寄存器中，将 esp 存入 ebp，并且将 eip 存入 esi（通过标号实现）。由于在 init.c 中进行了 handler 与 sysenter 的绑定，因此 sysenter 后，进入 sysenter_handler 函数，在这个函数中，我借用了 trapframe 的结构，将寄存器压入栈中，组成一个 trapframe，传给 syscall_helper 函数，这个函数我在 trap.c 中实现，作为一层包装，调用 kern/syscall.c 函数，当函数返回后，回到了 sysenter_handler,由于 eax 中存放着返回值，不应被覆盖，因此我没有用 popal，而是使用一个个 pop，保留了 eax 中的返回值。同时，由于 intel 的规定，sysexit 前，要将 eip（esi）的值放入 edx，而将 esp（ebp）放入 ecx，这样才能正确的返回。同时，在 kern/syscall.c 的 syscall 函数中，针对不同的系统调用，调用不同

的函数，以实现不同的功能。这样，就实现了系统调用。

四、sbrk

根据 memlayout.h，我们可以得知，Program data 和 heap 处于同一个区域。因此，我可以用代码之上的地址，进行堆的实现。从 load_icode 函数中，我们可以获得用户代码段的最高地址，用参数 break 记录在 env 中，这之上的虚拟地址可以用于堆的分配。通过 page_alloc 和 page_insert（类似于 region_alloc），我们可以实现虚拟地址的分配。与此同时，将 break 设为分配地址的末尾的地址，保存在 env 中，并作为返回值返回，这就实现了 sbrk。

五、breakpoint

这部分代码实现了 breakpoint 的相关调试。由于在 ring3 时能够触发 breakpoint，因此在 SETGATE 中需要将 breakpoint 的 dpl 设为 3。当用户触发 checkpoint，最终会进入 trap 函数，并进入 trap_dispatch 函数，因此我们在这个函数中判断 trapno，如果是 checkpoint 异常，就调用 monitor 函数，并传入相应的 trapframe。同时，我在 monitor.c 中实现了三个 command,分别为 x、si 以及 c，这些只有在 trapframe 不为空的时候可以被调用用于调试。x 比较简单，直接打印该虚拟地址中的值即可。

si 的实现需要用到 EFLAGS 中的 TrapFlag，位于 0x100 位。当该位置为 1 时，开启单步调试，当执行完一条语句时，触发 debug 异常。在这样的前提下，我们先打印 eip，以及 eipdebuginfo，然后修改 trapframe，将 TF 位置为 1，然后保存 tf 到 env 中，调用 env_run，继续运行。由于 TF 被置为 1，因此运行完一条语句后，触发 DEBUG 异常。因此，在 trap_dispatch 中，我也加入了 debug 异常的判定，如果触发 debug 异常，修改 TF 位为 0，再调用 monitor 函数。这样，就实现了 si。

c 类似于 si，不过更简单。直接保存 tf 进 env，调用 env_run 即可。

六、user_mem_check

在调用 syscall.c 中的 sys_cputs 时，要判断那段地址用户是否有权限访问。这里就需要用到 user_mem_check。对从 ROUNDDOWN（va，PGSIZE）开始到 ROUNDUP（va+len,PGSIZE）这些虚拟地址，如果这些地址中有地址大于等于 ULIM，那么显然不能访问。如果小于，则根据 walk_pgdir 来通过页表获取相应的权限，如果 pte 不存在，或者 pte 的 PTE_P 为为 0，或者不具有相应权限，则无法访问，并将出错的地址放在 user_mem_check_addr。这样，就实现了检查权限。

七、获得 RING0 权限

这里，我需要在用户态，获得 ring0 权限。首先，通过 sgdt，获得 gdt 的 base 地址。其次，通过系统调用中的 sys_map_kernel_page 函数，将 gdt 所在的物理页映射到我提供的虚拟地址上。由于映射的是页，因此存在着一定的偏移。由于 va 不一定是 PGSIZE 对齐，因此我们需要将其 PGOFF 设为 0，并加上 base 地址的 PGOFF，这样获得的地址，才是 gdt 的地址。这之后，需要 GDT 中的用户的一段（我选择了 GD_UT）修改为 CallGateDescriptor，并将原先的 SEG Descriptor 保存起来，用于恢复。因此，在找到 GD_UT 对应的 SEG 的地址后，调用 SETCALLGATE，在 GDT 中存入调用门描述符，并指向一个 wrapper 函数。之后，使用 lcall 指令，调用刚才修改的 CallGateDescriptor。在 wrapper 函数中，调用 evil，并恢复 gdt，pop ebp，再调用 lret，回到原先的函数。这样，在 wrapper 函数中，就以 RING0 的状态，调用了 evil 函数。