JOS_Lab4_Report

姓名：方嘉聪，学号：2200017849

I. Challenge

Challenge 5: Extend kernel to *all* types of processor exceptions in user mode

模仿 `pgfault` 的相关实现，在这里我实现了在用户态下对 `divide by zero`, `illegal operation` 和 `general protection fault` 的相关处理函数。注意到这三个的实现是类似的，

- 在 `kern/trap.c:trap_dispatch()` 中添加相关的case，并实现相应的handler
- 在 `lib/` 下添加 `zerodiv.c, illop.c, gpflt.c`，并实现相应的汇编代码 `xxentry.S`
- 在 `inc/` 下的相关文件添加 `set_xxx_upcall` 对应的 system call numbers， 并实现对应的系统调用 `kern/syscall.c:sys_env_set_xxx_upcall()`。
- 模仿 `faultdie.c` 实现了 `faultdivzero.c, faultillop.c, faultgpflt.c`， 这里的handler都是在 fault 时直接退出。
  值得注意的是，需要修改对应的 `lib/Makefrag, kern/Makefrag`，使得我们增加的文件和用户程序可以正常编译与链接。
  结果见下：

```
Booting from Hard Disk..
6828 decimal is 15254 octal!
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 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
Illegal Opcode at 0x00800071
i faulted at va 0, err 0
[00001000] exiting gracefully
[00001000] free env 00001000
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

```
Booting from Hard Disk..
6828 decimal is 15254 octal!
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 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
Divide by zero at 0x0080007f
i faulted at va 0, err 0
[00001000] exiting gracefully
[00001000] free env 00001000
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

> ⑦ 对于 general protection fault
>
> 可以像 lab3 中改为 `SETGATE(idt[T_BRKPT], 0, GD_KT, &handler_brkpt, 0);`，而后通过 `int3` 触发，但是同时会触发 page fault，不清楚具体的原因。

Challenge 5: `sfork()`

实现一个 `fork()` 函数，使得父子进程之间共享除了运行栈以外的内存页。

- 首先实现一个 `sduppage()` 函数，类似 `duppage` 但是不修改权限位，而是直接将父进程中的页表直接复制映射到子进程中，不采用 Copy-on-Write。
- 此外对 `fork()` 中父子进程内存页映射的部分进行修改，从栈底开始复制（或者说共享内存页）
  - 对于栈的部分使用 `duppage` 进行 COW 复制，不共享。
  - 对于其他内存区域使用 `sduppage` 进行内存页共享
  注意到由于 `thisenv` 是全局变量，存放在 `.text` 段，因此使用 `sfork()`，父子进程会共享 `thisenv`，也就意味着父进程的 `thisenv` 会被子进程意外修改，为了使 `sfork()` 正常运行，我们需要修改 `thisenv` 的实现。

1. 一个最简单的想法是将每个调用 `thisenv` 的地方用 `sysgetenvid()` 替代，但是这需要改动内核中太多的代码。
2. 因此在最终实现中采用 定义同名的宏的方式解决这一问题，在 `inc/lib.h` 中添加

```
extern const volatile struct Env *penv[NENV];
```

```
#define thisenv (penv[ENVX(sys_getenvid())])
```

在 `lib\libmain.c` 中声明 `penv` 的定义 `const volatile struct Env *penv[NENV];`。`pingpongs.c` 的输出结果见下：

```
[00000000] new env 00001000
[00001000] new env 00001001
i am 00001000; thisenv is 0xeec00000
send 0 from 1000 to 1001
1001 got 0 from 1000 (thisenv is 0xeec00088 1001)
1000 got 1 from 1001 (thisenv is 0xeec00000 1000)
1001 got 2 from 1000 (thisenv is 0xeec00088 1001)
1000 got 3 from 1001 (thisenv is 0xeec00000 1000)
1001 got 4 from 1000 (thisenv is 0xeec00088 1001)
1000 got 5 from 1001 (thisenv is 0xeec00000 1000)
1001 got 6 from 1000 (thisenv is 0xeec00088 1001)
1000 got 7 from 1001 (thisenv is 0xeec00000 1000)
1001 got 8 from 1000 (thisenv is 0xeec00088 1001)
1000 got 9 from 1001 (thisenv is 0xeec00000 1000)
[00001000] exiting gracefully
[00001000] free env 00001000
1001 got 10 from 1000 (thisenv is 0xeec00088 1001)
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Part A: Multiprocessor Support and Cooperative Multitasking

Exercise 1&2

比较容易，按照指导修改即可。

Question 1

1. AP在实模式中运行，不激活 `A20`，此时只能寻址低20位。`boot_aps()` 将 AP entry code `(kern/mpentry.S)` 拷贝到实模式下可以运行的低地址 `MPENTRY_PADDR`。故需要利用

   ```
   #define MPBOOTPHYS(s) ((s) - mpentry_start + MPENTRY_PADDR)
   ```

   计算符号(symbols)对应的低地址下的物理地址，使得 AP 能够在实模式下正常访问这些代码。
2. 对于 `boot/boot.S` 中的 bootloader 而言，使用 linker 得到对应的 symbol 变量的地址(相对于 `0x7c00`)
3. 如果不使用 `MPBOOTPHYS`，那么AP会尝试访问实模式下不可访问的高地址。

Exercise 3&4

比较容易，整体按照文档引导即可。

对于 Exercise 4， 通过 `thiscpu->cpu_id` 获得当前CPU id，而后更新 `thiscpu->cpu_ts` 的对应内通即可（主要是要替换lab3中使用的全局变量 `ts`）

Exercise 5

按照文档引导和代码中的注释提示，上锁和解锁即可。

Question 2

例如，CPU0正在 kernel mode 处理一个trap，但此时 CPU1 出发了一个 trap 将对应的 `TrapFrame` 压入共享的栈中，此时尽管有一个 big kernel lock，当 CPU0完成 trap 处理后返回 user mode 时会读取到错误的 `TrapFrame`（在这里是 CPU1 的）

Exercise 6

按照文档引导和代码注释实现即可，值得注意的是每实现一个 syscall 都需要在 `kern/syscall.c:syscall()` 中添加相应的 dispatch 的 case，以正确处理。

Question 3

注意到所有进程的地址空间中内核部分的代码 `UTOP~UVPT` 是相同的。

Question 4

这是由于每个进程没有单独的内核栈，需要在上下文切换前将寄存器等信息存储到PCB等结构中。具体地在JOS中，当触发 `sys_yield()` 的系统调用后会将寄存器压入栈中，而后 trap handler `kern/trap.c/trap()` 会将栈上的 `TrapFrame` 复制到 `curenv->env_tf` 中。当 `env_run()` 结束后 `env_pop_tf()` 会恢复这些信息。

Exercise 7

按照提示一次实现这些系统调用，注意在 `sys_page_map()` 等系统调用中需要注意各种边界 error（特别是权限的检查）。

Part B: Copy-on-Write Fork

Exercise 8 & 9:

按照指导实现 `sys_env_set_pgfault_upcall` 和 `page_fault_handler` 即可。
注意为了处理递归的页故障，如果已经在异常栈中则需要额外插入 32bits 的 0

```
if (tf->tf_esp >= UXSTACKTOP - PGSIZE && tf->tf_esp < UXSTACKTOP){
        esp = (tf->tf_esp - sizeof(struct UTrapframe) - 4);
        *(uint32_t *)(tf->tf_esp - 4) = 0; // push a 32-bit empty word
}
```

Exercise 10:

相当有挑战性的一题，需要分析在当前 `TrapFrame` 下栈上的内容，见下

```
// After `_pgfault_upcall` is called, the stack looks like this:
//
//    Previous Frame                User Trap Frame
// +------------------+            +------------------+
// | stack data       |      +---- | trap-time esp-4  |
// | ...              |      |     | trap-time eflags |
// | trap-time eip    |      |     | trap-time eip    | (*)
// +------------------+ <----+     | trap-time eax    | <-- utf_regs end
//                               | trap-time ecx    | .
//                               | trap-time edx    | .
//                               | trap-time ebx    | .
//                               | trap-time esp    | .
//                               | trap-time ebp    | .
//                               | trap-time esi    | .
//                               | trap-time edi    | <-- utf_regs start
//                               | tf_err           |
//                               | fault_va         |
//                               +------------------+  <-- %esp
```

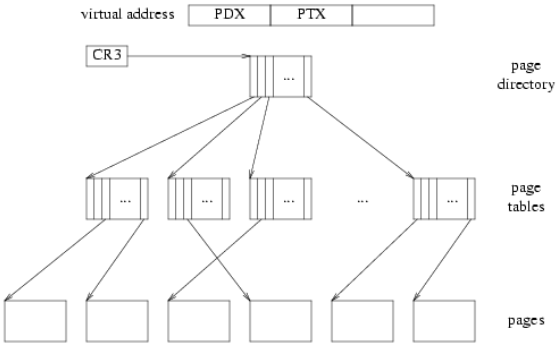
首先需要在栈上读取出应当恢复的运行地址，push 到上一帧的对应地址中，而后一次恢复寄存器，trap-time eflags 等等，可见 `pfentry.S` 中我写的注释。

Exercise 11:

这是容易的，按照指导实现 `lib/pgfault.c:set_pgfault_handler()` 即可

Exercise 12:

用到了页表自映射，见下图



在JOS会用到下面的数组

```
// uvpd: pointer to the current page directory
// uvpd[i]: the page directory entry for the page table that maps virtual address i
// uvpt: pointer to the page table
// uvpt[i]: the page table entry for page i.
```

这里值得注意的是，我们不能对父进程中 `addr = UTOP-PGSIZE` 的页表，复制映射到子进程，因为该地址指向异常栈，我们需要额外为子进程的异常栈分配一个新页。
否则会导致写权限错误，无法通过`forktree.c`

Part C: Preemptive Multitasking and Inter-Process communication (IPC)

Exercise 13&14

按照指导完成即可，注意IRQs的 traphandler 不会插入错误码，因此要使用`

```
TRAPHANDLER_NOEC(handler_irq_timer, IRQ_OFFSET + IRQ_TIMER);
```

Exercise 15

按照要求实现 `sys_ipc_recv, sys_ipc_try_send` 和相应的封装函数。这是新的系统调用，主要要在`kern/syscall.c:syscall()` 中添加相应的 dispatch 的 case，以正确处理。
此外在封装函数 `ipc_send()` 中需要循环发送 ipc_sends 直到超时或成功recv。而send 和 recv之间的相互等待通过设置`recv`下的对应环境是否 `env->env_status = ENV_RUNNABLE;` 来实现。

Results

顺利通过所有测试，`make grade` 结果见下

```
make[1]: Leaving directory '/home/ubuntu/6.828/lab'
dumbfork: OK (1.1s)
Part A score: 5/5

faultread: OK (0.9s)
faultwrite: OK (1.0s)
faultdie: OK (1.0s)
faultregs: OK (1.0s)
faultalloc: OK (0.9s)
faultallocbad: OK (1.0s)
faultnostack: OK (1.0s)
faultbadhandler: OK (0.9s)
faultevilhandler: OK (1.0s)
forktree: OK (1.1s)
Part B score: 50/50

spin: OK (1.0s)
stresssched: OK (1.6s)
sendpage: OK (0.6s)
pingpong: OK (0.8s)
primes: OK (7.6s)
Part C score: 25/25

Score: 80/80
ubuntu:lab/ (lab4_new*) $
```