Lab #4

陈威宇

Exercise 1

`mmio_map_region` 将 `pa` 开始的 `size` 个字节 map 到 MMIO region.

```
void *
mmio_map_region(physaddr_t pa, size_t size)
{
  static uintptr_t base = MMIOBASE;

  int mo = pa%PGSIZE;
  size+=mo;
  pa-=mo;
  if (size%PGSIZE!=0)
    size=size-size%PGSIZE+PGSIZE;
  if (base + size>MMIOBASE+0x400000)
    panic("MMIO region is not enough!!!");
  boot_map_region(kern_pgdir, base, size, pa, PTE_PCD|PTE_PWT|PTE_W);
  uint32_t t = base + mo;
  base += size;
  return (void *)t;
}
```

Exercise 2

修改了 `page_init` 使得 `MPENTRY_PADDR` 开始的一个页不是 free page.

```
void
page_init(void)
{
  page_free_list=NULL;
  size_t i;
  for (i = 0; i < npages; i++) {
    pages[i].pp_ref = 0;
    if (i==0 || (i*PGSIZE >= IOPHYSMEM && i*PGSIZE < EXTPHYSMEM) || (i*PGSIZE>=0x100000 && i*P
      pages[i].pp_link = NULL;
    }
    else{
      pages[i].pp_link = page_free_list;
```

```
      page_free_list = &pages[i];
    }
  }
}

```

Questions after Exercise 2

1

因为 mpentry.S 是在高地址链接的 (KERNBASE 以上), 但是被 `bootaps` load 到低地址 (`MPENTRY_ADDR`), 所以要用 `MPBOOTPHYS` 求物理地址.

Exercise 3

`mem_init_mp` 给每个 CPU 都 map 了一个 kernel stack.

```
static void
mem_init_mp(void)
{
  for (int i=0;i<NCPU;++i){
    uintptr_t kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
    boot_map_region(kern_pgdir, kstacktop_i-KSTKSIZE, KSTKSIZE, PADDR(percpu_kstacks[i]), PTE_
  }
}
```

Exercise 4

`trap_init_percpu` 对每个 CPU 初始化了 TSS and IDT.

```
void
trap_init_percpu(void)
{
/*
  ts.ts_esp0 = KSTACKTOP;
  ts.ts_ss0 = GD_KD;
  ts.ts_iomb = sizeof(struct Taskstate);
*/
  (thiscpu->cpu_ts.ts_esp0) = KSTACKTOP - (thiscpu->cpu_id) * (KSTKSIZE + KSTKGAP);
  (thiscpu->cpu_ts.ts_ss0) = GD_KD;
  (thiscpu->cpu_ts.ts_iomb) = sizeof(struct Taskstate);
  // Initialize the TSS slot of the gdt.
```

```
/*
  gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
          sizeof(struct Taskstate) - 1, 0);
  gdt[GD_TSS0 >> 3].sd_s = 0;
*/
  gdt[(GD_TSS0 >> 3)+cpunum()] = SEG16(STS_T32A, (uint32_t) (&(thiscpu->cpu_ts)),
          sizeof(struct Taskstate) - 1, 0);
  gdt[(GD_TSS0 >> 3)+cpunum()].sd_s = 0;

  // Load the TSS selector (like other segment selectors, the
  // bottom three bits are special; we leave them 0)
  ltr(GD_TSS0 + ((cpunum())<<3));

  // Load the IDT
  lidt(&idt_pd);
}
```

Exercise 5

按文档指示在对应位置加上 kernel lock 的语句就行了.

Questions after Exercise 5

2

因为发生异常或中断时，硬件会把寄存器状态放到 kernel stack 上，这是不需要 kernel lock 的.

Exercise 6

`sched_yield` 实现了 round-robin scheduling.

```
void
sched_yield(void)
{
  struct Env *idle;

  int id;
  if (curenv==NULL)
    id=-1;
  else
    id = ENVX(curenv->env_id);
```

```
  int h = id;
  while (1){
    ++h;
    if (h==id)
      break;
    if (h>=NENV){
      if (id<=0)
        break;
      h=0;
    }
    if (envs[h].env_status == ENV_RUNNABLE){
      env_run(&envs[h]);
    }

  }
  if (id>=0 && envs[id].env_status == ENV_RUNNING)
    env_run(&envs[id]);

  // sched_halt never returns
  sched_halt();
}
```

Questions after Exercise 6

3

虽然换了一个 env，但是它们的虚拟内存的 `UTOP` 以上部分是一样的，所以不受影响.

4

需要保存旧的 `env` 的寄存器等，为了之后接着运行的时候能运行. 在 `trap` 函数里的时候存到了 `env_tf`, 之后 `env_pop_tf()` 的时候又放回了 kernel stack.

Exercise 7

`sys_exofork` 分配了一个新的 environment.

```
static envid_t
sys_exofork(void)
{
  struct Env * t;
  int e = env_alloc(&t, curenv->env_id);
```

```
  if (e<0) return e;
  (t->env_status) = ENV_NOT_RUNNABLE;
  (t->env_tf) = curenv->env_tf;
  t->env_tf.tf_regs.reg_eax = 0;
  return t->env_id;
}
```

sys_env_set_status 给一个 environment 设置了 status.

```
static int
sys_env_set_status(envid_t envid, int status)
{
  if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
    return -E_INVAL;
  struct Env * t;
  int e = envid2env(envid, &t, 1);
  if (e<0)
    return e;
  t->env_status = status;
  return 0;
}
```

sys_page_alloc 分配了一页物理页给一个 env 的 va 开始的虚拟页来映射.

```
static int
sys_page_alloc(envid_t envid, void *va, int perm)
{
  if ((uint32_t)va >= UTOP || (uint32_t)va%PGSIZE!=0)
    return -E_INVAL;
  if ( (perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P) || (((perm ^ (PTE_U | PTE_P)) | (PTE_AVAIL
    return -E_INVAL;
  struct Env * t;
  int e = envid2env(envid, &t, 1);
  if (e<0)
    return e;
  struct PageInfo * pp = page_alloc(1);
  if (pp == NULL)
    return -E_NO_MEM;
  e = page_insert(t->env_pgdir, pp, va, perm);
  if (e<0)
    return e;
  return 0;
}
```

sys_page_map 把一个 page mapping 从一个 env 复制到另一个 env.

```
static int
sys_page_map(envid_t srcenvid, void *srcva,
       envid_t dstenvid, void *dstva, int perm)
{
  if ((uint32_t)srcva >= UTOP || (uint32_t)srcva%PGSIZE!=0 || (uint32_t)dstva >= UTOP || (uint
    return -E_INVAL;
  if ( (perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P) || (((perm ^ (PTE_U | PTE_P)) | (PTE_AVAIL
    return -E_INVAL;
  struct Env * srcenv;
  int e;
  e = envid2env(srcenvid, &srcenv, 1);
  if (e<0)
    return e;
  struct Env * dstenv;
  e = envid2env(dstenvid, &dstenv, 1);
  if (e<0)
    return e;
  pte_t * pt;
  struct PageInfo * pp = page_lookup(srcenv->env_pgdir, srcva, &pt);
  if (pp == NULL)
    return -E_INVAL;
  if (((*pt)&PTE_W)==0 && (perm & PTE_W))
    return -E_INVAL;
  e = page_insert(dstenv->env_pgdir, pp, dstva, perm);
  if (e<0)
    return e;
  return 0;
}
```

sys_page_unmap 把一个 env 的虚拟地址 va 开始的页 unmap 了.

```
static int
sys_page_unmap(envid_t envid, void *va)
{
  if ((uint32_t)va >= UTOP || (uint32_t)va%PGSIZE != 0)
    return -E_INVAL;
  struct Env * env;
  int e = envid2env(envid, &env, 1);
  if (e<0)
    return e;
```

```
  page_remove(env->env_pgdir, va);
  return 0;
}
```

Exercise 8

`sys_env_set_pgfault_upcall` 对一个 env 设置了 page fault upcall.

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
  // LAB 4: Your code here.
  struct Env * env;
  int e = envid2env(envid, &env, 1);
  if (e<0)
    return e;
  env->env_pgfault_upcall = func;
  return 0;
}
```

Exercise 9

`page_fault_handler` 在 user exception stack 上放了个 user trap frame, 然后去跑这个 env，从 page fault upcall 的位置开始跑，让 env 解决 page fault.

```
void
page_fault_handler(struct Trapframe *tf)
{
  uint32_t fault_va;

  // Read processor's CR2 register to find the faulting address
  fault_va = rcr2();

  // Handle kernel-mode page faults.

  // LAB 3: Your code here.
  if ((tf->tf_cs & 3) == 0) {
    panic("kernel-mode page fault!!!");
  }

  // LAB 4: Your code here.
```

```
  if ((curenv->env_pgfault_upcall) != NULL){
    struct UTrapframe *utf;

        if (tf->tf_esp >= UXSTACKTOP - PGSIZE && tf->tf_esp < UXSTACKTOP) {
            *(uint32_t *)(tf->tf_esp - 4) = 0;
            utf = (struct UTrapframe *)(tf->tf_esp - 4 - sizeof(struct UTrapframe));
        } else {
            utf = (struct UTrapframe *)(UXSTACKTOP - sizeof(struct UTrapframe));
        }

        user_mem_assert(curenv, (void *)utf, sizeof(struct UTrapframe), PTE_W | PTE_U);

        utf->utf_esp = tf->tf_esp;
        utf->utf_eflags = tf->tf_eflags;
        utf->utf_eip = tf->tf_eip;
        utf->utf_regs = tf->tf_regs;
        utf->utf_err = tf->tf_err;
        utf->utf_fault_va = fault_va;

        tf->tf_esp = (uint32_t)utf;
        tf->tf_eip = (uint32_t)curenv->env_pgfault_upcall;
        env_run(curenv);

  }

  // Destroy the environment that caused the fault.
  cprintf("[%08x] user fault va %08x ip %08x\n",
    curenv->env_id, fault_va, tf->tf_eip);
  print_trapframe(tf);
  env_destroy(curenv);
}
```

Exercise 10

_pgfault_upcall 的如下代码实现了跳回 env 发生 page fault 的地方，继续运行.

```
  movl 0x30(%esp), %ecx
  subl $4, %ecx
  movl %ecx, 0x30(%esp)
  movl 0x28(%esp), %edx
  movl %edx, (%ecx)
```

```
6
7     addl $8, %esp
8     popal
9
10    addl $4, %esp
11    popfl
12
13    pop %esp
14
15    ret
16
```

Exercise 11

`set_pgfault_handler` 设置了 fault handler.

```
1  void
2  set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
3  {
4    int r;
5
6    if (_pgfault_handler == 0) {
7      // First time through!
8      // LAB 4: Your code here.
9      int e = sys_page_alloc(thisenv->env_id, (void *)(UXSTACKTOP - PGSIZE), PTE_U|PTE_P|PTE_W);
10     if (e<0)
11       panic("set_pgfault_handler failed!!!");
12     e = sys_env_set_pgfault_upcall(thisenv->env_id, _pgfault_upcall);
13     if (e<0)
14       panic("set_pgfault_handler failed!!!");
15
16
17   }
18
19   // Save handler pointer for assembly to call.
20   _pgfault_handler = handler;
21 }
```

Exercise 12

`fork` 实现了一个 copy-on-write 的 fork.

```
envid_t
fork(void)
{
  int r;
  envid_t id = sys_getenvid();
  set_pgfault_handler(pgfault);
  envid_t t = sys_exofork();
  if (t<0)
    panic("fork: exofork error!!!");
  if (t == 0){
    thisenv = (&envs[ENVX(sys_getenvid())]);
  }
  else{
    for (uint32_t addr = 0;addr<USTACKTOP; addr+=PGSIZE)
      if ((uvpd[PDX(addr)] & PTE_P) == PTE_P && (uvpt[PGNUM(addr)] & PTE_P) == PTE_P)
        duppage(t, addr/PGSIZE);
    r = sys_page_alloc(t, (void *)(UTOP-PGSIZE), PTE_U | PTE_P | PTE_W);
    if (r<0)
      panic("fork: %e", r);
    void _pgfault_upcall();
    r = sys_env_set_pgfault_upcall(t, _pgfault_upcall);
    if (r<0)
      panic("set_pgfault_handler failed!!!");
    r = sys_env_set_status(t, ENV_RUNNABLE);
    if (r<0)
      panic("fork: %e", r);
  }
  return t;
}
```

duppage 把当前 env 的第 pn 个虚拟页 map 到另一个 env 的相同虚拟位置. 如果当前页是 copy on write 或 writable，那么将页映射都设为 copy on write.

```
static int
duppage(envid_t envid, unsigned pn)
{
  int r;
  void * va = (void *)(pn * PGSIZE);
  envid_t id = sys_getenvid();
  if ((uvpt[pn] & PTE_W) == PTE_W || (uvpt[pn] & PTE_COW) == PTE_COW){
    r = sys_page_map(id, va, envid, va, PTE_COW | PTE_U | PTE_P);
```

```
    if (r<0)
      return r;
    r = sys_page_map(id, va, id, va, PTE_COW | PTE_U | PTE_P);
    if (r<0)
      return r;
  }
  else{
    r = sys_page_map(id, va, envid, va, PTE_U | PTE_P);
    if (r<0)
      return r;
  }
  return 0;
}
```

这个是真正的 page fault handler, 处理了 copy on write 的情况.

```
static void
pgfault(struct UTrapframe *utf)
{
  void *addr = (void *) utf->utf_fault_va;
  uint32_t err = utf->utf_err;
  int r;

  if ( ((uvpt[(uint32_t)addr/PGSIZE] & PTE_COW) == PTE_COW) && ((err & FEC_WR) == FEC_WR) ){

  }
  else
    panic("pgfault: not a write to a copy-on-write page!!!");

  envid_t id = sys_getenvid();
  r = sys_page_alloc(id, PFTEMP, PTE_U | PTE_W | PTE_P);
  if (r<0)
    panic("pgfault: %e", r);
  memcpy(PFTEMP, ROUNDDOWN(addr, PGSIZE), PGSIZE);
  r = sys_page_map(id, PFTEMP, id, ROUNDDOWN(addr, PGSIZE), PTE_U | PTE_W | PTE_P);
  if (r<0)
    panic("pgfault: %e", r);
  r = sys_page_unmap(id, PFTEMP);
  if (r<0)
    panic("pgfault: %e", r);
}
```

Exercise 13

就和 lab3 的时候设立 IDT 的方法一样，现在在上面又加了一些.

```
TRAPHANDLER_NOEC(TIMER, IRQ_OFFSET + IRQ_TIMER)
TRAPHANDLER_NOEC(KBD, IRQ_OFFSET + IRQ_KBD)
TRAPHANDLER_NOEC(SERIAL, IRQ_OFFSET + IRQ_SERIAL)
TRAPHANDLER_NOEC(SPURIOUS, IRQ_OFFSET + IRQ_SPURIOUS)
TRAPHANDLER_NOEC(IDE, IRQ_OFFSET + IRQ_IDE)
TRAPHANDLER_NOEC(ERROR, IRQ_OFFSET + IRQ_ERROR)
```

```
  void TIMER();
  SETGATE(idt[IRQ_OFFSET + IRQ_TIMER], 0, GD_KT, TIMER, 0);
  void KBD();
  SETGATE(idt[IRQ_OFFSET + IRQ_KBD], 0, GD_KT, KBD, 0);
  void SERIAL();
  SETGATE(idt[IRQ_OFFSET + IRQ_SERIAL], 0, GD_KT, SERIAL, 0);
  void SPURIOUS();
  SETGATE(idt[IRQ_OFFSET + IRQ_SPURIOUS], 0, GD_KT, SPURIOUS, 0);
  void IDE();
  SETGATE(idt[IRQ_OFFSET + IRQ_IDE], 0, GD_KT, IDE, 0);
  void ERROR();
  SETGATE(idt[IRQ_OFFSET + IRQ_ERROR], 0, GD_KT, ERROR, 0);
```

Exercise 14

在 `trap_dispatch` 里增加了对 timer interrupt 的处理.

```
  if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
    lapic_eoi();
    sched_yield();
    return;
  }
```

Exercise 15

在 `sys_ipc_recv` 这个 syscall 将 env 设为在 recv 的状态，并将 `env_status` 设为 `ENV_NOT_RUNNABLE` 使得在收到东西前不去跑这个 env.

```
static int
sys_ipc_recv(void *dstva)
{
```

```
  if ((uint32_t)dstva < UTOP && (uint32_t)dstva%PGSIZE!=0 )
    return -E_INVAL;
  curenv->env_ipc_recving = 1;
  curenv->env_ipc_dstva = dstva;
  curenv->env_status = ENV_NOT_RUNNABLE;
  return 0;
}
```

在 sys_ipc_try_send 这个 syscall 尝试向一个 env 发送东西.

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
  // LAB 4: Your code here.
  int r;
  struct Env * env;
  struct PageInfo * pp;
  pte_t * pte;
  if ((r=envid2env(envid,&env,0))<0)
    return r;

  if (env->env_ipc_recving != 1)
    return -E_IPC_NOT_RECV;
  if ((uint32_t)srcva < UTOP && (uint32_t)srcva%PGSIZE!=0)
    return -E_INVAL;

  if ((uint32_t)srcva < UTOP && ((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P) || (((perm ^ (PTE
    return -E_INVAL;
  if ((uint32_t)srcva < UTOP && (pp = page_lookup(curenv->env_pgdir, srcva, &pte)) == NULL)
        return -E_INVAL;

    if ((uint32_t)srcva < UTOP && ((*pte)&PTE_W) == 0 && (perm & PTE_W)!=0)
    return -E_INVAL;

  if ((uint32_t)(env->env_ipc_dstva) < UTOP && (uint32_t)srcva < UTOP){
    if ((r = page_insert(env->env_pgdir, pp, env->env_ipc_dstva, perm))<0)
      return r;
    env->env_ipc_perm = perm;
  }
  else
    env->env_ipc_perm = 0;
```

```
32
33    env->env_ipc_recving = 0;
34    env->env_ipc_from = curenv->env_id;
35    env->env_ipc_value = value;
36    env->env_status = ENV_RUNNABLE;
37    return 0;
38  }
```

在 ipc_recv 用来收东西，基本直接调用 sys_ipc_recv.

```
1  int32_t
2  ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
3  {
4    if (pg == NULL)
5      pg = (void *)UTOP;
6    int r;
7    if ((r = sys_ipc_recv(pg))<0){
8      if (from_env_store) (*from_env_store) = 0;
9      if (perm_store) (*perm_store)=0;
10     return r;
11   }
12   if (from_env_store) (*from_env_store) = thisenv->env_ipc_from;
13   if (perm_store) (*from_env_store) = thisenv->env_ipc_perm;
14   return thisenv->env_ipc_value;
15 }
```

在 ipc_send 用来发东西,基本直接调用 sys_ipc_send,在对面不处在收东西的状态的时候就先 sys_yield 调度别的进程，减少了对 CPU 资源的浪费.

```
1  void
2  ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
3  {
4    int r;
5    if (pg == NULL)
6      pg = (void *)UTOP;
7    while (1){
8      r=sys_ipc_try_send(to_env, val, pg, perm);
9      if (r<0 && r!=-E_IPC_NOT_RECV)
10       panic("ipc_send: %e", r);
11     if (r==0) break;
12     sys_yield();
13   }
14 }
```