Lab4：进程环境

Lab 4

- 多处理器和多任务
- 抢占与锁
- COW、Fork和Fault

多核系统内核初始化流程?

- 在启动 APs 之前，BSP 先搜集多处理器系统的信息，在mp_init()中实现。
- 将 AP 的入口代码 (kern/mpentry.S) 拷贝到实模式可以寻址的内存区域。
- boot_aps() 调用lapic_startap()，发送 STARTUP 到 LAPIC 单元，激活 AP。
- AP 调用 mpentry_start ()。 *mpentry_start做了什么？*
- boot_aps() 等待 AP 初始化结束并发送 CPU_STARTED 信号后，再激活下一个 AP

1. 加载GDT；2. 切换到保护模式； 3. 开启分页

Mpentry.S中为什么要使用MPBOOTPHYS转换的值?

- 此时只能读取低地址，所以需要把高地址转换成低地址

```
# This code is similar to boot/boot.S except that
#    - it does not need to enable A20
#    - it uses MPBOOTPHYS to calculate absolute addresses of its
#      symbols, rather than relying on the linker to fill them
```

```
#define RELOC(x) ((x) - KERNBASE)
#define MPBOOTPHYS(s) ((s) - mpentry_start + MPENTRY_PADDR)
```

```
.code16
.globl mpentry_start
mpentry_start:
    cli

    xorw    %ax, %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss

    lgdt    MPBOOTPHYS(gdtdesc)
    movl    %cr0, %eax
    orl     $CR0_PE, %eax
    movl    %eax, %cr0

    ljmpl   $(PROT_MODE_CSEG), $(MPBOOTPHYS(start32))
```

mpentry.S的链接地址和加载地址?

- 链接在高地址，因为mpentry.S是内核文件的一部分
- 加载在低地址，因为AP还在实模式，只能访问低地址
- lapic_startap(c->cpu_id, PADDR(code));
  - code的物理地址，这里转换成低地址
  - 在mpentry.S中用MPBOOTPHYS把start32的链接地址转换成物理地址

Kern/mpentry.S中76行的问题，为什么使用间接跳转

- 从低地址跳到高地址?

\- 此时处在一个低地址，使用直接跳转 dir = offset + dir_now
目前汇编的地址（dir_now）很小，由此计算的dir 出错
直接跳转的话，就是直接指定IP，不会出错
\- boot loader 都在 低地址，所以可以直接跳转

```
  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call bootmain
```

init.s 里没有使用间接跳转

```
53  .code32
54  start32:
55      movw    $(PROT_MODE_DSEG), %ax
56      movw    %ax, %ds
57      movw    %ax, %es
58      movw    %ax, %ss
59      movw    $0, %ax
60      movw    %ax, %fs
61      movw    %ax, %gs
62
63      # Set up initial page table. We cannot use kern_pgdir yet because
64      # we are still running at a low EIP.
65      movl    $(RELOC(entry_pgdir)), %eax
66      movl    %eax, %cr3
67      # Turn on paging.
68      movl    %cr0, %eax
69      orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
70      movl    %eax, %cr0
71
72      # Switch to the per-cpu stack allocated in boot_aps()
73      movl    mpentry_kstack, %esp
74      movl    $0x0, %ebp       # nuke frame pointer
75
76      # Call mp_main().  (Exercise for the reader: why the indirect call?)
77      movl    $mp_main, %eax
78      call    *%eax
```

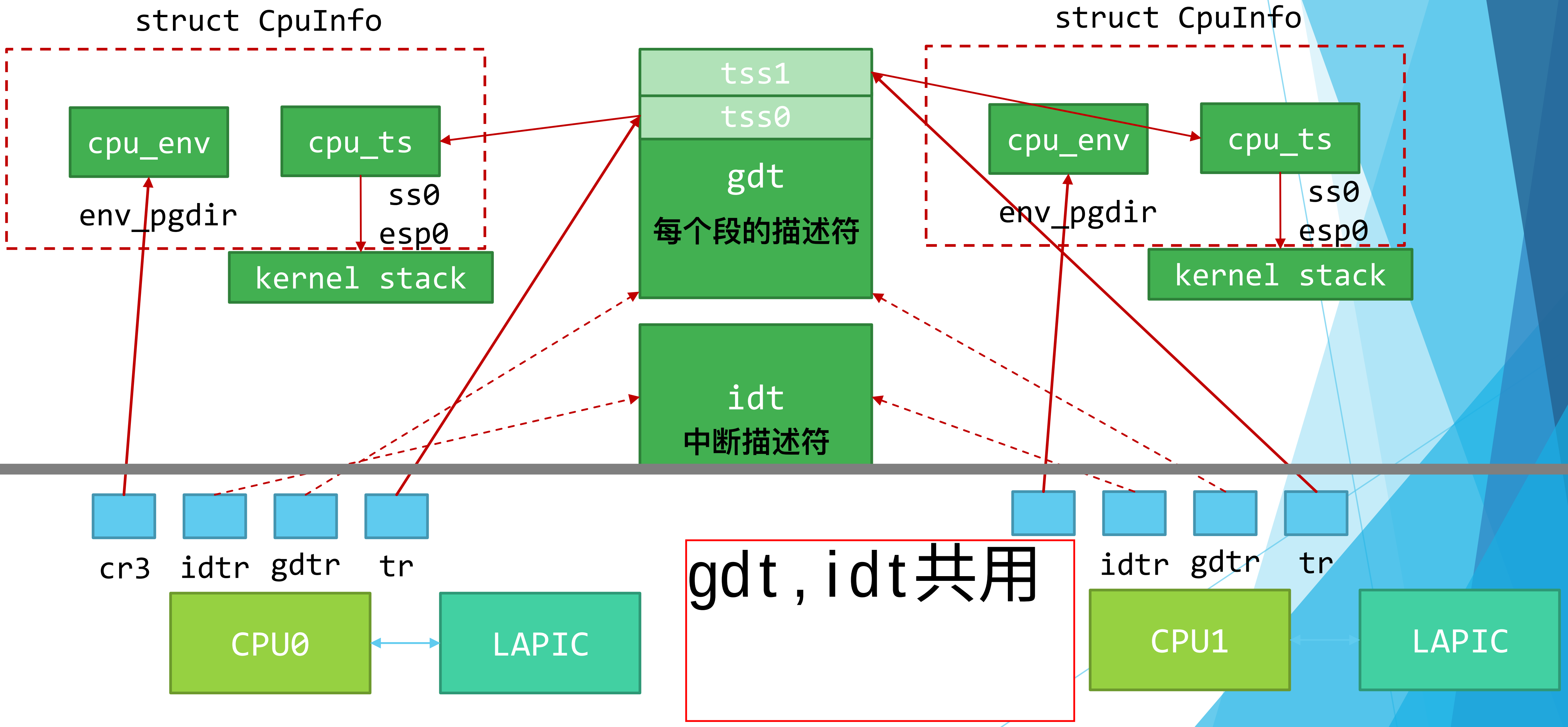
mpentry.s 里使用了间接跳转

AP、BSP启动过程有何异同?

- BSP会进行必要的初始化，mem_init，trap_init，建立物理内存管理、虚拟内存管理、建立进程的执行环境，多核执行环境
- AP不需要做这些东西，只需要把自己的东西加载进来，比如加载自己的GDT，加载自己的TSS和IDT，但是不用进行配置

Ap的TSS IDT是BSP维护的

哪些数据结构是每个CPU独有的?



多核为何不能共享内核栈？

- 即使使用大内核锁确保同时只能有1个CPU进入内核，也会造成问题
  - 考虑以下场景：CPU0正在处理用户态的中断，struct Trapframe被压入内核栈进入trap()函数后才获取大内核锁lock_kernel()
  - 在获得大内核锁之前，别的CPU可能会修改内核栈，导致栈帧被破坏。

Lock_kernel和unlock_kernel的工作原理

xchg是指令(原子化)

- 使用xchg实现
- 如何用spinlock实现mutex。
- 如何用spinlock实现信号量。
  - mutex是信号量值为1时的特殊情况

```
struct mutex {
    atomic_long_t       owner;
    spinlock_t      wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
    struct list_head    wait_list;
#ifdef CONFIG_DEBUG_MUTEXES
    void            *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map  dep_map;
#endif
};
```

进入内核（加锁）和离开内核（解锁）的时机有哪些

- 加锁
  - I386_init()在BSP启动其他CPU之前对内核加锁
  - 在mp_main()初始化AP后，加锁，然后调用sched_yield()运行当前AP上第一个进程
  - 在trap中，当程序从用户态进入内核态时加锁
    - 如果是从内核态再次进入内核态（即内核发生trap）时不需要加锁
- 解锁
  - 在切换回用户态之前释放锁 (env_pop_tf)

JOS的进程调度算法

- Round-robin调度
- 寻找下一个可运行的env
  - env是RUNNING，说明别的CPU在运行
  - env是RUNNABLE，就调度到它

```c
// Choose a user environment to run and run it.
void sched_yield(void) {
    struct Env *idle;

    // Implement simple round-robin scheduling.
    //
    // Search through 'envs' for an ENV_RUNNABLE environment in
    // circular fashion starting just after the env this CPU was
    // last running.  Switch to the first such environment found.
    //
    // If no envs are runnable, but the environment previously
    // running on this CPU is still ENV_RUNNING, it's okay to
    // choose that environment. Make sure curenv is not null before
    // dereferencing it.
    //
    // Never choose an environment that's currently running on
    // another CPU (env_status == ENV_RUNNING). If there are
    // no runnable environments, simply drop through to the code
    // below to halt the cpu.

    // LAB 4: Your code here.
    int end = 0, i;
    if (curenv) {
        if (curenv->env_status == ENV_RUNNING)
            curenv->env_status = ENV_RUNNABLE;
        end = curenv - envs;
    }
    i = end;
    do {
        i = (i + 1) % NENV;
        if (envs[i].env_status == ENV_RUNNABLE)
            env_run(&envs[i]); // never returns
    } while (i != end);

    // sched_halt never returns
    sched_halt();
    panic("sched_yield: attempted to return");
}
```

触发进程调度的时机有哪些

```c
// Deschedule current environment and pick a different one to run
static void
sys_yield(void)
{
    sched_yield();
}
```

当前进程执行系统调用sys_yield时

```c
    if ((tf->tf_cs & 3) == 3) {
        // Trapped from user mode.
        // Acquire the big kernel lock before doing any
        // serious kernel work.
        // LAB 4: Your code here.
        lock_kernel();
        assert(curenv);

        // Garbage collect if current enviroment is a zombie
        if (curenv->env_status == ENV_DYING) {
            env_free(curenv);
            curenv = NULL;
            sched_yield();
        }
```

进入陷阱时已死亡的用户进程

```c
        // Copy trap frame (which is currently on the stack)
        // into 'curenv->env_tf', so that running the environment
        // will restart at the trap point.
        curenv->env_tf = *tf;
        // The trapframe on the stack should be ignored from here on
        tf = &curenv->env_tf;
    }

    // Record that tf is the last real trapframe so
    // print_trapframe can print some additional information.
    last_tf = tf;

    // Dispatch based on what type of trap occurred
    trap_dispatch(tf);

    // If we made it to this point, then no other environment was
    // scheduled, so we should return to the current environment
    // if doing so makes sense.
    if (curenv && curenv->env_status == ENV_RUNNING)
        env_run(curenv);
    else
        sched_yield();
}
```

进入trap_dispatch 返回时

```c
void
env_destroy(struct Env *e)
{
    // If e is currently running on other CPUs, we change its state to
    // ENV_DYING. A zombie environment will be freed the next time
    // it traps to the kernel.
    if (e->env_status == ENV_RUNNING && curenv != e) {
        e->env_status = ENV_DYING;
        return;
    }

    env_free(e);

    if (curenv == e) {
        curenv = NULL;
        sched_yield();
    }
}
```

进程死亡时

```c
// Handle clock interrupts. Don't forget to acknowledge the
// interrupt using lapic_eoi() before calling the scheduler!
// LAB 4: Your code here.
if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
    cprintf("Clock interrupts on irq 0\n");
    lapic_eoi();
    sched_yield();
    return;
}
```

时钟中断时

```c
// Setup code for APs
void
mp_main(void)
{
    // We are in high EIP now, safe to switch to kern_pgdir
    lcr3(PADDR(kern_pgdir));
    cprintf("SMP: CPU %d starting\n", cpunum());

    lapic_init();
    env_init_percpu();
    trap_init_percpu();
    xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up

    // Now that we have finished some basic setup, call sched_yield()
    // to start running processes on this CPU.  But make sure that
    // only one CPU can enter the scheduler at a time!
    //
    // Your code here:
    lock_kernel();
    sched_yield();
}
```

每个AP启动完毕时

```c
    // Starting non-boot CPUs
    boot_aps();

    // Start fs.
    ENV_CREATE(fs_fs, ENV_TYPE_FS);

#if defined(TEST)
    // Don't touch -- used by grading script!
    ENV_CREATE(TEST, ENV_TYPE_USER);
#else
    // Touch all you want.
    ENV_CREATE(user_icode, ENV_TYPE_USER);
#endif // TEST*

    // Should not be necessary - drains keyboard because interrupt has given up.
    kbd_intr();

    // Schedule and run the first user environment!
    sched_yield();
}
```

BSP运行结束时

解读user/dumbfork.c

- 创建新进程必须要做哪些事情?
  - 创建一个env
  - 新分配一个pgdir
- dumbfork与fork的相同点和不同点?
- dumbfork和fork都调用了sys_exofork，该函数为什么必须强制内联?

dumbfork:直接复制
fork:父子进程是COW共享的

不内联的话，会创建一个临时栈帧，然后子进程会指向这个栈帧，但是创建完就返回了（这个临时栈帧会被释放），这个栈帧实际上不会用到，就会出现一些bug。

```c
// This must be inlined.  Exercise for reader: why?
static inline envid_t __attribute__((always_inline))
sys_exofork(void)
{
	envid_t ret;
	asm volatile("int %2"
		     : "=a" (ret)
		     : "a" (SYS_exofork), "i" (T_SYSCALL));
	return ret;
}
```

第一次触发COW是什么时候

- 当对USTACK添加COW后，fork第一次调用duppage时会发生COW。
  - fork调用duppage会将参数压栈，对USTACK进行了修改，产生page fault
- 硬件（MMU）知道COW的存在吗

不知道
是操作系统判断的
对硬件来说是因为写违反

为什么要先把COW页映射到子进程空间

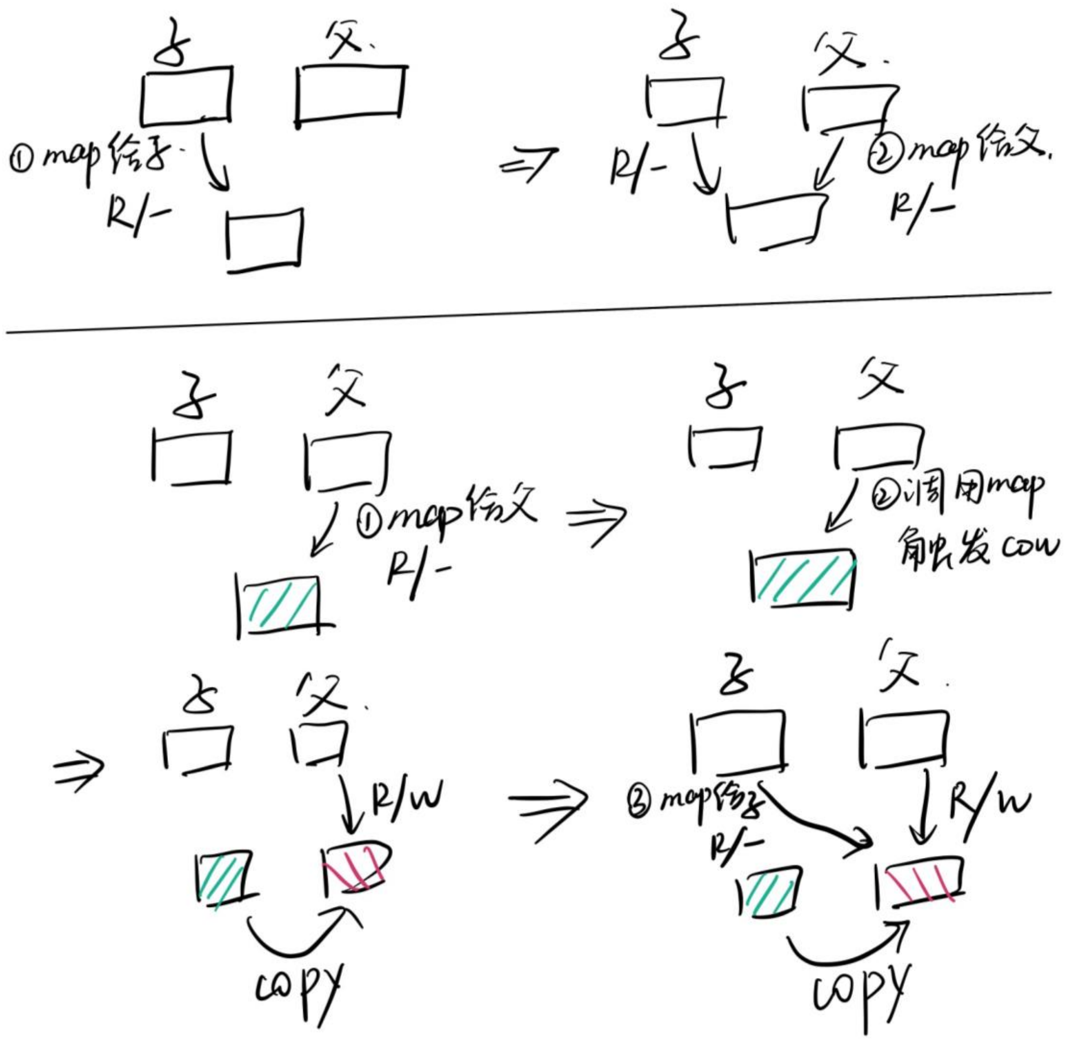
- 为什么要先把COW页映射到子进程空间，然后再映射到父进程空间？这个顺序为什么不能反？
- 考虑父进程的用户运行栈。如果首先把父进程的运行栈的页映射到父进程空间，那么当映射函数返回时（或调用给子进程映射的映射函数）会修改这个页，父进程会复制一个新页，权限是可写的。
- 之后，当父进程把用户运行栈的页映射到子进程空间时，因为这个页是新分配的页，对于父进程不是COW，而对于子进程是COW的。这就导致父进程可以随意修改子进程COW的页

```c
static int
duppage(envid_t envid, unsigned pn)
{
    int r;

    // LAB 4: Your code here.
    // panic("duppage not implemented");
    int perm = PTE_U | PTE_P;
    void* addr = (void*)(pn * PGSIZE);
    if(uvpt[pn] & (PTE_W | PTE_COW))
        perm |= PTE_COW;

    if((r = sys_page_map(0, addr, envid, addr, perm)) < 0)
        return r;
    if((r = sys_page_map(0, addr, 0, addr, perm)) < 0)
        return r;

    return 0;
}
```

先映射子进程

后映射父进程

为什么要先把COW页映射到子进程空间

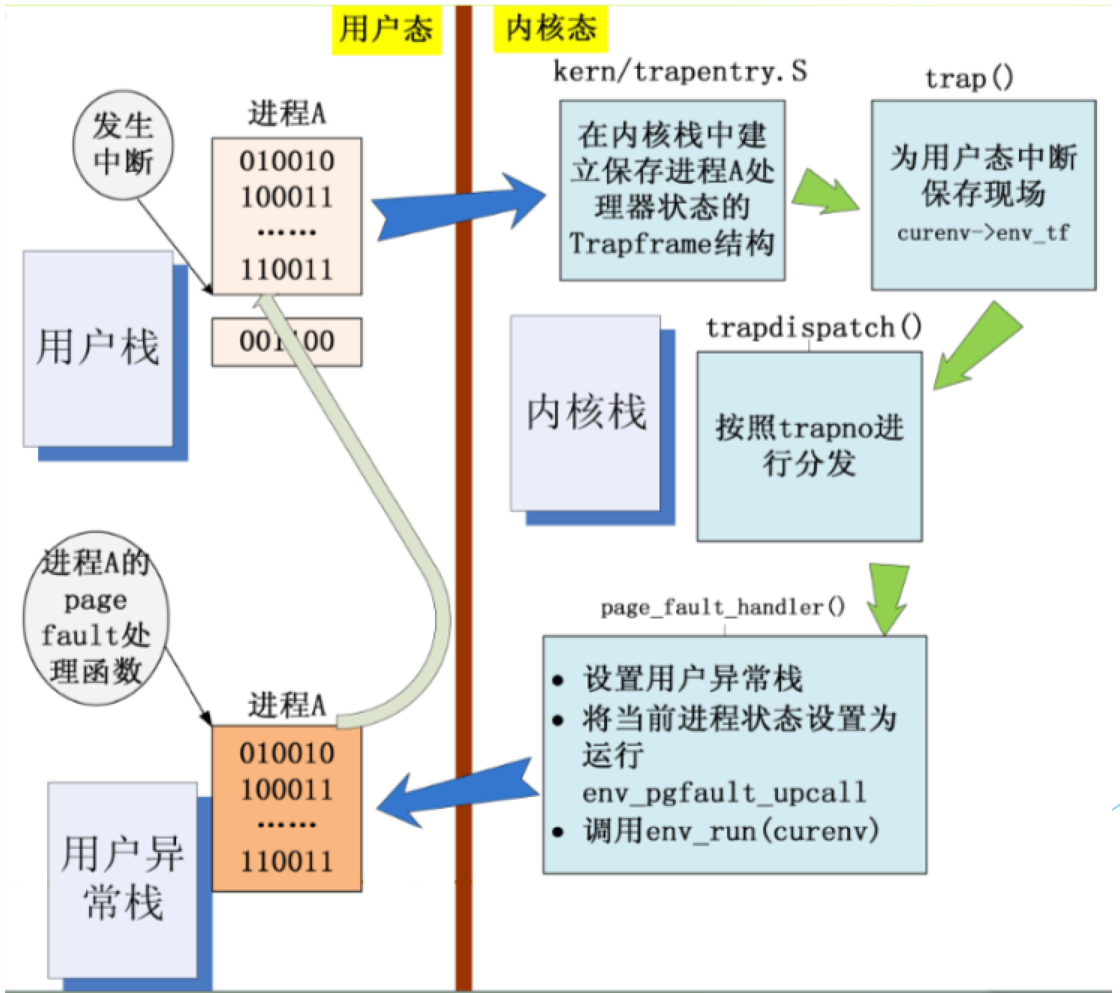


能否对UXSTACK、页表和页目录使用COW?

- 因为page fault处理程序需要使用UXSTACK，如果对UXSTACK使用COW会导致死循环
- 页目录+页表都加COW
  - 写某个页面触发COW ->
  - 进入page fault handler修改页表项 ->
  - 触发COW进入嵌套page fault handler，修改页目录项->
  - 触发COW…死循环
- 只给页表加COW
  - 写某个页面触发COW ->
  - 进入page fault handler修改页表项 ->
  - 触发COW进入嵌套page fault，修改页目录项，再次触发page fault

UXSTACK是在用户态处理fault用的，与架构相关
如果没有这个栈，直接在USTACK处理，调用的时候就会混起来.
如果有中断嵌套的话可能栈会出现破坏。
- UTrapframe 和TrapFrame
— 会多记录出错的地址falut_va，在内核中可以直接从 cr2 读出
— 但是没有段寄存器信息比如cs ds es（用户态不需要切换栈）

总结用户态处理Page fault的流程



其他一些问题

- JOS在用户态处理page fault有哪些优势?
- Lab4实现用户态处理page fault与Lab3实现的内核态处理page fault有什么不同?
- 进程间通信有哪些方式? JOS支持的IPC属于哪种方式
- 如何实现非忙等的ipc_send

Thanks