
LunaixOS
从零开始
DEVELOPING YOUR OWN
自制操作系统
OPERATING SYSTEM FROM SCRATCH
多进程
进程，中断，与虚拟内存
EP 10-1

第 10 集规划

由于内容较多，细节较为复杂，我将拆分为以下几个话题，
分数个视频，进行讲解

1. 进程，中断与虚拟内存（还有调度！）
2. 进程中的内存
3. To fork or not to fork?
4. 进入用户空间
5. 为系统调用做好准备
6. 一些 Unix 系统调用的具体实现

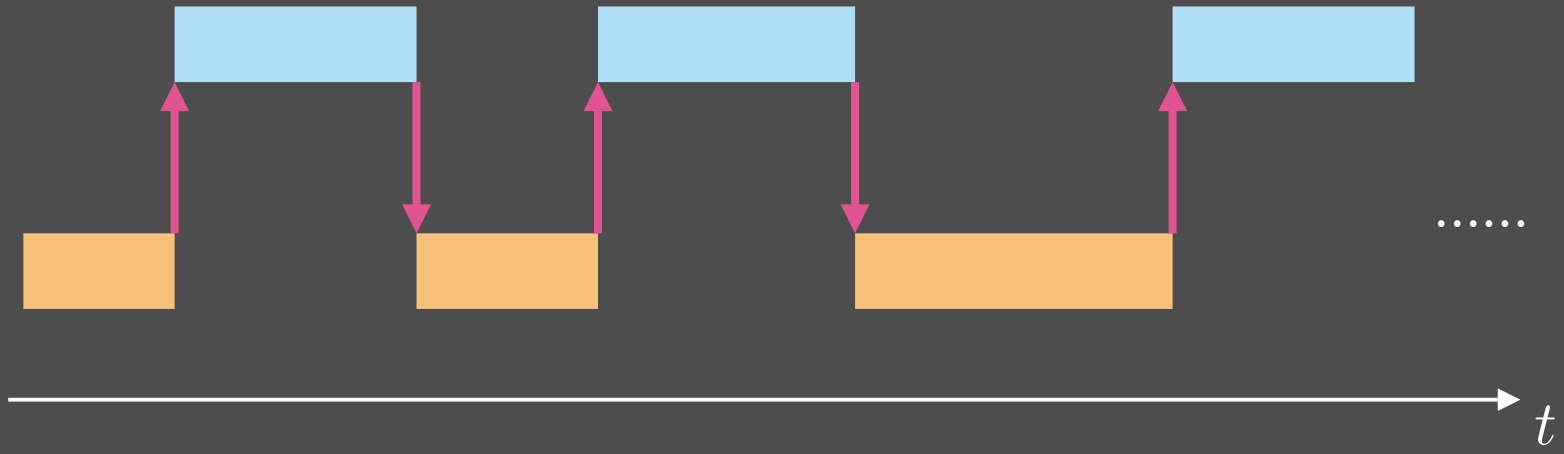
《小马宝莉》其实是一个计算机科学启蒙片……

如何一心多用?

好玩的事情有两个，可是 Pinkie Pie 只有一个……

同云宝黛茜
戏水休闲

帮苹果嘉儿
建设谷仓



这正是进程的核心思想!

如何实现这玩意？

我们需要……

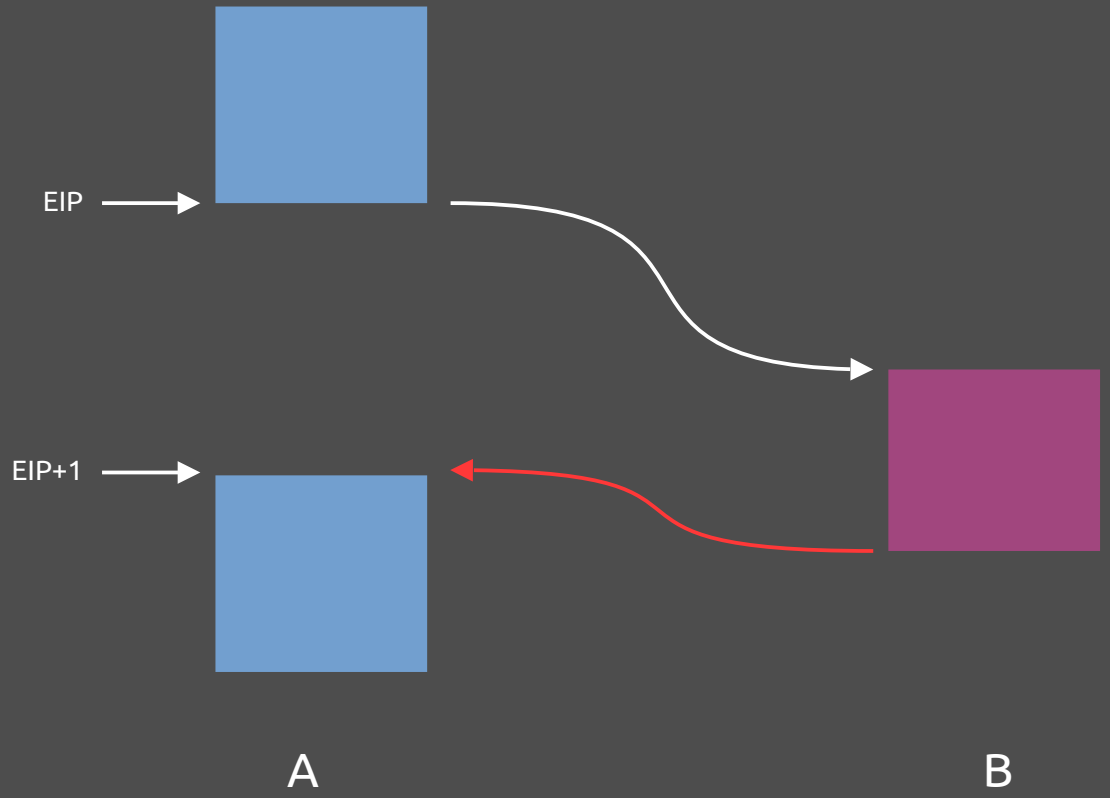
1. 知道一个进程在哪里被切走，这样我们才能在稍后恢复执行。（保证**连续性**）

2. 知道进程被切走前一瞬间内的所有状态，并且能够正确的恢复这些状态！（保证**正确性**）

3. 知道什么时候应该进行切换，并且确保每个进程都会被执行。（保证**公平性**）

连续性的保证

简单！我们只需要记录程序指针就好了。 CS:IP



正确性?

一个程序的正确运行取决于什么呢?

1. 所有通用寄存器的内容
2. 栈的内容
3. 应用程序分配出来的所有内存空间
4. 程序本身

麻烦！太多状态了！好像根本不可能！

事实果真如此?

正确性 & 连续性

让我们从全局来仔细想想……

0. CS 和 IP
1. 所有通用寄存器的内容

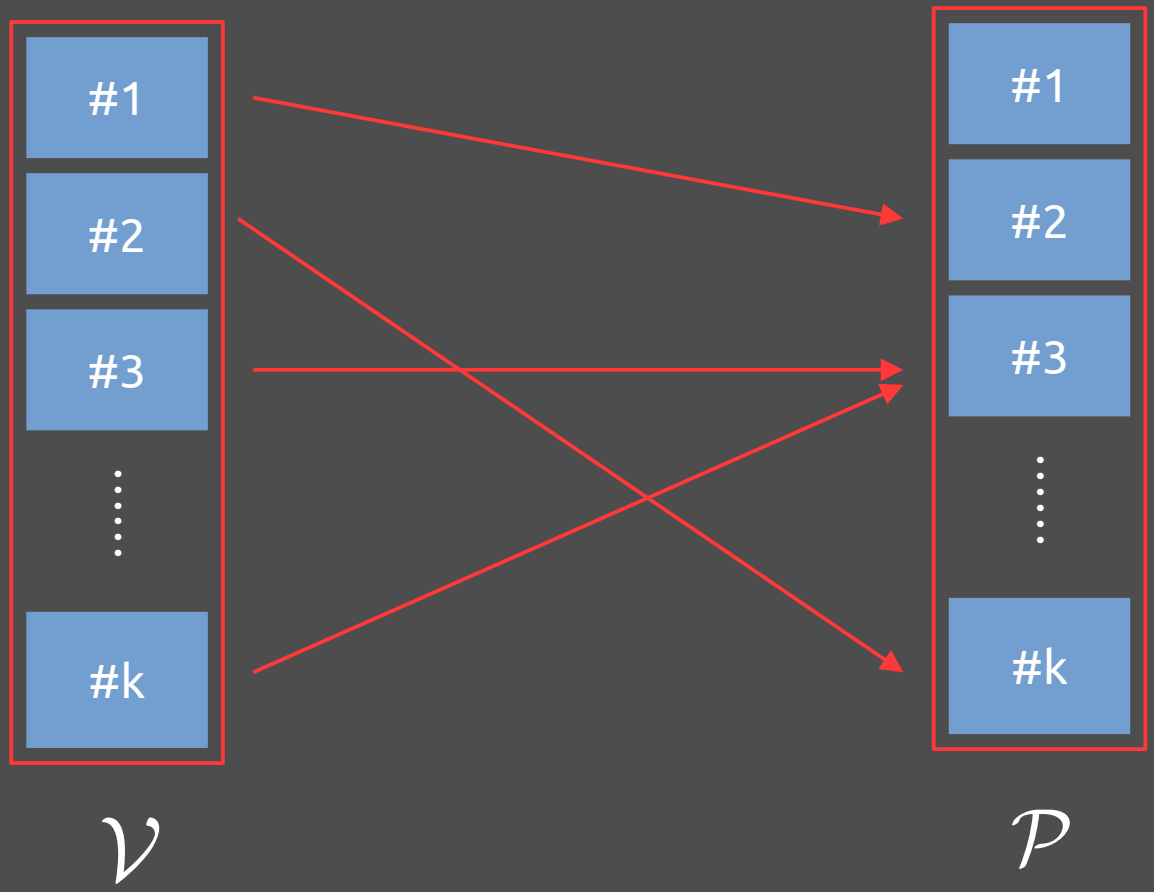
} 中断的上下文环境！

2. 栈的内容
3. 应用程序分配出来的所有内存空间
4. 程序本身

} 虚拟内存！

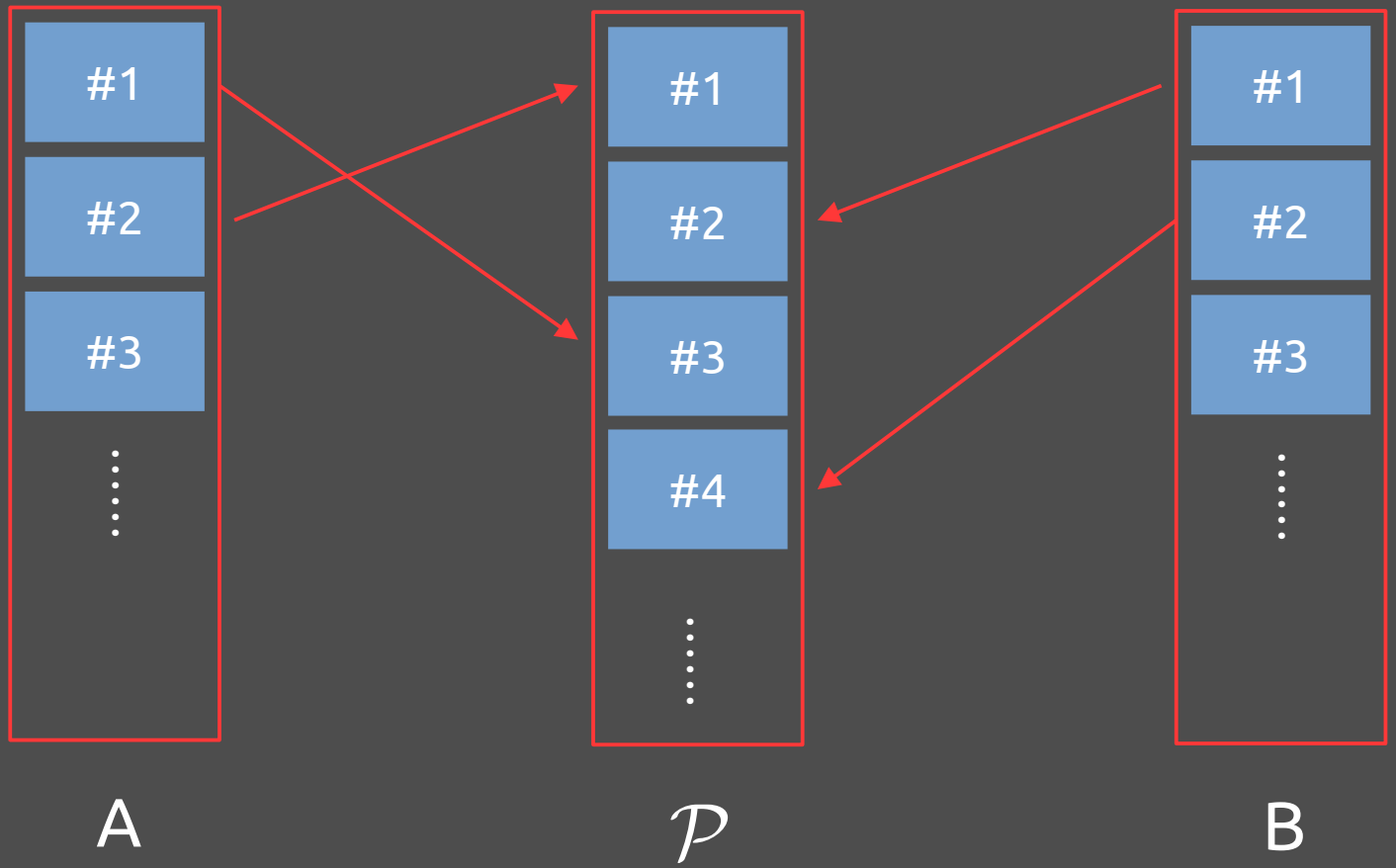
进程 x 虚拟内存

回忆一下虚拟内存……



进程 x 虚拟内存

虚拟内存已经帮我们实现了进程内存隔离

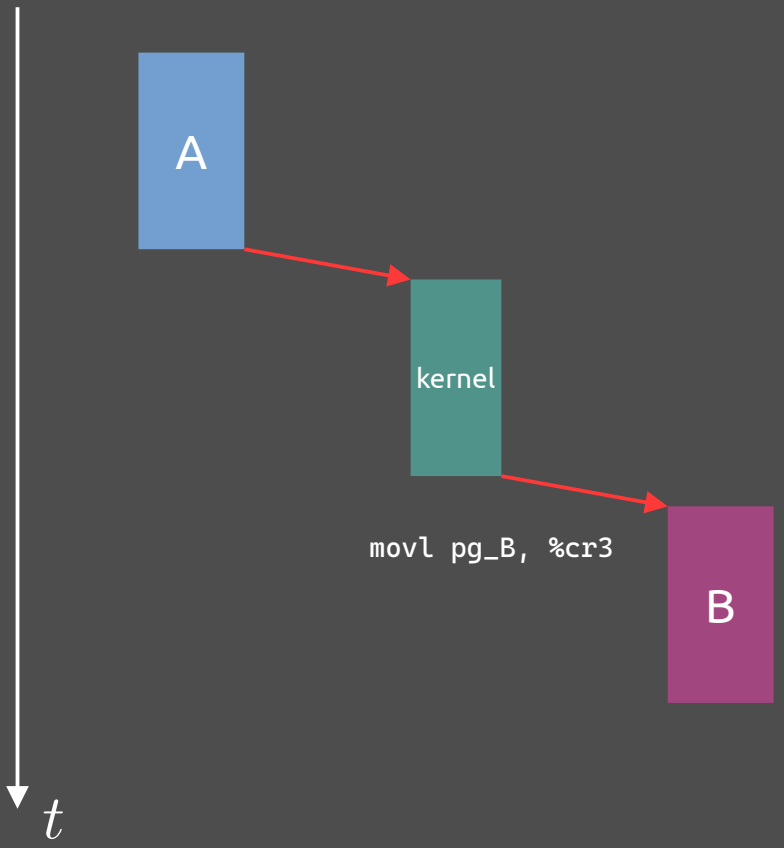


进程 x 虚拟内存

每个进程都有自己的一套页表

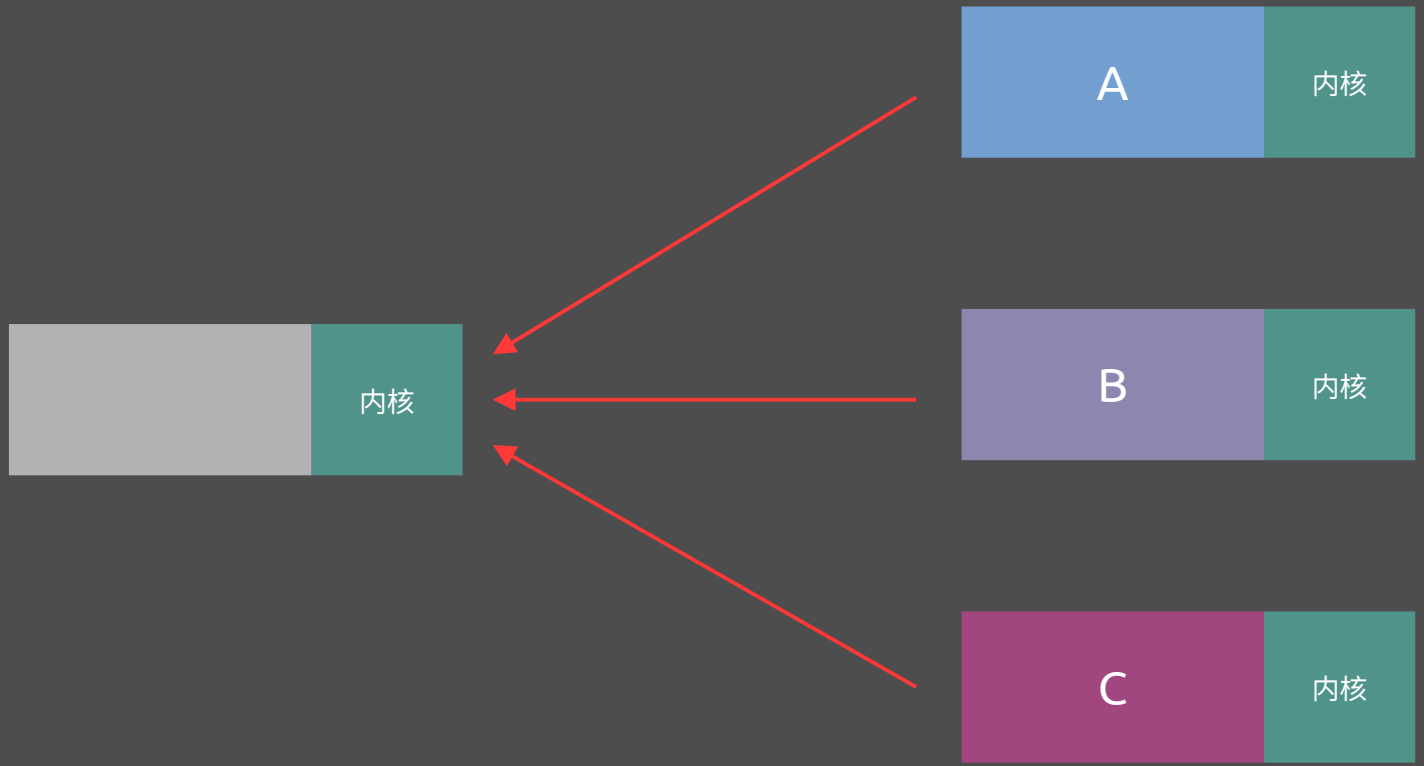
切换页表 $\Longrightarrow$ 切换“内存”

切换“内存” $\Longrightarrow$ 切换进程的所有运行时状态！



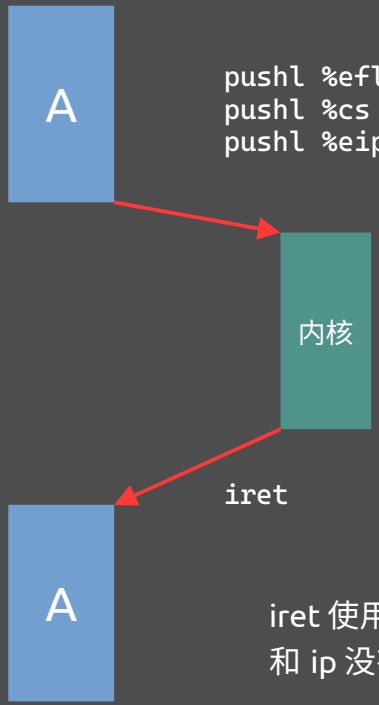
那么，内核怎么办？

在 LunaixOS 中，所有进程的页表都继承自内核页表

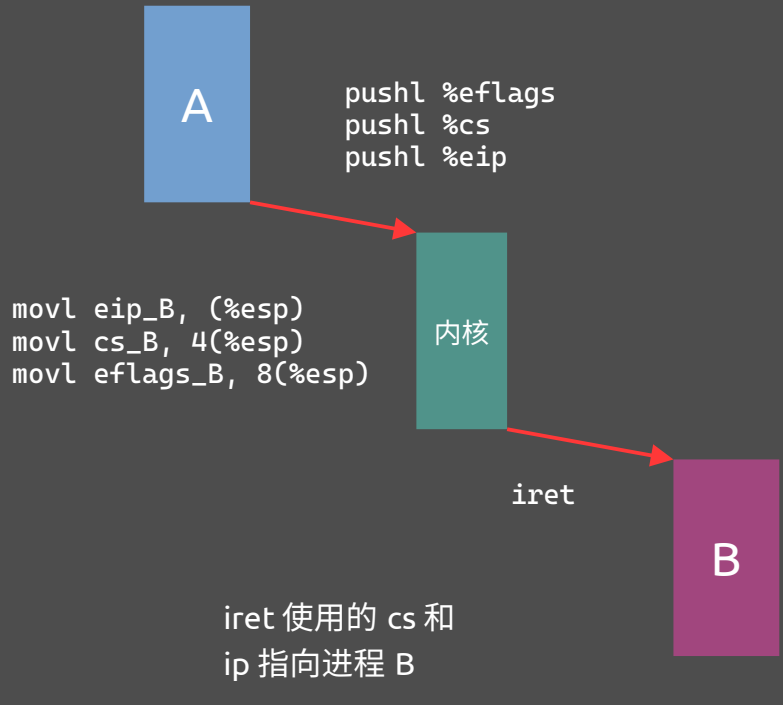


进程 x 中断

我们可以用中断的触发为信号，进行
上下文切换（ Context Switch ）



一般的中断处理流程



用于上下文切换的中断处理流程

进程 x 中断

其实我们很早就实现了寄存器状态的保存和恢复

```
38      interrupt_wrapper:
39          pushl %esp
40
41          pushl %esi
42          pushl %ebp
43          pushl %edi
44          pushl %edx
45          pushl %ecx
46          pushl %ebx
47          pushl %eax
48
49          movl %esp, %eax
50          andl $0xfffffff0, %esp
51          subl $16, %esp
52          movl %eax, (%esp)
53
54          call intr_handler
55
```

```
56      .global soft_iret
57      soft_iret:
58          popl %esp
59
60          popl %eax
61          popl %ebx
62          popl %ecx
63          popl %edx
64          popl %edi
65          popl %ebp
66          popl %esi
67          popl %esp
68
69          addl $8, %esp
70
71  › #ifdef __ASM_INTR_DIAGNOSIS …
75  › 1: …
80    #else
81          iret
82    #endif
```

You, 3 months ago | 1 author (You)

```
8   typedef struct {
9       gp_regs registers;
10      unsigned int vector;
11      unsigned int err_code;
12      unsigned int eip;
13      unsigned int cs;
14      unsigned int eflags;
15      unsigned int esp;
16      unsigned int ss;
17  } __attribute__((packed)) isr_param;
```

保证公平性

中断的使用让我们可以回答第三个问题的前半部分。

使用 APIC 计时器产生的中断来作为上下文切换的信号

但如何决定该切换到哪个进程呢? ⟶ 调度器!

那么如何识别一个进程? ⟶ pid

公平但不一定公正的调度器

一般而言，一个调度器是：

公平的：确保每个进程都有执行的机会。

不公正的：会给予某些进程特殊待遇，比如更长的执行时间。

在 LunaixOS 中，调度器是公平且公正的

轮询式调度器（Round-Robin Scheduler）

轮询式调度器与进程状态

没有优先级的概念，逐个执行注册在进程表中的每一个进程。

每个进程都有一样长的执行时间。

进程的是否执行取决于他们的**状态**。

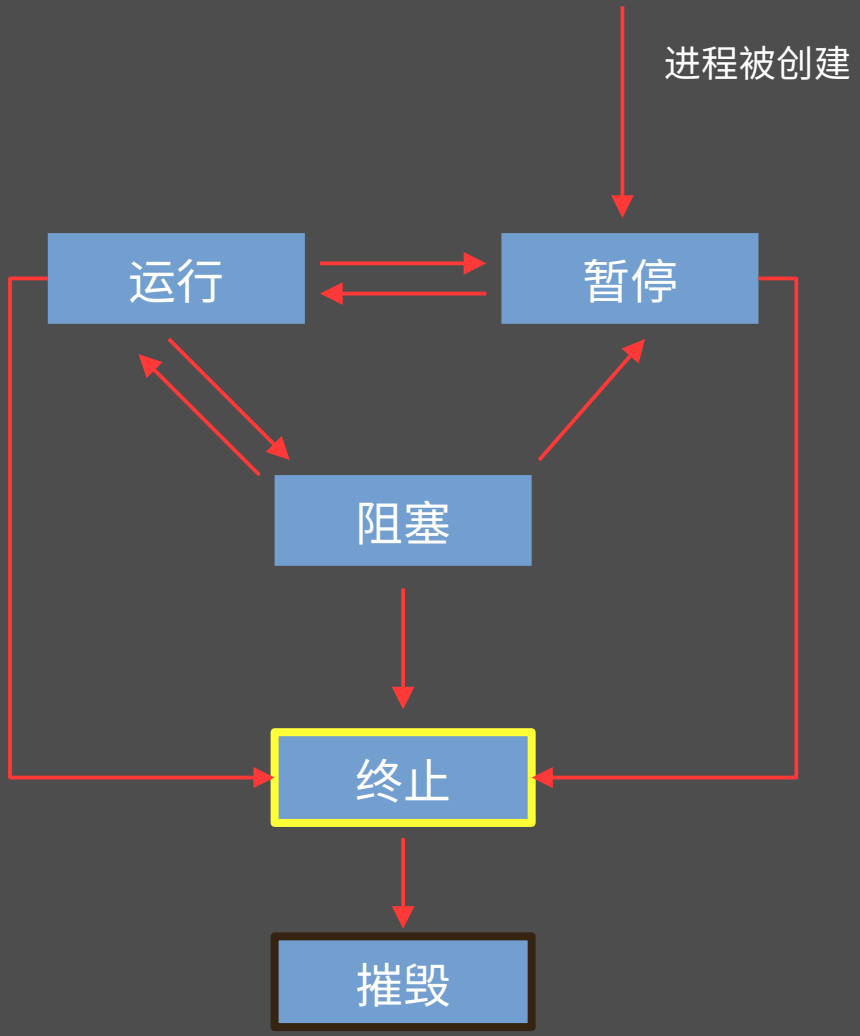
运行：正在使用 CPU

暂停：可以随时被调度

阻塞：只有当特定条件满足时，才会要么加载进 CPU，要么进入“暂停”状态

终止（僵尸）：已终止，但仍存留在进程表里，等待用户读取返回代码，不能被调度，并且 pid 不能被回收。

摧毁：进程已被释放，其 pid 可以被回收重用。



LunaixOS 的 RR 调度器实现

```
55  void schedule() {
56      if (!sched_ctx.ptable_len) {
57          return;
58      }
59
60      struct proc_info* next;
61      int prev_ptr = sched_ctx.procs_index;
62      int ptr = prev_ptr;
63      // round-robin scheduler
64      do {
65          ptr = (ptr + 1) % sched_ctx.ptable_len;
66          next = &sched_ctx._procs[ptr];
67      } while(next->state != PROC_STOPPED && ptr != prev_ptr);
68
69      sched_ctx.procs_index = ptr;
70
71      run(next);
72  }
```

```
37
38  void run(struct proc_info* proc) {
39      if (!(__current->state & ~PROC_RUNNING)) {
40          __current->state = PROC_STOPPED;
41      }
42      proc->state = PROC_RUNNING;
43
44      __current = proc;
45
46      cpu_lcr3(__current->page_table);
47
48      apic_done_servicing();
49
50      asm volatile (
51          "pushl %0\n"
52          "jmp soft_iret\n"::"r"(&__current->intr_ctx): "memory");
53  }
54
```

切换页目录

利用 iret 实现上下文切换

表示一个进程

我们可以看到，一个进程至少有这几样元素构成：

1. 中断上下文（isr_param）
2. 一个页表
3. 一个进程的标识符（pid）
4. 进程的状态

实际在 LunaixOS 里面……

```
You, 13 hours ago | 1 author (You)
27  struct proc_info {
28      pid_t pid;
29      struct proc_info* parent;
30      isr_param intr_ctx;
31      [redacted]
32      void* page_table;
33      time_t created;
34      uint8_t state;
35      int32_t exit_code;
36      [redacted]
37      [redacted]
38  };
```

父进程

中断上下文

页目录基地址

创建时间

状态

退出码

进程控制块儿

（ Process Control Block, **PCB** ）

What's Next?

进程的实现看上去非常简单，但事实不是如此……

许多问题没有得到解答：

分配内存需要内核的帮助，可是内核如何知道进程的内存结构？

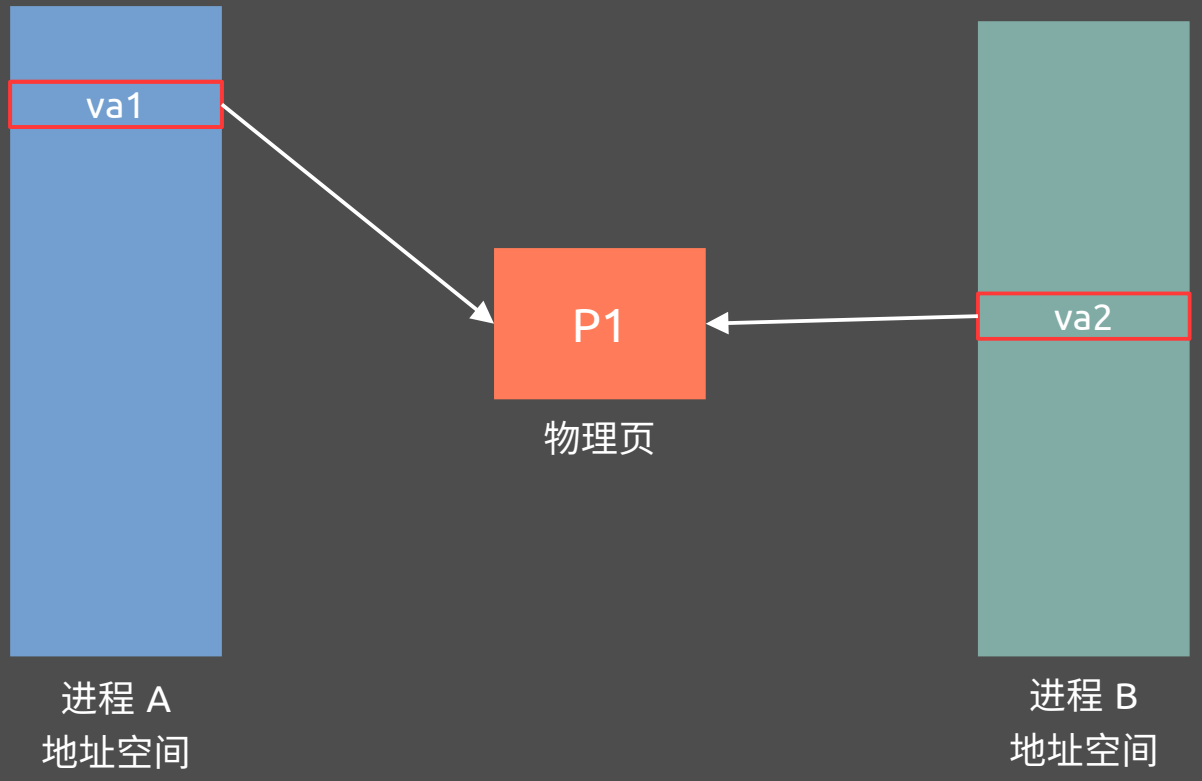
并发内存访问：物理内存只有一个，如何真正保证进程之间的互不干扰？

内核共享：每个进程都需要内核帮助来管理计算机，如何共享内核的**运行时**？

安全问题：进程 / 内核内存之间的访问控制可以通过设置虚拟页的优先级来实现；如何实现进程内的访问控制？

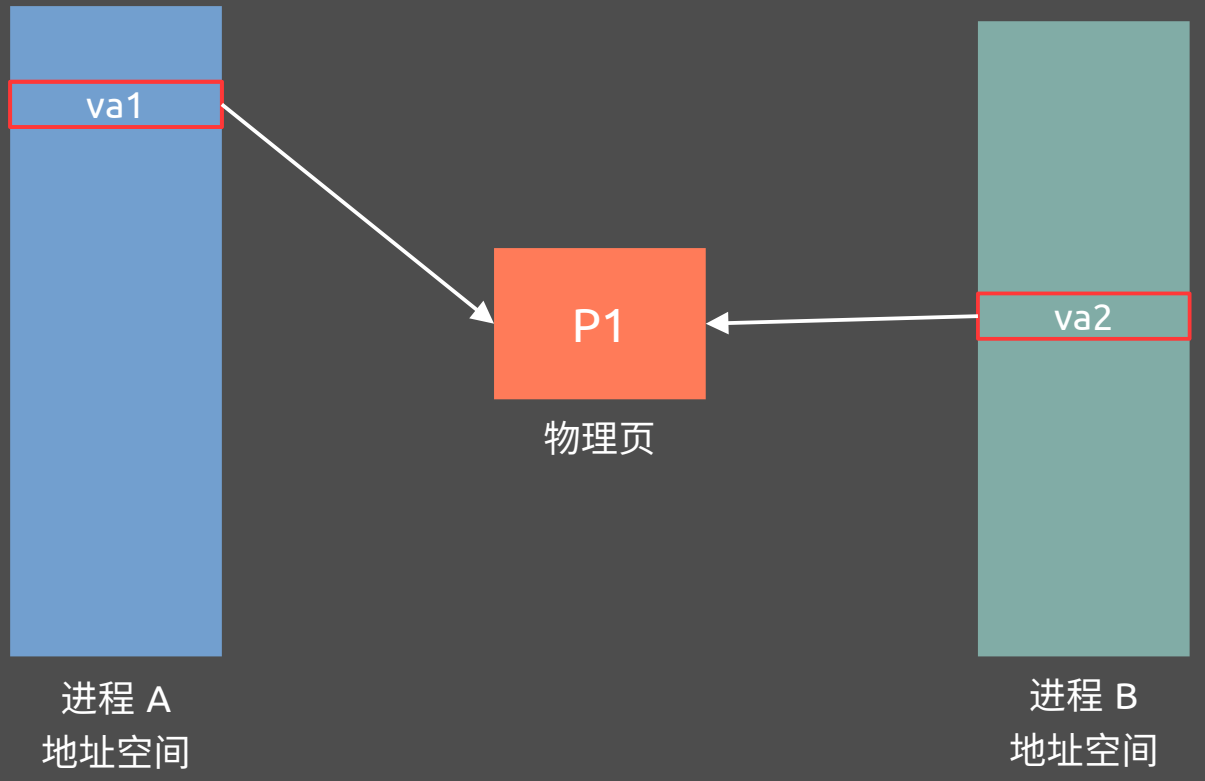

LunaixOS
从零开始
DEVELOPING YOUR OWN
自制操作系统
OPERATING SYSTEM FROM SCRATCH
多进程
进程中的内存
EP 10-2

并发物理内存访问



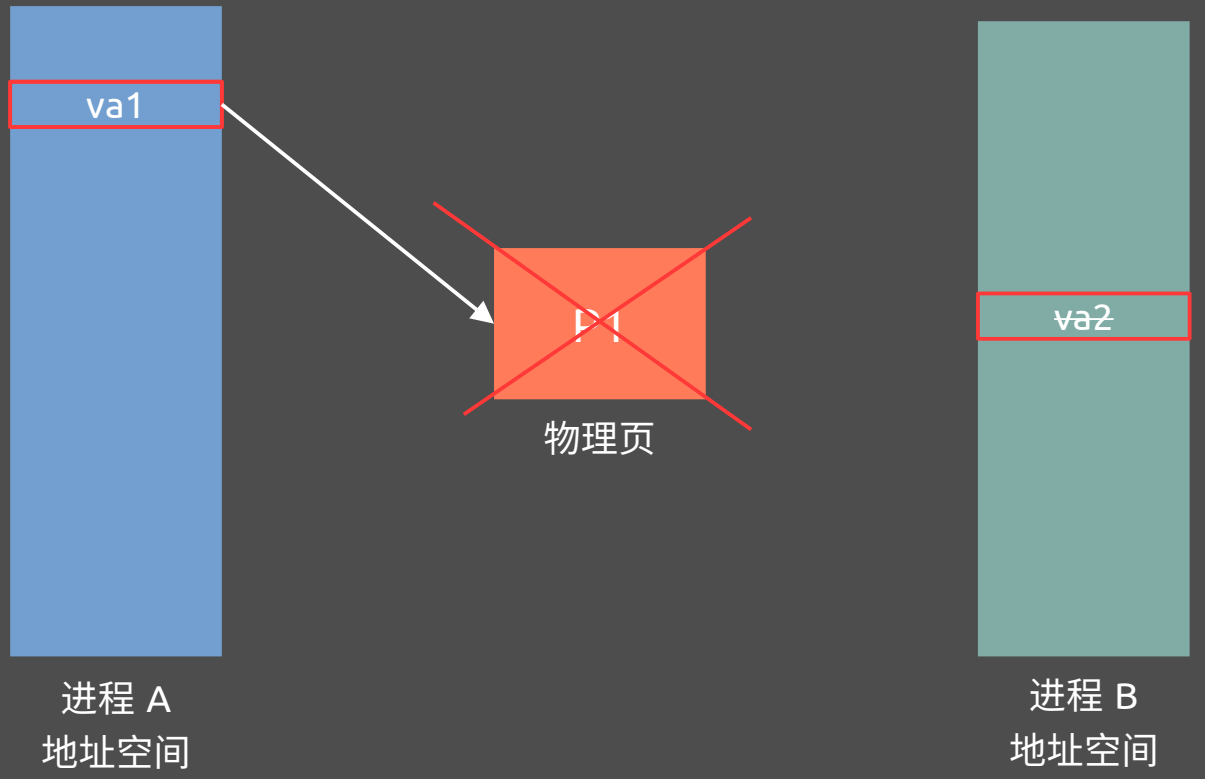
A 和 B 共享一个物理页

并发物理内存访问 - 共享内存释放



如果 B 释放掉映射 va2

并发物理内存访问 - 共享内存释放



按照现有设计， P1 也要被释放

看来要重构了

需要引入：**内存共享**和**引用计数**两个概念

```
You, 5 days ago | 1 author (You)
struct pp_struct {
    pid_t owner;
    uint32_t ref_counts;
    pp_attr_t attr;
};
```

记录物理页的所有者

引用次数（为 0 则被视为释放）

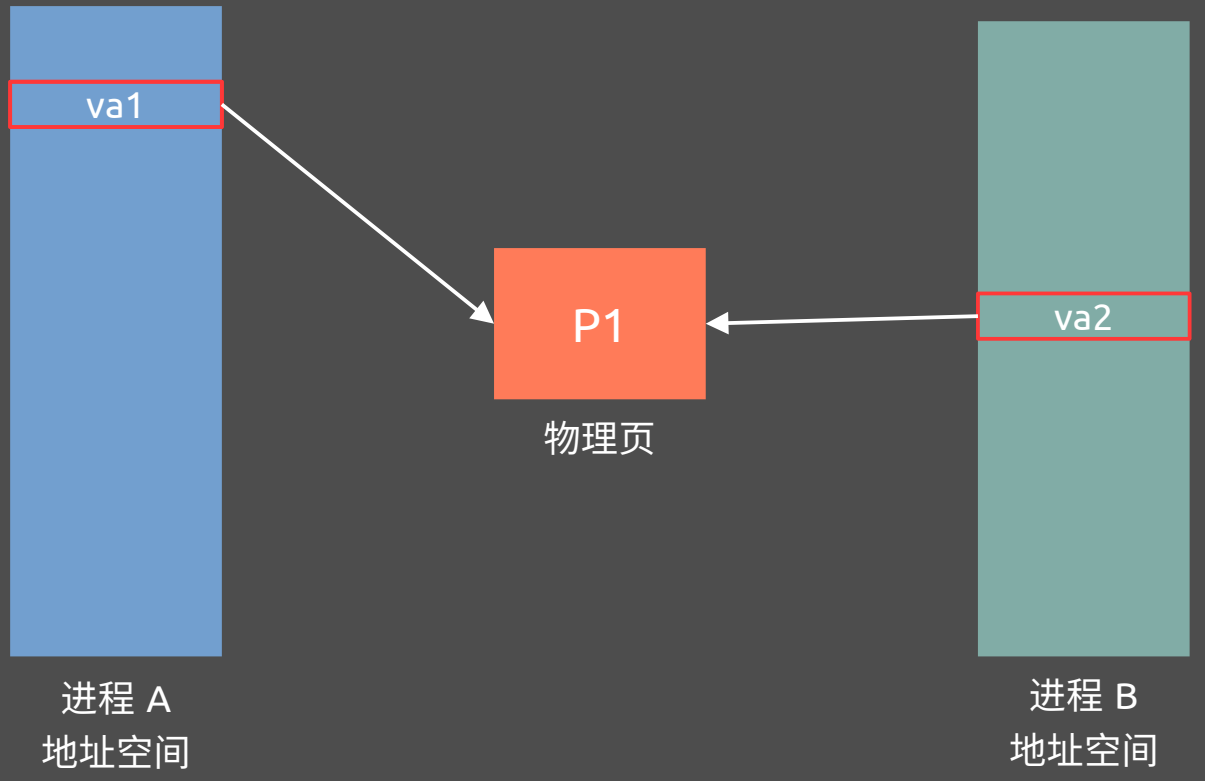
页属性（预留，暂时没用）

释放一个页就递减引用次数。

同理，分配一个页则递增。

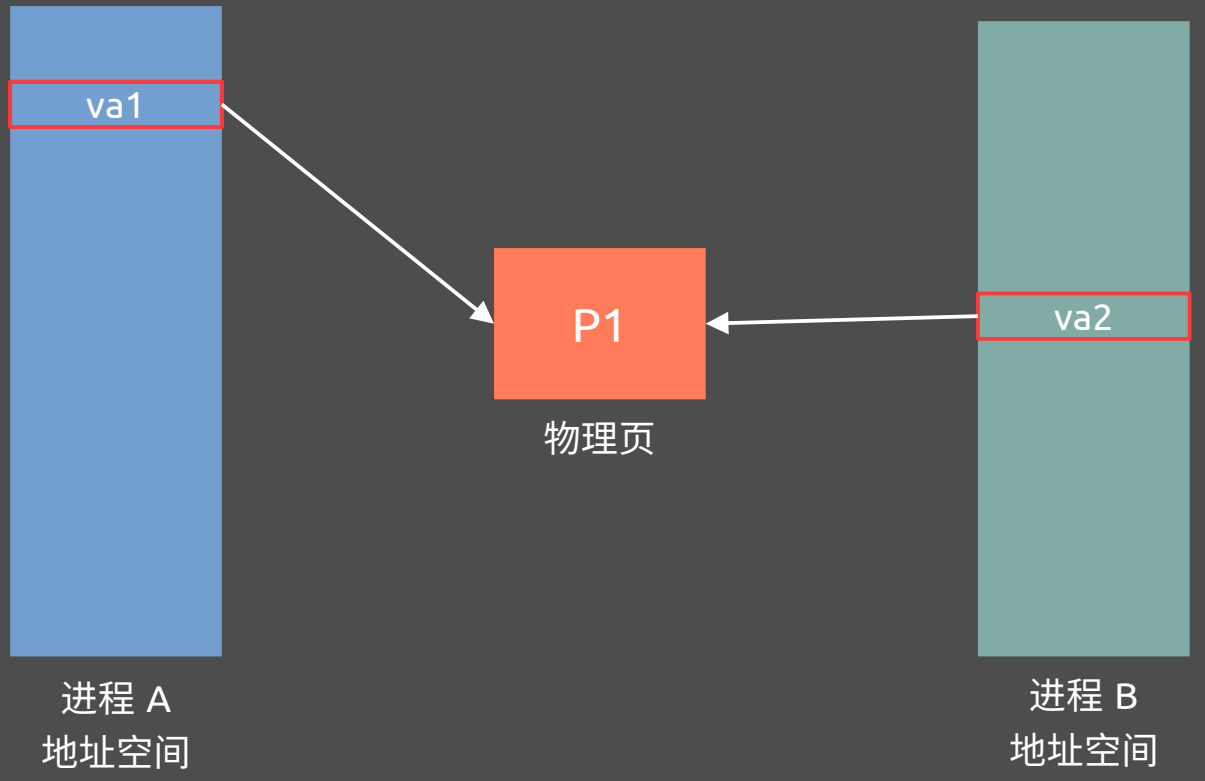
```
int
pmm_free_page(pid_t owner, void* page)
{
    struct pp_struct* pm = &pm_table[(intptr_t)page >> 12];

    // Is this a MMIO mapping or double free?
    if (((intptr_t)page >> 12) ≥ max_pg || !(pm->ref_counts)) {
        return 0;
    }

    pm->ref_counts--;
    return 1;
}
```

并发物理内存访问 - 读取



没有问题!

并发物理内存访问 - 写入?

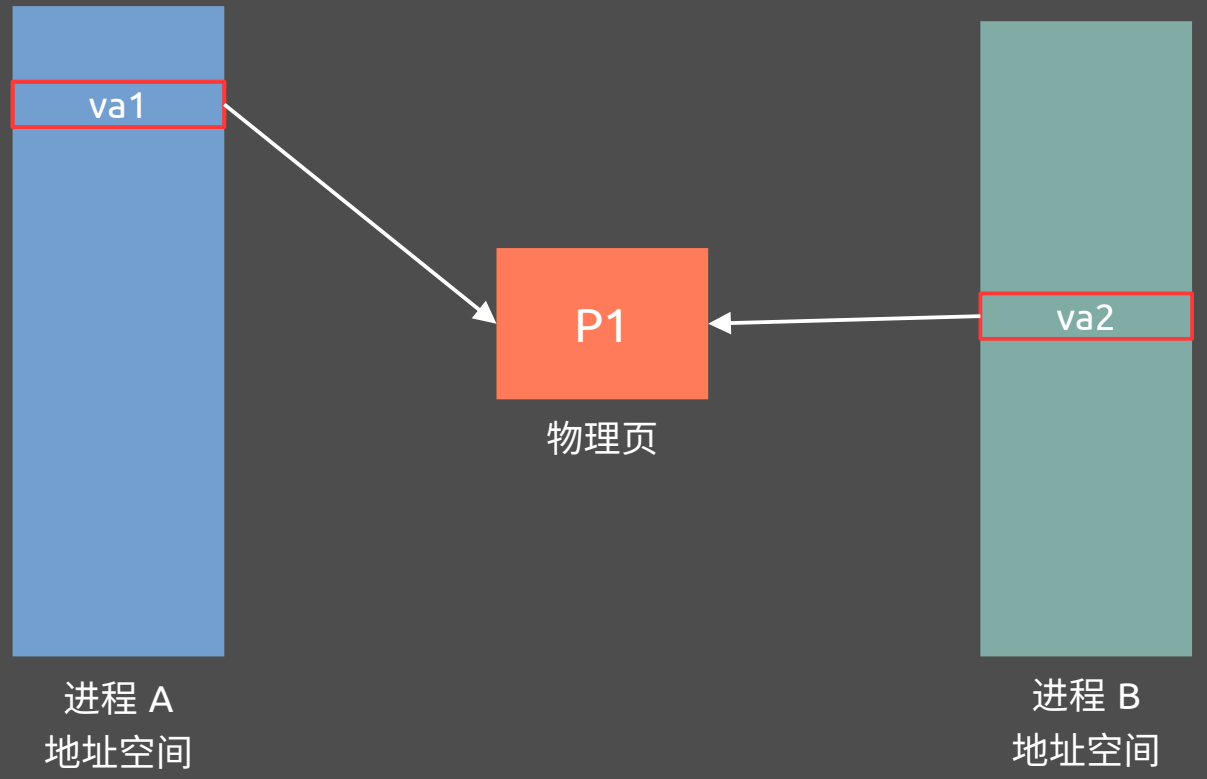


看情况!

并发物理内存访问 - 写入?

1. 该页完全私有
2. 该页共享读取
3. 该页共享写入

LunaixOS 内存的三个保护等级



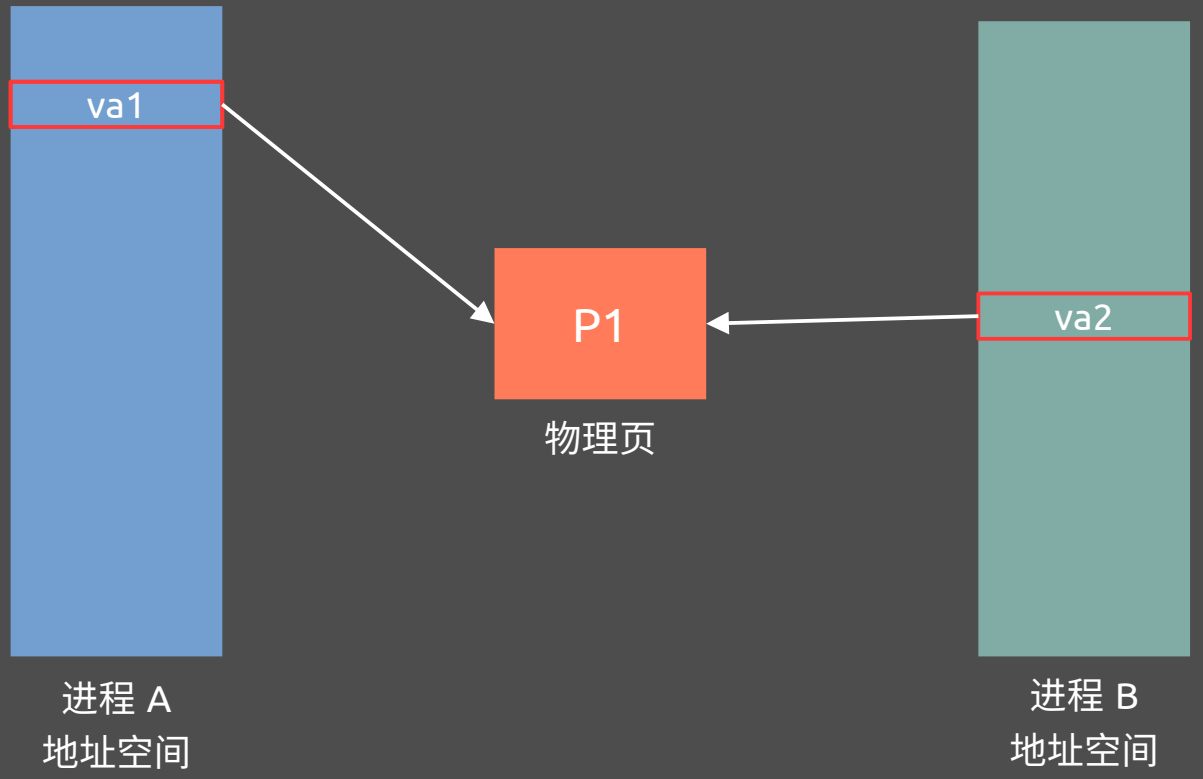
并发物理内存访问 - 写入?

私有就不会共享了!

1. ~~该页完全私有~~

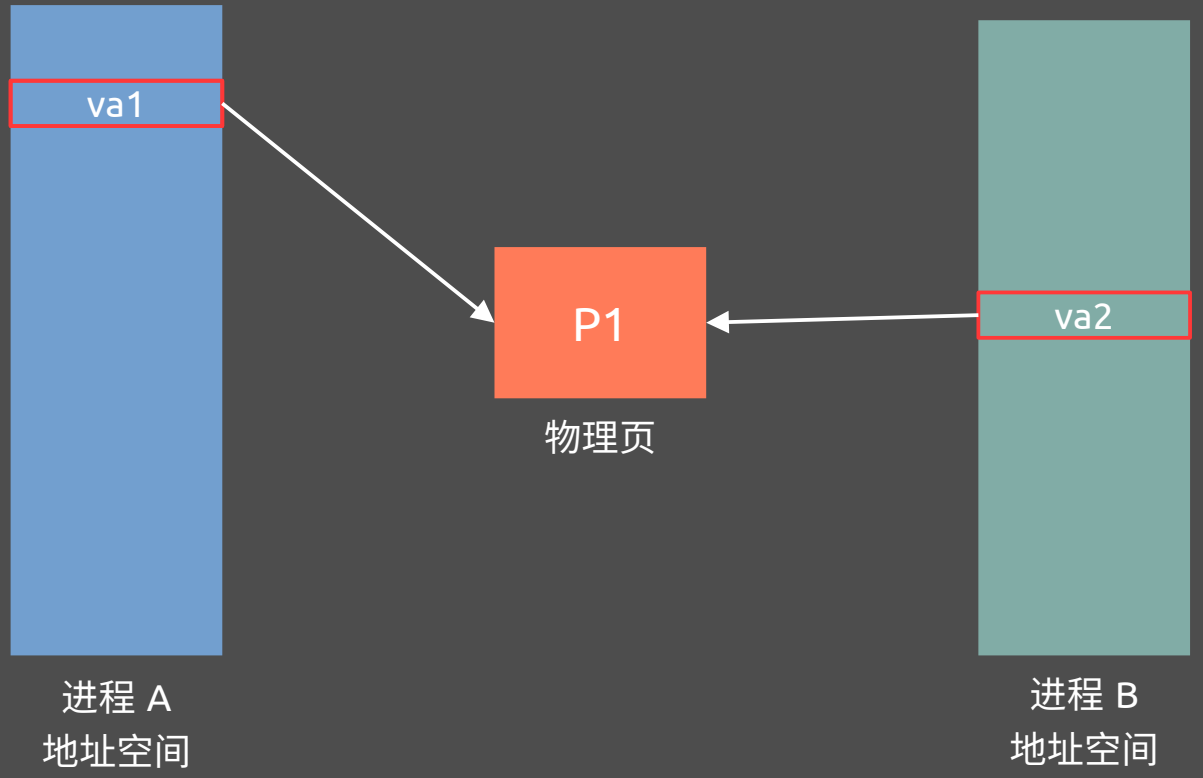
2. 该页共享读取

3. 改页共享写入



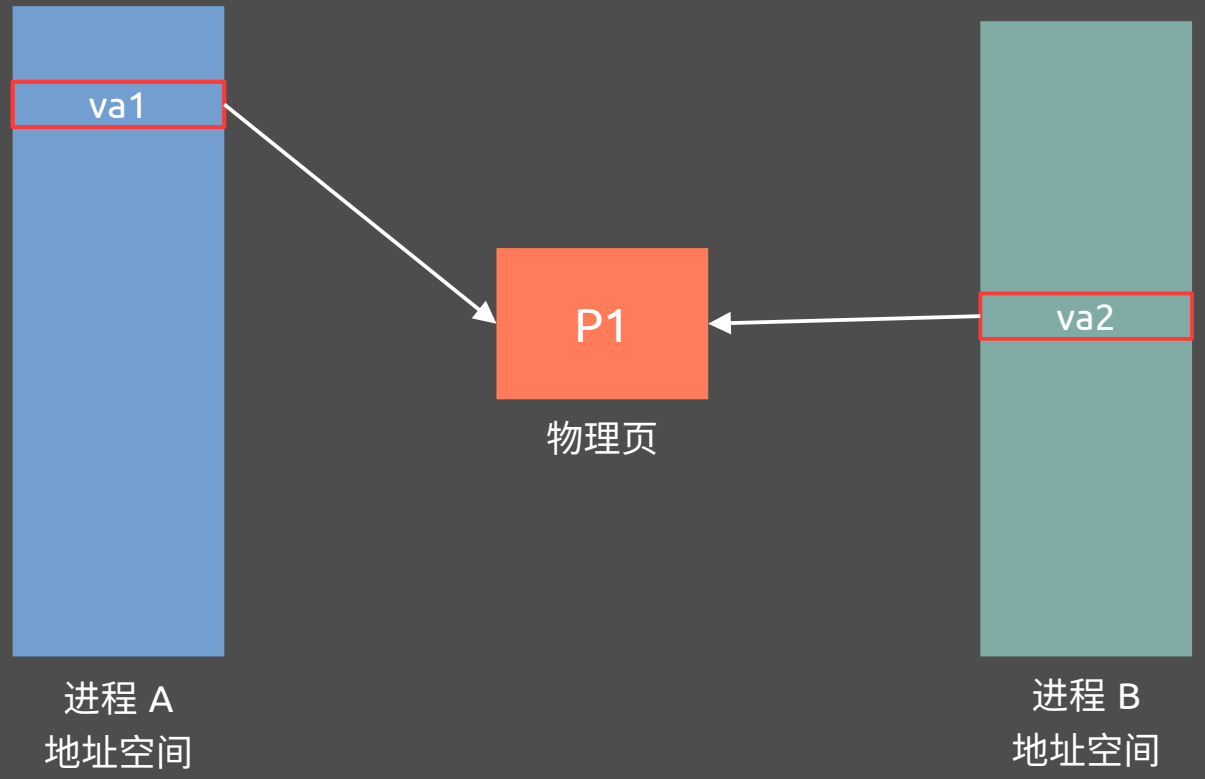
并发物理内存访问 - 共享写

一个进程写入的东西对另一个进程可见



并发物理内存访问 - 共享读

一个进程写入的东西对另一个进程不可见，但是写入前的内容可见

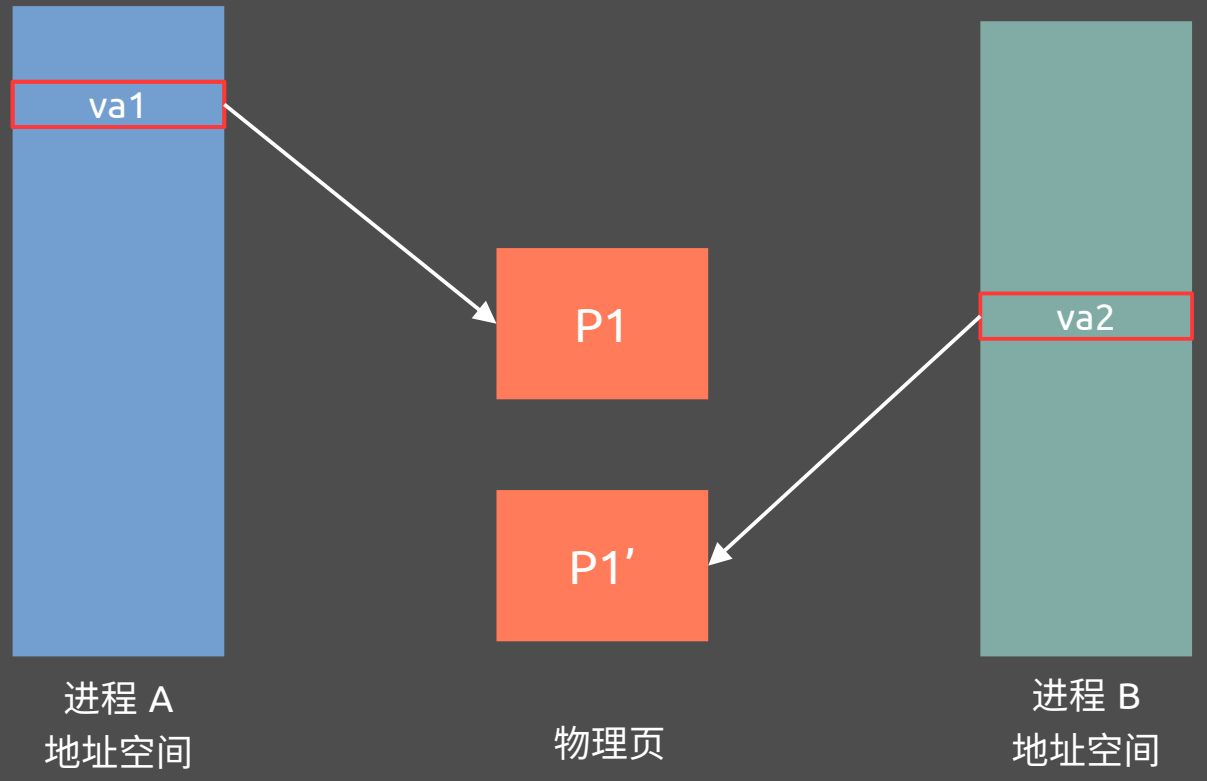


并发物理内存访问 - 共享读

一个进程写入的东西对另一个进程不可见，但是写入前的内容可见

写入则需要将该页复制一份，然后写入

写入时复制
Copy On Write, COW



我们如何记录这些权限?

```
You, 5 days ago | 1 author (You)
struct pp_struct {
    pid_t owner;
    uint32_t ref_counts;
    pp_attr_t attr;
};
```

塞进这里?

不行!

++，为什么?

不是所有的物理页在映射创建时就分配好的……

比如从磁盘加载几个 G 的大文件，不可能一下子全部载到物理内存中吧?

我们还是需要找另个地儿存放这些权限……

所以……一个进程的内存地图！

```
You, 3 days ago | 1 author (You)
struct proc_mm {
    heap_context_t u_heap;
    struct mm_region* regions;
};
```

用户栈

内存分区链表

```
You, 3 days ago | 1 author (You)
struct mm_region
{
    struct llist_header head;
    unsigned long start;
    unsigned long end;
    unsigned int attr;
};
```

这里记录了权限

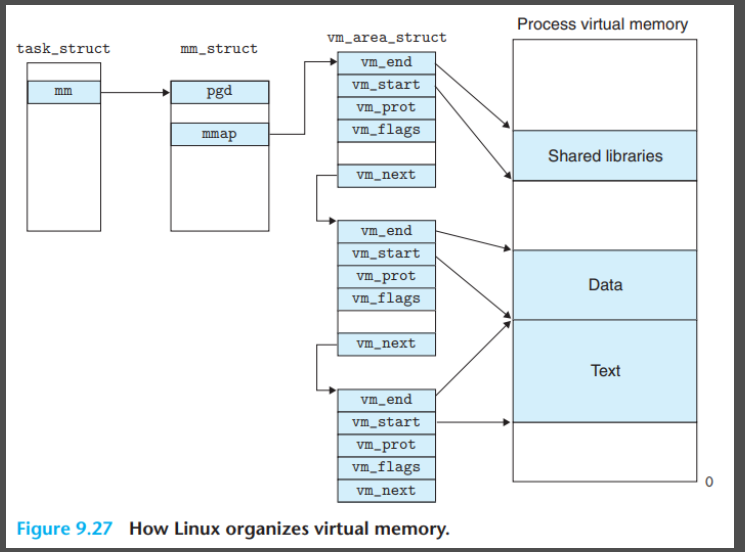


Figure 9.27 How Linux organizes virtual memory.

这和 Linux 的 vm_area_struct 有着异曲同工之妙

```
/** …
#define REGION_PRIVATE      0x0

/** …
#define REGION_RSHARED      0x1

/** …
#define REGION_WSHARED      0x2
```

一个进程的内存地图

通过往 mm_region 里添加属性，我们可以实现更加细粒度的，针对**进程内**的，访问控制

```
#define REGION_READ         (1 << 2)
#define REGION_WRITE        (1 << 3)
#define REGION_EXEC         (1 << 4)
```

内存区域可读

内存区域可写

内存区域可执行

如何确保这些权限不是摆设?

通过设置 PTE 的权限，使得 ：

PTE 权限 $\Longleftrightarrow$ `mm_region` 权限

CPU 鉴权失败或访问不存在的映射

$\Rightarrow$ 导致 Page Fault 中断 (int 14)

$\Rightarrow$ LunaixOS 进而确定具体的原因

1. 普通的页缺失：从磁盘加载或分配一个新页

2. 权限违反，或指向已知区域外的世界（野指针）：确定错误类型，并给出合适的错误报告

臭名昭著的 segmentation fault 就是这样出来的……

Page Fault 类型鉴别

```
intr_routine_page_fault (const isr_param* param)
{
    uintptr_t ptr = cpu_rcr2();
    if (!ptr) {
        goto segv_term;
    }

    struct mm_region* hit_region = region_get(__current, ptr);

    if (!hit_region) {
        // Into the void ...
        goto segv_term;
    }

    x86_pte_t* pte = CURPROC_PTE(ptr >> 12);
    if (*pte & PG_PRESENT) {
        if ((hit_region->attr & REGION_PERM_MASK) == (REGION_RSHARED | REGION_READ)) {
            // normal page fault, do COW
                uintptr_t pa = (uintptr_t)vmm_dup_page(__current->pid, PG_ENTRY_ADDR(*pte));
                pmm_free_page(__current->pid, *pte & ~0xFFF);
                *pte = (*pte & 0xFFF) | pa | PG_WRITE;
                return;
        }
        // impossible cases or accessing privileged page
        goto segv_term;
    }

    if (!(*pte)) {
        // Invalid location
        goto segv_term;
    }
    uintptr_t loc = *pte & ~0xfff;
    // a writable page, not present, pte attr is not null
    //   and no indication of cached page -> a new page need to be alloc
    if ((hit_region->attr & REGION_WRITE) && (*pte & 0xfff) && !loc) {
        uintptr_t pa = pmm_alloc_page(__current->pid, 0);
        *pte = *pte | pa | PG_PRESENT;
        return;
    }
    // page not present, bring it from disk or somewhere else
    __print_panic_msg("WIP page fault route", param);
    while (1);

segv_term:
    kprintf(KERROR "(pid: %d) Segmentation fault on %p (%p:%p)\n",
        __current->pid, ptr, param->cs, param->eip);
    terminate_proc(LXSEGFAULT);
    // should not reach
}
```

如何确保这些权限不是摆设?

三个 Page Fault 来源

Address of 4KB page frame			Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored												0	PTE: not present

U/S: 是否允许用户访问

R/W: 是否允许写入

1/0: 该页是否存在

```
#define REGION_READ     (1 << 2)
#define REGION_WRITE    (1 << 3)
#define REGION_EXEC     (1 << 4)
```

很可惜，只有 64 位模式下才支持“可执行”鉴权

实际在 LunaixOS 里面……

```
You, 13 hours ago | 1 author (You)
27  struct proc_info {
28      pid_t pid;
29      struct proc_info* parent;
30      isr_param intr_ctx;
31      struct proc_mm mm;
32      void* page_table;
33      time_t created;
34      uint8_t state;
35      int32_t exit_code;
36      [redacted]
37      [redacted]
38  };
```

父进程

中断上下文

页目录基地址

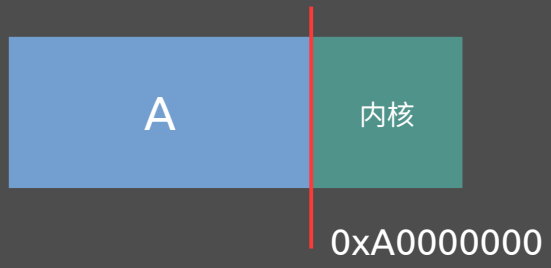
创建时间

状态

退出码

共享内核运行时

我们说过，每个进程都附带一份“不可见”的内核



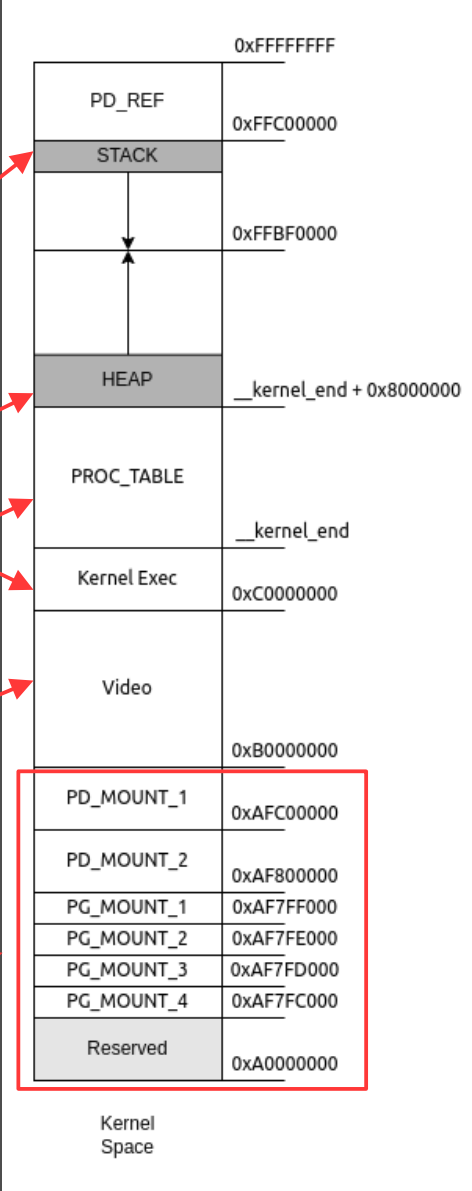
是通过虚拟映射共享？还是对于每个进程都全部复制一份？

共享内核运行时

目前， LunaixOS 的内核运行时主要有……

1. 内核的代码和数据
2. 内核栈
3. 内核堆
4. 进程表
5. CGA 显存
6. 页目录临时挂载点

1. ACPI 表结构
2. 计时器队列
3. mm_region



共享内核运行时

目前， LunaixOS 的内核运行时主要有……

进程共享读

进程私有

进程共享写

类似于临时变量，无需理会

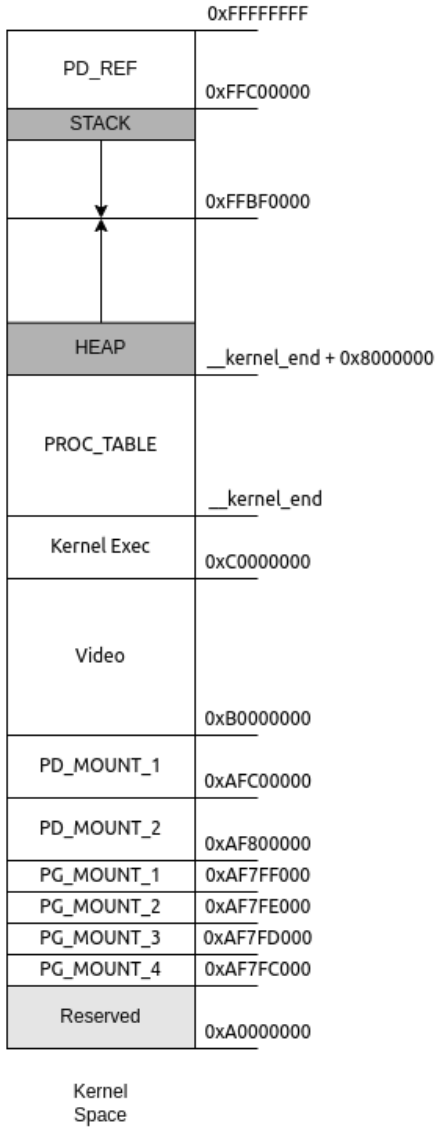
1. 内核的代码和数据
2. 内核栈
3. 内核堆
4. 进程表
5. CGA 显存
6. 页目录临时挂载点

为什么?

我们将会在实现系统调用时回答……

有必要吗?

微内核和宏内核的区别开始显现……



What's Next?

我们至少知道了该如何安排和管理进程中的内存。

但我们还不知道如何创建一个进程！

我们很快就会知道……


LunaixOS
从零开始
DEVELOPING YOUR OWN
自制操作系统
OPERATING SYSTEM FROM SCRATCH
多进程
To fork, or not to fork
EP 10-3

什么是 fork？

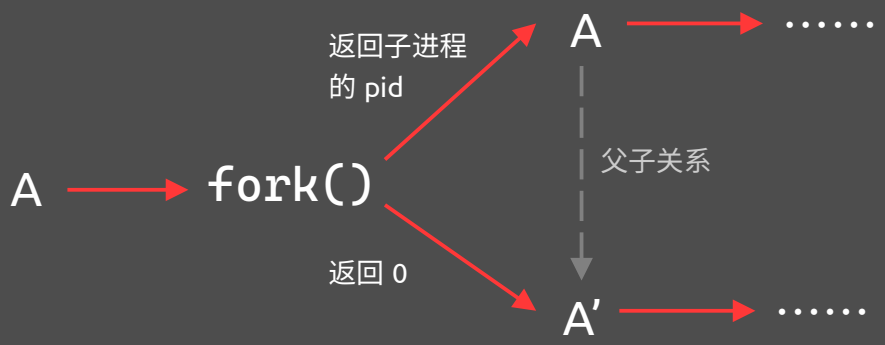
小马知道答案……

实用而古怪的 fork

粉色小马 Pinkie Pie 想起古老神秘的 `fork()` 传说，找到了隐藏在荆棘深处的系统调用，从而将自己一分为二

所以， `fork()` 的功能就是将进程一分为二。

也因为这一点， `fork()` 是仅有的能**返回两次**的函数！



实现 fork()

非常简单！把父进程的一切复制过来！

当然，还是需要注意……

显然， pid 等进程唯一的属性不能直接复制过来……

任何栈空间需要进行**完整**拷贝

对于任何读共享的内存区域，需要同时将父进程和子进程的对应映射标记为**只读**，从而保证 COW 的应用

*LunaixOS 特有：忽略掉私有空间的映射。

```
void* __dup_pagetable(pid_t pid, uintptr_t mount_point) {
    void* ptd_pp = pmm_alloc_page(pid, PP_FGPERSIST);
    x86_page_table* ptd = vmm_fmap_page(pid, PG_MOUNT_1, ptd_pp, PG_PREM_RW);
    x86_page_table* pptd = (x86_page_table*) (mount_point | (0x3FF << 12));

    for (size_t i = 0; i < PG_MAX_ENTRIES - 1; i++)
    {
        x86_pte_t ptde = pptd->entry[i];
        if (!ptde || !(ptde & PG_PRESENT)) {
            ptd->entry[i] = ptde;
            continue;
        }

        x86_page_table* ppt = (x86_page_table*) (mount_point | (i << 12));
        void* pt_pp = pmm_alloc_page(pid, PP_FGPERSIST);
        x86_page_table* pt = vmm_fmap_page(pid, PG_MOUNT_2, pt_pp, PG_PREM_RW);

        for (size_t j = 0; j < PG_MAX_ENTRIES; j++)
        {
            x86_pte_t pte = ppt->entry[j];
            pmm_ref_page(pid, pte & ~0xfff);
            pt->entry[j] = pte;
        }

        ptd->entry[i] = (uintptr_t)pt_pp | PG_PREM_RW;
    }

    ptd->entry[PG_MAX_ENTRIES - 1] = NEW_L1_ENTRY(T_SELF_REF_PERM, ptd_pp);

    return ptd_pp;
}
```

实现 fork() - 子进程 PCB 创建

```
pid_t dup_proc() {
    pid_t pid = alloc_pid();

    struct proc_info pcb = (struct proc_info) {
        .created = clock_systime(),
        .pid = pid,
        .mm = __current->mm,
        .intr_ctx = __current->intr_ctx,
        .parent = __current
    };

#ifdef USE_KERNEL_PG…
#else
    setup_proc_mem(&pcb, PD_REFERENCED);
#endif

    // 根据 mm_region 进一步配置页表
    if (!__current->mm.regions) {
        goto not_copy;
    }
```

实现 fork() - 复制与初始化地址空间

```
void setup_proc_mem(struct proc_info* proc, uintptr_t usedMnt) {
    // copy the entire kernel page table
    pid_t pid = proc->pid;
    void* pt_copy = __dup_pagetable(pid, usedMnt);

    vmm_mount_pd(PD_MOUNT_2, pt_copy);    // 将新进程的页表挂载到挂载点#2

    // copy the kernel stack
    for (size_t i = KSTACK_START >> 12; i <= KSTACK_TOP >> 12; i++)
    {
        volatile x86_pte_t *ppte = &PTE_MOUNTED(PD_MOUNT_2, i);

        /* ...
        cpu_invplg(ppte);          ! ! !

        x86_pte_t p = *ppte;
        void* ppa = vmm_dup_page(pid, PG_ENTRY_ADDR(p));
        *ppte = (p & 0xfff) | (uintptr_t)ppa;
    }
```

重写 PTE 时需要刷新 TLB 缓存，这一步非常重要！

实现 fork() - mm_region 复制

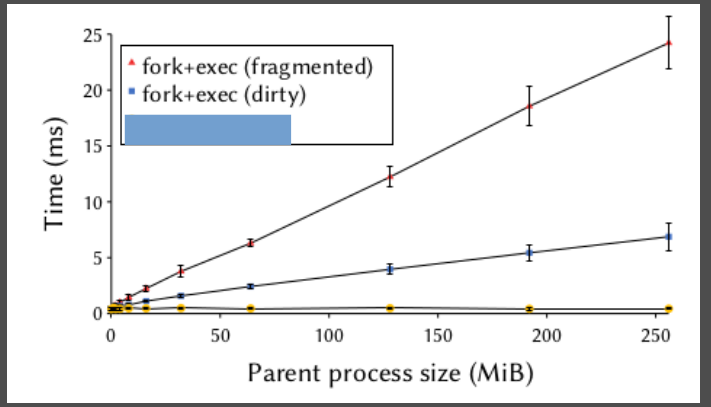
```
    struct mm_region *pos, *n;
    llist_for_each(pos, n, &__current->mm.regions->head, head) {
        region_add(&pcb, pos->start, pos->end, pos->attr);

        // 如果写共享，则不作处理。
        if ((pos->attr & REGION_WSHARED)) {
            continue;
        }

        uintptr_t start_vpn = PG_ALIGN(pos->start) >> 12;
        uintptr_t end_vpn = PG_ALIGN(pos->end) >> 12;
        for (size_t i = start_vpn; i < end_vpn; i++)
        {
            x86_pte_t *curproc =
                &((x86_page_table*)(PD_MOUNT_1 | ((i & 0xffc00) << 2)))->entry[i & 0x3ff];
            x86_pte_t *newproc =
                &((x86_page_table*)(PD_MOUNT_2 | ((i & 0xffc00) << 2)))->entry[i & 0x3ff];

            if (pos->attr == REGION_RSHARED) {
                // 如果读共享，则将两者的都标注为只读，那么任何写入都将会应用COW策略。
                *curproc = *curproc & ~PG_WRITE;
                *newproc = *newproc & ~PG_WRITE;      You, 1 second ago • Uncommitted chang
            }
            else {
                // 如果是私有页，则将该页从新进程中移除。
                *newproc = 0;
            }
        }
    }

not_copy:
    vmm_unmount_pd(PD_MOUNT_2);

    // 正如同fork，返回两次。
    pcb.intr_ctx.registers.eax = 0;

    push_process(&pcb);

    return pid;
```

返回两次的奥秘 →

天使与魔鬼—— fork 的两面性……

fork 非常的简洁，使用起来简单、灵活；实现起来更简单！

但任何好东西都有它的代价……



从内核的角度

#1 低效与极大的内存压力

从用户的角度

#1 容易出现竞态条件（ Race Condition ）！
#2 代码可读性非常的差
#3 需要在合适的时候使用 _exit ，否则……

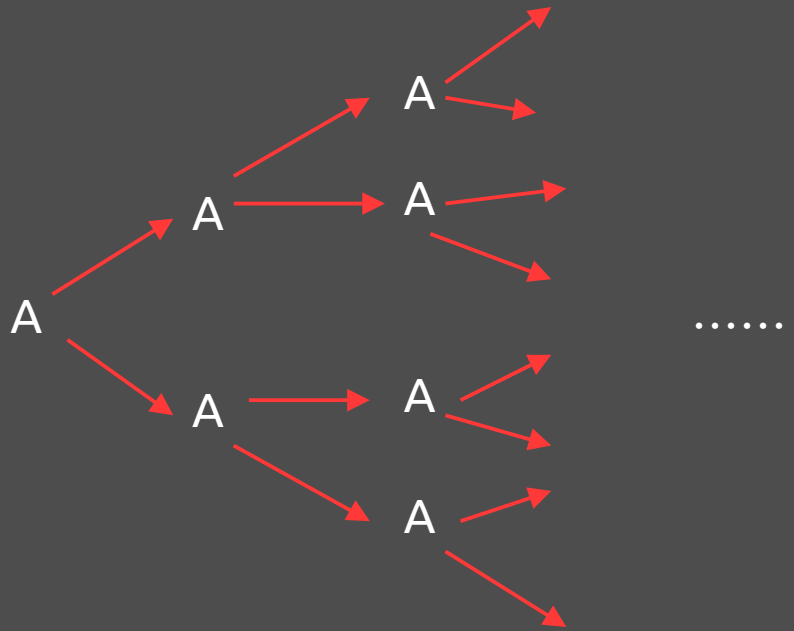
```
for (size_t i = 0; i < 10; i++)
{
    pid_t pid = 0;
    if (!(pid = fork())) {
        sleep(i);
        tty_put_char('0'+i);
        tty_put_char('\n');
    }
    kprintf(KINFO "Forked %d\n", pid);
}
```

这里的 fork() 总共被调用几次?

好心办错事—— fork 炸弹

Pinkie Pie 就犯了这种低级的编程错误！她只想 fork 出许多自己，而去多陪陪自己的朋友。却在操作过程中失去控制，最终过载了整个小马谷，耗尽了系统的资源……

```
for (;;) {
    fork();
}
```



另一条道路： posix_spawn()

从零创建一个进程：

地址空间直接派生自内核

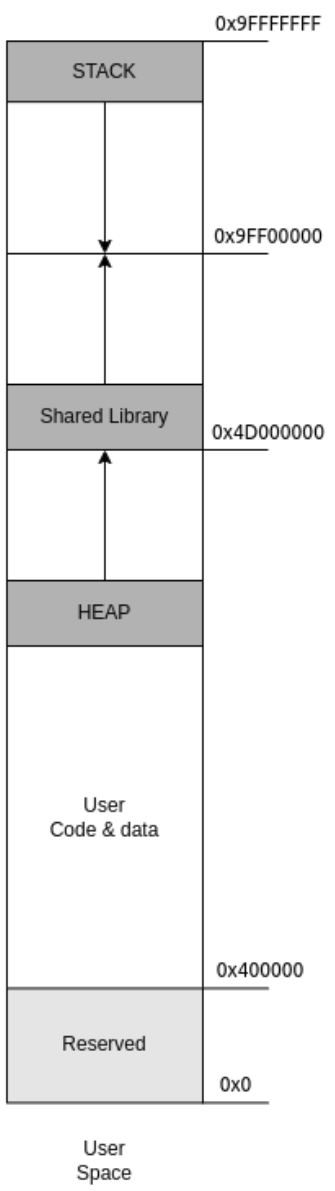
只需分配用户栈区域和代码区域。

比 fork 要轻量许多！

但这并不说明他能取代 fork！

posix_spawn 的近亲： Windows 下的 CreateProcess

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```



在 LunaixOS 中……

proc0 （LunaixOS 的零号进程）就是通过 **类似** 的方法创建出来的!

```
void spawn_proc0() {
    struct proc_info kinit;

    init_proc(&kinit);
    kinit.intr_ctx = (isr_param) {
        .registers.esp = KSTACK_TOP - 20,
        .cs = KCODE_SEG,
        .eip = (void*)__proc0,
        .ss = KDATA_SEG,
        .eflags = cpu_reflags()
    };

    setup_proc_mem(&kinit, PD_REFERENCED);
```

```
    asm volatile(
        "movl %%cr3, %%eax\n"
        "movl %%esp, %%ebx\n"
        "movl %0, %%cr3\n"
        "movl %1, %%esp\n"
        "pushf\n"
        "pushl %2\n"
        "pushl %3\n"
        "pushl $0\n"
        "pushl $0\n"
        "movl %%eax, %%cr3\n"
        "movl %%ebx, %%esp\n"
        ::
            "r"(kinit.page_table),
            "i"(KSTACK_TOP),
            "i"(KCODE_SEG),
            "r"(kinit.intr_ctx.eip)
        :
            "%eax", "%ebx", "memory"
    );

    // 向调度器注册进程，然后这里阻塞等待调度器调度就好了。
    push_process(&kinit);
```

等等，零号进程?

其实只是个占位进程。

做的唯一的一件事就是 fork 进 lxinit 进程（也就是 LunaixOS 下的 initd！）

主要目的就是为了让调度器有事可做。

```
void __proc0() {
    if (!fork()) {
        asm ("jmp _lxinit_main");
    }

    asm ("1: jmp 1b");
}
```

没错，就这么点儿东西……

这是什么意思?

我们会在实现 sleep 系统调用时看到……

为什么说是“类似”？

回想一下上述的代码：

复制了完整的内核地址空间。

代码和数据段是共用的。

```
void setup_proc_mem(struct proc_info* proc, uintptr_t usedMnt) {
    // copy the entire kernel page table
    pid_t pid = proc->pid;
    void* pt_copy = __dup_pagetable(pid, usedMnt);
```

```
void spawn_lxinit() {
    struct proc_info kinit;

    memset(&kinit, 0, sizeof(kinit));
    kinit.parent = (void*)0;
    kinit.pid = 1;
    kinit.intr_ctx = (isr_param) {
        .registers.esp = KSTACK_TOP - 20,
        .cs = KCODE_SEG,
        .eip = (void*)_lxinit_main,
        .ss = KDATA_SEG,
        .eflags = cpu_reflags()
    };

    setup_proc_mem(&kinit, PD_REFERENCED);
```

唯一不同的地方在于它使用了新的栈，并且我们手动设置了上下文环境……

```
asm volatile(
    "movl %%cr3, %%eax\n"
    "movl %%esp, %%ebx\n"
    "movl %0, %%cr3\n"
    "movl %1, %%esp\n"
    "pushf\n"
    "pushl %2\n"
    "pushl %3\n"
    "pushl $0\n"
    "pushl $0\n"
    "movl %%eax, %%cr3\n"
    "movl %%ebx, %%esp\n"
    ::"r"(kinit.page_table),
      "i"(KSTACK_TOP),
      "i"(KCODE_SEG),
      "r"(kinit.intr_ctx.eip)
    :"%eax", "%ebx", "memory"
);

// 向调度器注册进程，然后这里阻塞等待调度器调度就好了。
push_process(&kinit);
```

从某种意义上，这还是 fork！

但我们在讨论 posix_spawn……

posix_spawn 其实本质上就是 fork ，只不过 fork 的对象，和内存的多少不一样……

严谨的来说， posix_spawn 其实可以看成 fork+execve 的组合函数。

要想真正实现 spawn ，我们得需要实现 execve ，而实现 execve ，我们需要一个基本的文件系统……

不过这不妨碍我们先写个四分之一成品。 :)

execve?

我们会在实现文件系统后深入探讨

To fork, or not to fork. That is the question.

A fork() in the road

Andrew Baumann
Microsoft Research

Jonathan Appavoo
Boston University

Orran Krieger
Boston University

Timothy Roscoe
ETH Zurich

ABSTRACT

The received wisdom suggests that Unix's unusual combination of `fork()` and `exec()` for process creation was an inspired design. In this paper, we argue that fork was a clever hack for machines and programs of the 1970s that has long outlived its usefulness and is now a liability. We catalog the ways in which fork is a terrible abstraction for the modern programmer to use, describe how it compromises OS implementations, and propose alternatives.

As the designers and implementers of operating systems, we should acknowledge that fork's continued existence as a first-class OS primitive holds back systems research, and deprecate it. As educators, we should teach fork as a historical artifact, and not the first process creation mechanism students encounter.

ACM Reference Format:
Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. 2019. A `fork()` in the road. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3317550.3321435

1 INTRODUCTION

When the designers of Unix needed a mechanism to create processes, they added a peculiar new system call: `fork()`. As every undergraduate now learns, fork creates a new process identical to its parent (the caller of fork), with the exception of the system call's return value. The Unix idiom of `fork()` followed by `exec()` to execute a *different* program in the child is now well understood, but still stands in stark contrast to process creation in systems developed independently of

50 years later, fork remains the default process creation API on POSIX: Atlidakis et al. [8] found 1304 Ubuntu packages (7.2% of the total) calling fork, compared to only 41 uses of the more modern `posix_spawn()`. Fork is used by almost every Unix shell, major web and database servers (e.g., Apache, PostgreSQL, and Oracle), Google Chrome, the Redis key-value store, and even Node.js. The received wisdom appears to hold that fork is a good design. Every OS textbook we reviewed [4, 7, 9, 35, 75, 78] covered fork in uncritical or positive terms, often noting its "simplicity" compared to alternatives. Students today are taught that "the `fork` system call is one of Unix's great ideas" [46] and "there are lots of ways to design APIs for process creation; however, the combination of `fork()` and `exec()` are simple and immensely powerful … the Unix designers simply got it right" [7].

Our goal is to set the record straight. Fork is an anachronism: a relic from another era that is out of place in modern systems where it has a pernicious and detrimental impact. As a community, our familiarity with fork can blind us to its faults (§4). Generally acknowledged problems with fork include that it is not thread-safe, it is inefficient and unscalable, and it introduces security concerns. Beyond these limitations, fork has lost its classic simplicity; it today impacts all the other operating system abstractions with which it was once orthogonal. Moreover, a fundamental challenge with fork is that, since it conflates the process and the address space in which it runs, fork is hostile to user-mode implementation of OS functionality, breaking everything from buffered IO to kernel-bypass networking. Perhaps most problematically, fork *doesn't compose*—every layer of a system from the kernel to the smallest user-mode library must support it.

We illustrate the havoc fork wreaks on OS implementations using our experiences with prior research systems (§5),

这一直是一个非常火热的 OS 话题

毕竟 HotOS :-)

甚至有学者专门写了一篇 Opinion 来讨论这件事。

大部分对 fork 的忧虑主要集中在：

1. fork 的效率（特别是用来执行另一个程序）。
2. fork 给内存带来的压力。
3. fork 的安全性
4. 过分依赖虚拟内存
5. 对多线程的不友好

What's Next?

我们似乎距离实现真正的操作系统越来越近了。 LunaixOS 终于实现了大名鼎鼎的 fork() 。

但目前，进程还是运行最高权限的 Ring 0 级，也就是内核空间。如果我们想要运行不受信任的外来程序，我们需要去往权限更低的 Ring 3……

我们将会在下一期中探索这个神秘的 Ring 3 世界，也就是我们所熟悉的用户空间……

```
c010606f: 55                push   %ebp
c0106070: 89 e5             mov    %esp,%ebp
c0106072: 56                push   %esi
c0106073: 53                push   %ebx
c0106074: 83 ec 0c          sub    $0xc,%esp
c0106077: 68 44 a2 12 c0    push   $0xc012a244
c010607c: e8 ed 1c 00 00    call   c0107d6e <printf>
c0106081: be 00 60 10 00    mov    $0x106000,%esi
c0106086: c1 ee 0c          shr    $0xc,%esi
c0106089: 83 c4 08          add    $0x8,%esp
c010608c: 56                push   %esi
c010608d: 68 6c a2 12 c0    push   $0xc012a26c
c0106092: e8 d7 1c 00 00    call   c0107d6e <printf>
c0106097: 83 c4 10          add    $0x10,%esp
c010609a: bb 00 00 00 00    mov    $0x0,%ebx
c010609f: eb 14             jmp    c01060b5 <_kernel_post_init+0x46>
c01060a1: 89 d8             mov    %ebx,%eax
c01060a3: c1 e0 0c          shl    $0xc,%eax
c01060a6: 83 ec 0c          sub    $0xc,%esp
c01060a9: 50                push   %eax
c01060aa: e8 9f 0a 00 00    call   c0106b4e <vmm_unmap_page>
c01060af: 83 c3 01          add    $0x1,%ebx
c01060b2: 83 c4 10          add    $0x10,%esp
c01060b5: 39 f3             cmp    %esi,%ebx
c01060b7: 72 e8             jb     c01060a1 <_kernel_post_init+0x32>
c01060b9: e8 54 ...         call   <kalloc_init>
c01060be: 85 c0 ...
c01060c0: 74 07 ...                <_kernel_post_init+0x5a>
c01060c2: 8d 65 f8 ...             0x-8(%ebp),%esp
c01060c5: 5b                pop    %ebx
c01060c6: 5e                pop    %esi
c01060c7: 5d                pop    %ebp
c01060c8: c3                ret
c01060c9: 83 ec 04          sub    $0x4,%esp
c01060cc: 6a 40             push   $0x40
c01060ce: 68 af a0 12 c0    push   $0xc012a0af
```

LunaixOS

从零开始

自制操作系统

多进程

进入用户空间

EP 10-4

让用户们自娱自乐。

能力越大，责任越大。内核的世界要求你万事都要小心谨慎，
稍不注意，就有可能……

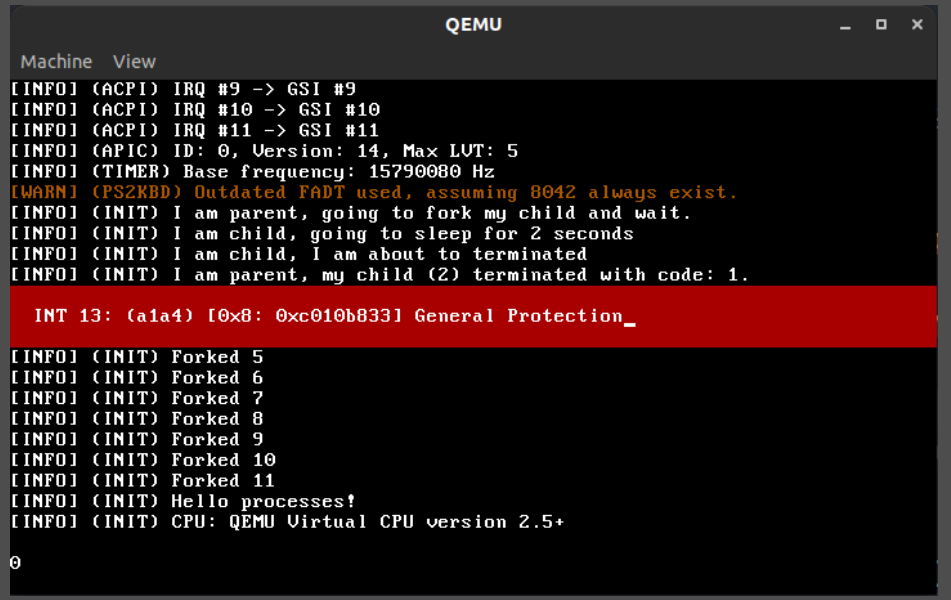
但电脑是给人用的，而不是程序员的专属！

我们内核需要……

限制任何正在运行的代码的能力

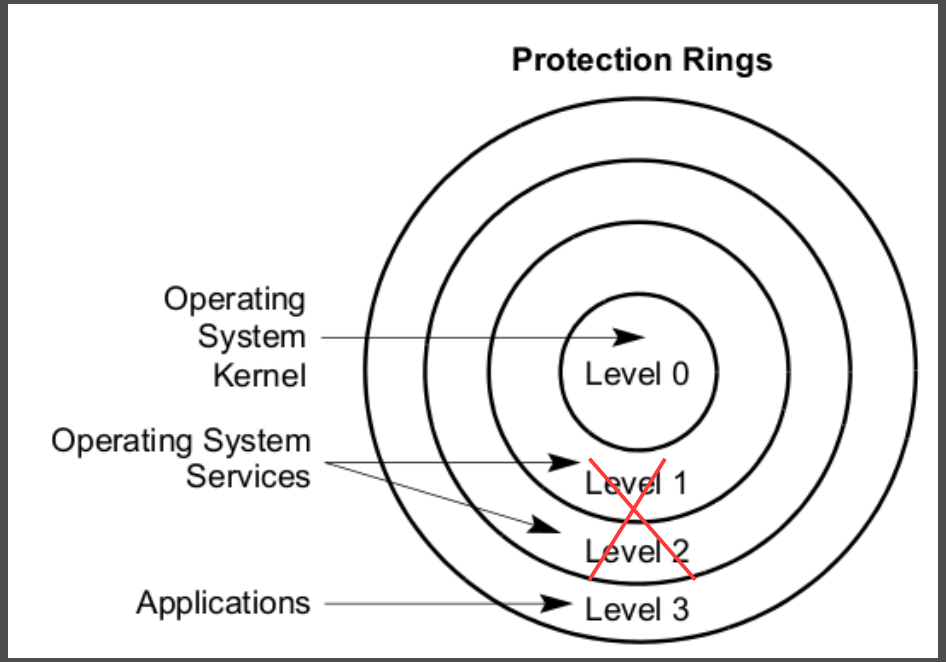
可是还要有足够的能力去管理计算机！

防止用户做傻事！



一个圈子里还要分个三六九等!

迅速回忆一下 x86 平台下的特权级划分



熟悉吧?

不同的 Ring 决定了……

一些指令能不能使用

虚拟页能不能被访问

某个中断能不能被调用

和 Linux 系统一样， LunaixOS 只使用 Level 0 和 Level 3

也是大多数人都熟悉的 Ring 0 和 Ring 3

进入 Ring 3！

太简单了！

往段寄存器里载 Ring3 的段选择子就好了

```
14  #define KCODE_SEG       0x08
15  #define KDATA_SEG       0x10
16  #define UCODE_SEG       0x1B
17  #define UDATA_SEG       0x23
18  #define TSS_SEG         0x28
```

需要调整的寄存器：

```
%cs = $UCODE_SEG
%ds
%es
%ss  $UDATA_SEG
%fs
%gs
```

等等！好像简单的有点不对劲……

进入用户空间远不止换几个寄存器那么轻松……

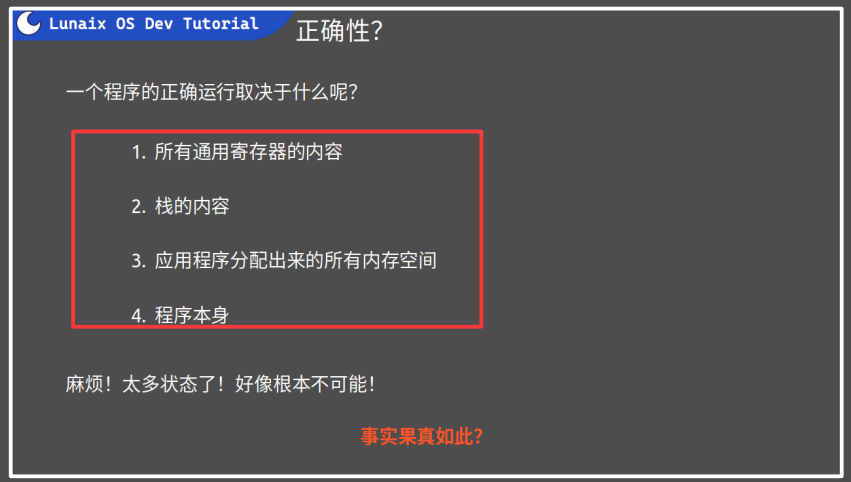
很多问题需要考虑：

1. 当中断产生时，如何切换回内核空间？

2. 进程运行时需要与内核完全分开，怎么做？

3. 从第 1 和第 2 点， 如何进行**进程内**的“上下文切换”？

先考虑一下运行时隔离



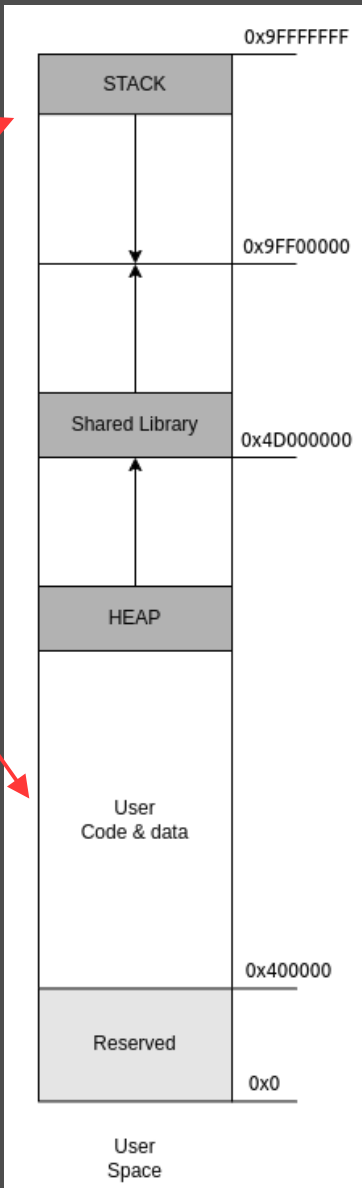
回忆一下我们在第一个视频里提到的

寄存器已经由上下文切换实现隔离

栈隔离——需要**用户栈!**

程序本身——需要一个专门的区域来存放

内存区域隔离——使用 mm_region ，将内核的区域排除在外



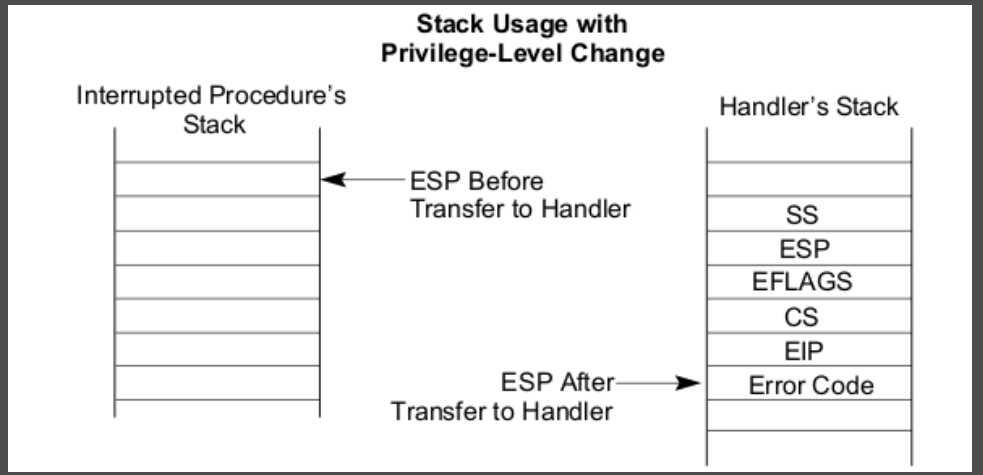
进程内的“上下文切换”

切换操作模式 ⇒ 切换所使用的栈。

中断已经帮我们处理好了！

当 ISR 的特权级 (RPL) 与当前特权级 (CPL) 不一致时……

1. CPU 切换到**内核栈**
2. CPU 推入先前的使用的 SS 和 ESP



CPU 如何知道哪个是我们的内核栈？ ⇒ TSS

被“误用”的 TSS

TSS 在 LunaixOS 中是被“误用”的！因为……

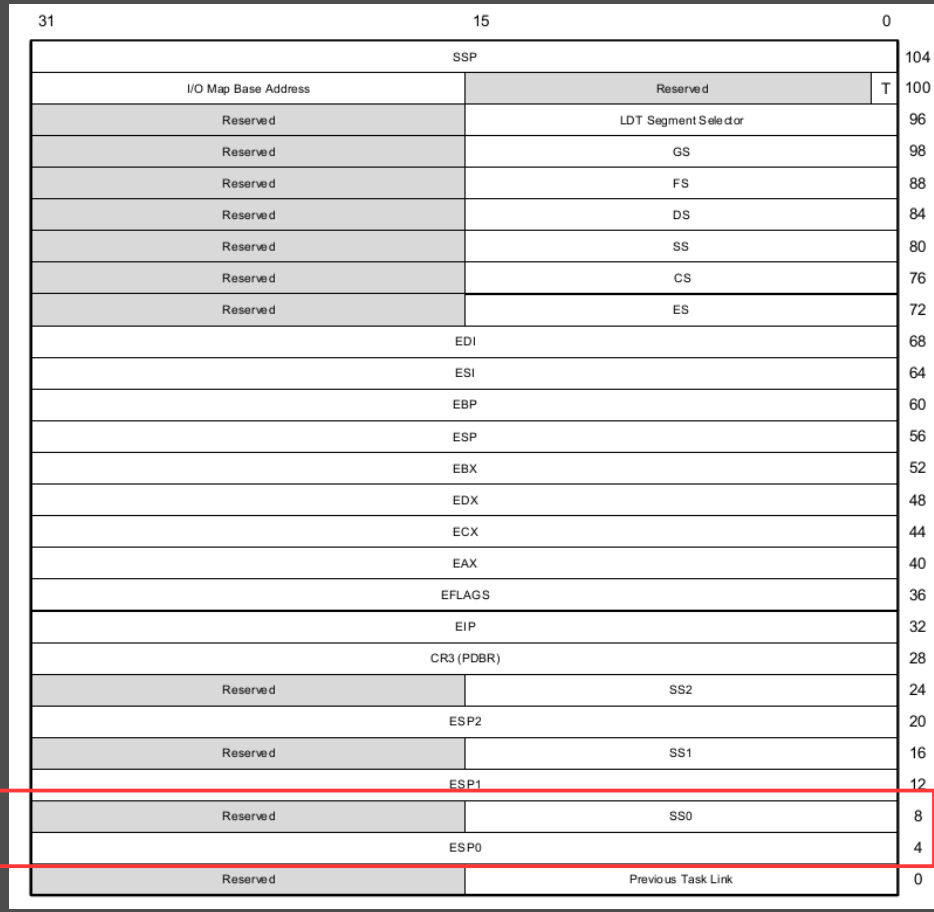
TSS 的初衷便是帮助我们进行上下文切换。每个进程对应一个 TSS

CPU 会在我们执行特定的动作时，会将当前进程上下文保存到 TSS 里，然后切换。

于是，我们可以看到两种上下文切换的方式：

1. 使用 TSS 的**硬件上下文切换**
2. LunaixOS 目前使用的**软件上下文切换**

CPU 会将这两个值作为内核栈的 SS : ESP



在 LunaixOS 中……

```
struct x86_tss _tss = {
    .link = 0,
    .esp0 = KSTACK_START,
    .ss0  = KDATA_SEG
};
```

```
void
_init_gdt() {
    _set_gdt_entry(0, 0, 0, 0);
    _set_gdt_entry(1, 0, 0xfffff, SEG_R0_CODE);
    _set_gdt_entry(2, 0, 0xfffff, SEG_R0_DATA);
    _set_gdt_entry(3, 0, 0xfffff, SEG_R3_CODE);
    _set_gdt_entry(4, 0, 0xfffff, SEG_R3_DATA);
    _set_gdt_entry(5, &_tss, sizeof(struct x86_tss) - 1, SEG_TSS);
}
```

和其他段描述符一样， TSS 描述符注册在 GDT 里面

```
/* 加载TSS段选择器 */
movw $TSS_SEG, %ax
ltr %ax
```

就这么点儿?

对！就这么点儿。

进入用户空间其实只是权限级的切换。而上下文切换的实现，已经帮助我们解决了这里面的大部分的问题。

有没有具体的代码?

暂时还没有！我们将会留到实现信号机制（ Signal ）的时候统一测试……

但进入用户空间的原理相当的简单，实现起来也只是改几个常量值的事情。

What's Next ?

我们简单的讨论了一下用户空间切换的理论和实现时的注意要点。

内核空间和用户程序空间也就由此彻底的隔离开了。

可是我们还是希望，用户程序能在一定程度上和内核进行**受保护的**交互。

我们将在下期里实现系统调用……


c010606f: 55 push %ebp
c0106070: 89 e5 mov %esp,%ebp
c0106072: 56 push %esi
c0106073: 53 push %ebx
c0106074: 83 ec 0c sub $0xc,%esp
c0106077: 68 44 a2 12 c0 push $0xc012a244
c010607c: e8 ed 1c 00 00 call c0107d6e <printf>
c0106081: be 00 60 10 00 mov $0x106000,%esi
c0106086: c1 ee 0c shr $0xc,%esi
c0106089: 83 c4 08 add $0x8,%esp
c010608c: 56 push %esi
c010608d: 68 6c a2 12 c0 push $0xc012a26c
c0106092: e8 d7 1c 00 00 call c0107d6e <printf>
c0106097: 83 c4 10 add $0x10,%esp
c010609a: bb 00 00 00 00 mov $0x0,%ebx
c010609f: eb 14 jmp c01060b5 <_kernel_post_init+0x46>
c01060a1: 89 d8 mov %ebx,%eax
c01060a3: c1 e0 0c shl $0xc,%eax
c01060a6: 83 ec 0c sub $0xc,%esp
c01060a9: 50 push %eax
c01060aa: e8 9f 0a 00 00 call c0106b4e <vmm_unmap_page>
c01060af: 83 c3 01 add $0x1,%ebx
c01060b2: 83 c4 10 add $0x10,%esp
c01060b5: 39 f3 cmp %esi,%ebx
c01060b7: 72 e8 jb c01060a1 <_kernel_post_init+0x32>
c01060b9: e8 54 call <kalloc_init>
c01060be: 85 c0 test
c01060c0: 74 07 je c01060c9 <_kernel_post_init+0x5a>
c01060c2: 8d 65 f8 lea -0x8(%ebp),%esp
c01060c5: 5b pop %ebx
c01060c6: 5e pop %esi
c01060c7: 5d pop %ebp
c01060c8: c3 ret
c01060c9: 83 ec 04 sub $0x4,%esp
c01060cc: 6a 40 push $0x40
c01060ce: 68 af a0 12 c0 push $0xc012a0af
LunaixOS
从零开始
自制操作系统
多进程
为系统调用做好准备
EP 10-5

如何实现系统调用

系统调用的本质就是临时提升优先级，并且在受监管的情况下执行内核例程。

在 x86 架构上，有这三种办法：

1. syscall / sysret → 64 位专享
2. sysenter / sysexit → 32 位可用，但并非全平台可用
3. ✅ 中断

```
_set_idt_intr_entry(LUNAIX_SYS_CALL, 0x08, _asm_isr33, 3);
```

```
// LunaixOS related
#define LUNAIX_SYS_PANIC            32
#define LUNAIX_SYS_CALL             33
```

在 LunaixOS 中， 33 号中断为系统调用

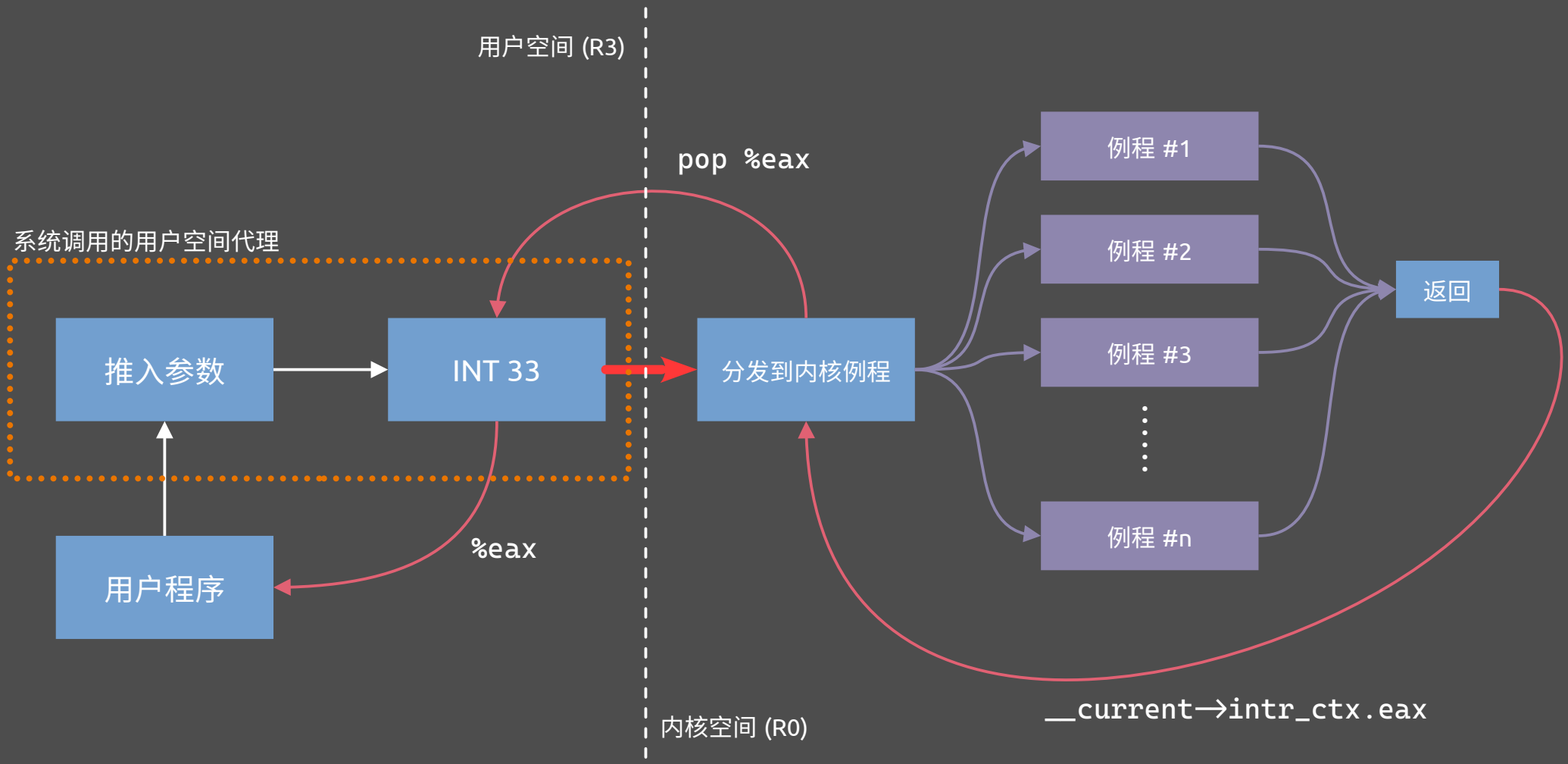
别忘了将中断描述符设置特权级设置为 Ring3 级

如何实现系统调用

系统调用需要做到：

1. 搞明白用户调用的是哪个例程
2. 传入用户给的参数
3. 获取内核例程的返回码，并传回用户

如何实现系统调用


用户空间 (R3)
pop %eax
系统调用的用户空间代理
推入参数
INT 33
分发到内核例程
例程 #1
例程 #2
例程 #3
例程 #n
返回
%eax
用户程序
内核空间 (R0)
__current→intr_ctx.eax

实现系统调用：例程分发

通过调用号来决定（存放于 eax）

调用号 ⇒ 调用表内索引

```
syscall_table:
    1:
    .long 0
    .long __lxsys_fork          /* 1 */
    .long __lxsys_yield
    .long __lxsys_sbrk
    .long __lxsys_brk
    .long __lxsys_getpid        /* 5 */
    .long __lxsys_getppid
    .long __lxsys_sleep
    .long __lxsys_exit
    .long __lxsys_wait
    .long __lxsys_waitpid       /* 10 */
    2:
    .rept __SYSCALL_MAX - (2b - 1b)/4
        .long 0
    .endr
```

```
syscall_hndlr:
    pushl %ebp
    movl 8(%esp), %ebp

    movl  (%ebp), %eax
    cmpl  $__SYSCALL_MAX, %eax
    jae 2f

    shll $2, %eax
    addl $syscall_table, %eax
    cmpl $0, (%eax)
    jne 1f
2:
    neg   %eax
    popl  %ebp
    ret
```

计算表内位移（还包括一些检查）

完成后， %eax 将会包含需要执行的内核例程的地址

系统调用：参数与返回值

在 x86_32 下参数通过栈传入，返回值存入 eax

向系统调用传入参数，有两种办法：

1. 在执行 33 号中断前推入参数

由于栈的切换，我们还是需要从用户栈读取，然后逐一推入内核栈（两次内存操作！）

2. 先把参数存入寄存器，中断，而后逐个推入。 ✅

实现系统调用：参数与返回值

```
struct
{
    reg32 eax;
    reg32 ebx;
    reg32 ecx;
    reg32 edx;
    reg32 edi;
    reg32 ebp;
    reg32 esi;
```

取决于 isr_param 的布局

```
1:
    pushl 24(%ebp)      /* esi - #6 arg */
    pushl 20(%ebp)      /* ebp - #5 arg */
    pushl 16(%ebp)      /* edi - #4 arg */
    pushl 12(%ebp)      /* edx - #3 arg */
    pushl 8(%ebp)       /* ecx - #2 arg */
    pushl 4(%ebp)       /* ebx - #1 arg */

    call (%eax)

    movl %eax, (%ebp)    /* save the return value */

    addl $24, %esp      /* remove the parameters from stack */

    popl %ebp

    ret
```

执行内核例程

系统调用：全部结合在一起……

```
 9  extern void syscall_hndlr(isr_param* param);
10
11  void syscall_install() {
12      intr_subscribe(LUNAIX_SYS_CALL, syscall_hndlr);
13  }
```

将系统调用处理器挂载到 33 号中断上

```
syscall_hndlr:
    pushl %ebp
    movl 8(%esp), %ebp

    movl  (%ebp), %eax          /* eax: cal
    cmpl  $__SYSCALL_MAX, %eax
    jae 2f

    shll $2, %eax
    addl $syscall_table, %eax
    cmpl $0, (%eax)       You, 2 weeks ago
    jne 1f
2:
    neg   %eax
    popl  %ebp
    ret
1:
    pushl 24(%ebp)      /* esi - #6 arg */
    pushl 20(%ebp)      /* ebp - #5 arg */
    pushl 16(%ebp)      /* edi - #4 arg */
    pushl 12(%ebp)      /* edx - #3 arg */
    pushl 8(%ebp)       /* ecx - #2 arg */
    pushl 4(%ebp)       /* ebx - #1 arg */

    call (%eax)

    movl %eax, (%ebp)    /* save the return

    addl $24, %esp      /* remove the param

    popl %ebp

    ret
```

声明一个内核例程：一些例子

```
#define __DEFINE_LXSYSCALL(rettype, name) \
            rettype __lxsys_##name()
```

```
__DEFINE_LXSYSCALL(pid_t, fork)
{
    return dup_proc();
}
```

比如我们之前所讲的 `fork()`

```
__DEFINE_LXSYSCALL3(pid_t, waitpid, pid_t, pid, int*, status, int, options)
{
    return _wait(pid, status, options);
}
```

或者是我们即将所讲的 `waitpid(2)`

导出系统调用给用户程序

```
__LXSYSCALL1(pid_t, wait, int*, status);

__LXSYSCALL3(pid_t, waitpid, pid_t, pid, int*, status, int, options);
```

声明在需要暴露给用户的头文件中

```
static pid_t waitpid(__PARAM_MAP3(pid_t, pid, int*, status, int, options)) {
asm("\n" ::"b"(pid), "c"(status), "d"(options));
__DOINT33(__SYSCALL_waitpid, pid_t) }
```

展开后的宏

将参数塞进对应寄存器

传入调用号到 %eax，执行 int 33，并获取返回码

```
#define __DOINT33(callcode, rettype)
    int v;
    asm volatile("int %1\n" : "=a"(v) : "i"(LUNAIX_SYS_CALL), "a"(callcode));
    return (rettype)v;
```

```
push   %ebp
mov    %esp,%ebp
push   %esi
push   %ebx
mov    %edx,%esi
mov    %ecx,%edx
mov    %eax,%ebx
mov    %esi,%ecx
mov    $0xa,%eax
int    $0x21
pop    %ebx
pop    %esi
pop    %ebp
ret
```

假若内核例程出错了怎么办?

我们需要一个错误代码，并且是全局的（相对于进程）。

```
You, 13 hours ago | 1 author (You)
struct proc_info {
    pid_t pid;
    struct proc_info* parent;
    isr_param intr_ctx;
    struct proc_mm mm;
    void* page_table;
    time_t created;
    uint8_t state;
    int32_t exit_code;
    int32_t k_status;
};
```

可重入的系统调用

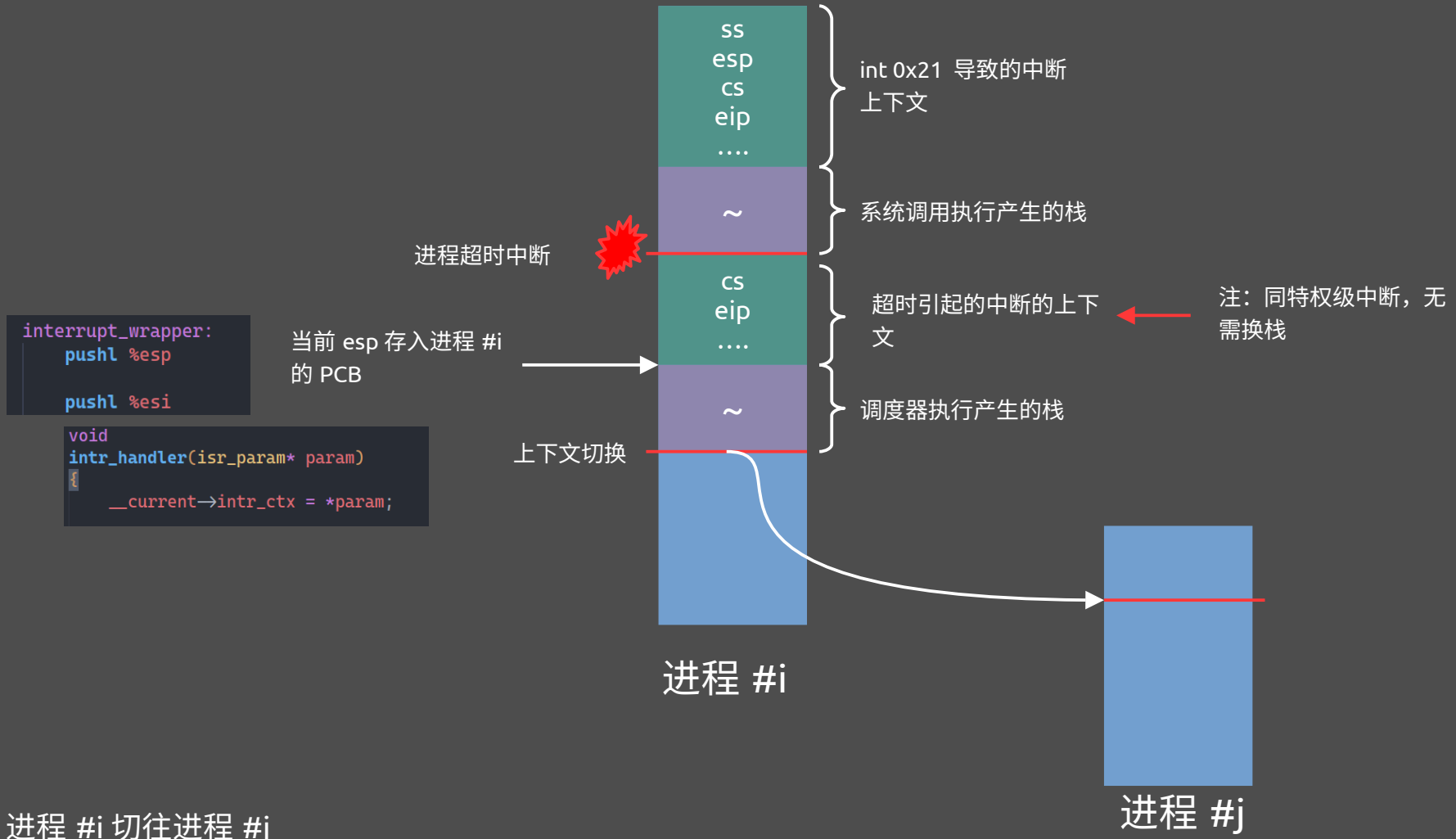
如果进程在执行系统调用时，用完了调度器分配的时间。会发生什么?

1. 什么都不会发生：调度器不会在系统调用执行的过程中进行上下文切换。

2. 调度器执行上下文切换。未完成的系统调用将会在下次调度时完成。（重入系统调用）

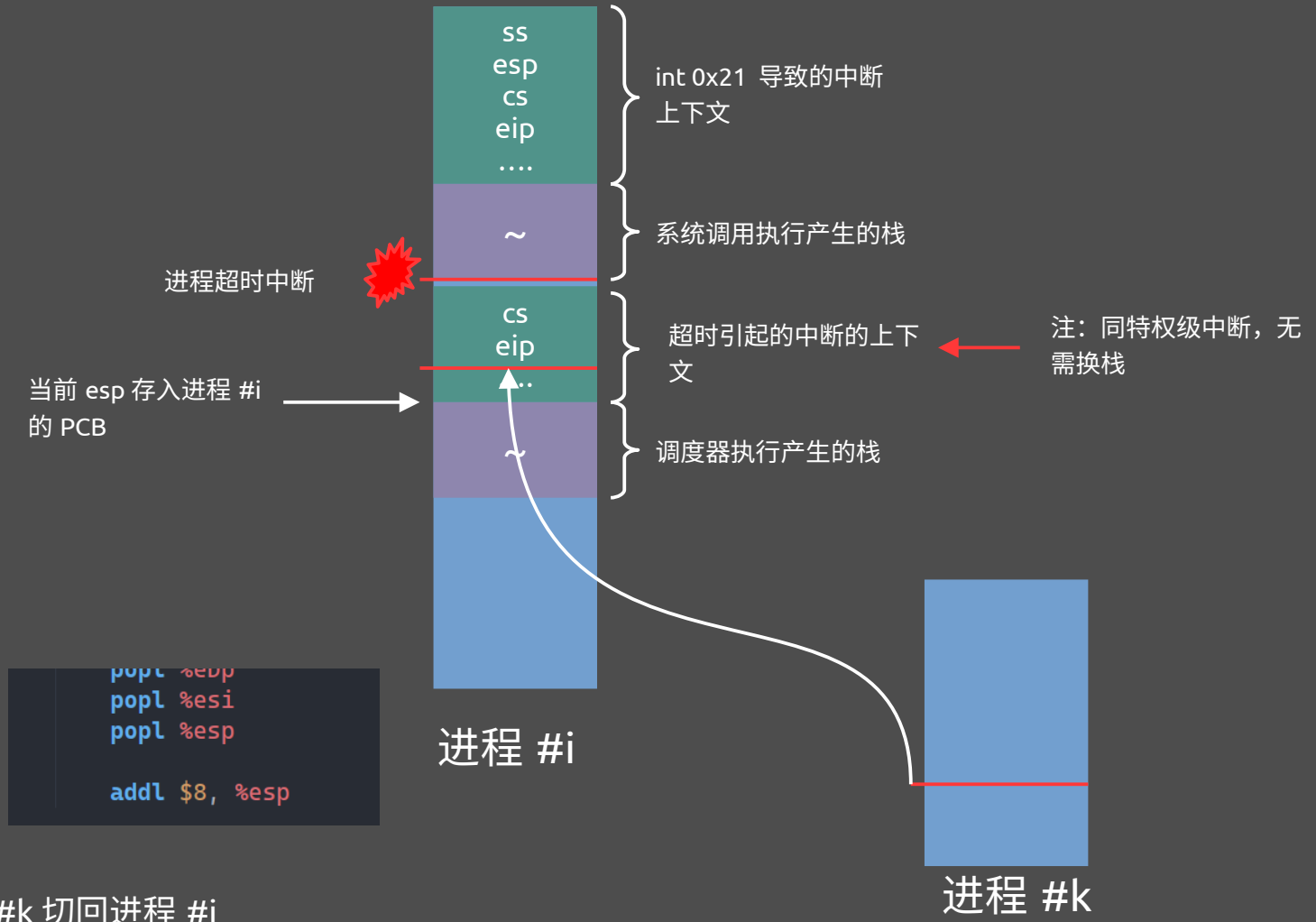
那么 **LunaixOS** 现有设计能够做到这一点吗?

可重入的系统调用



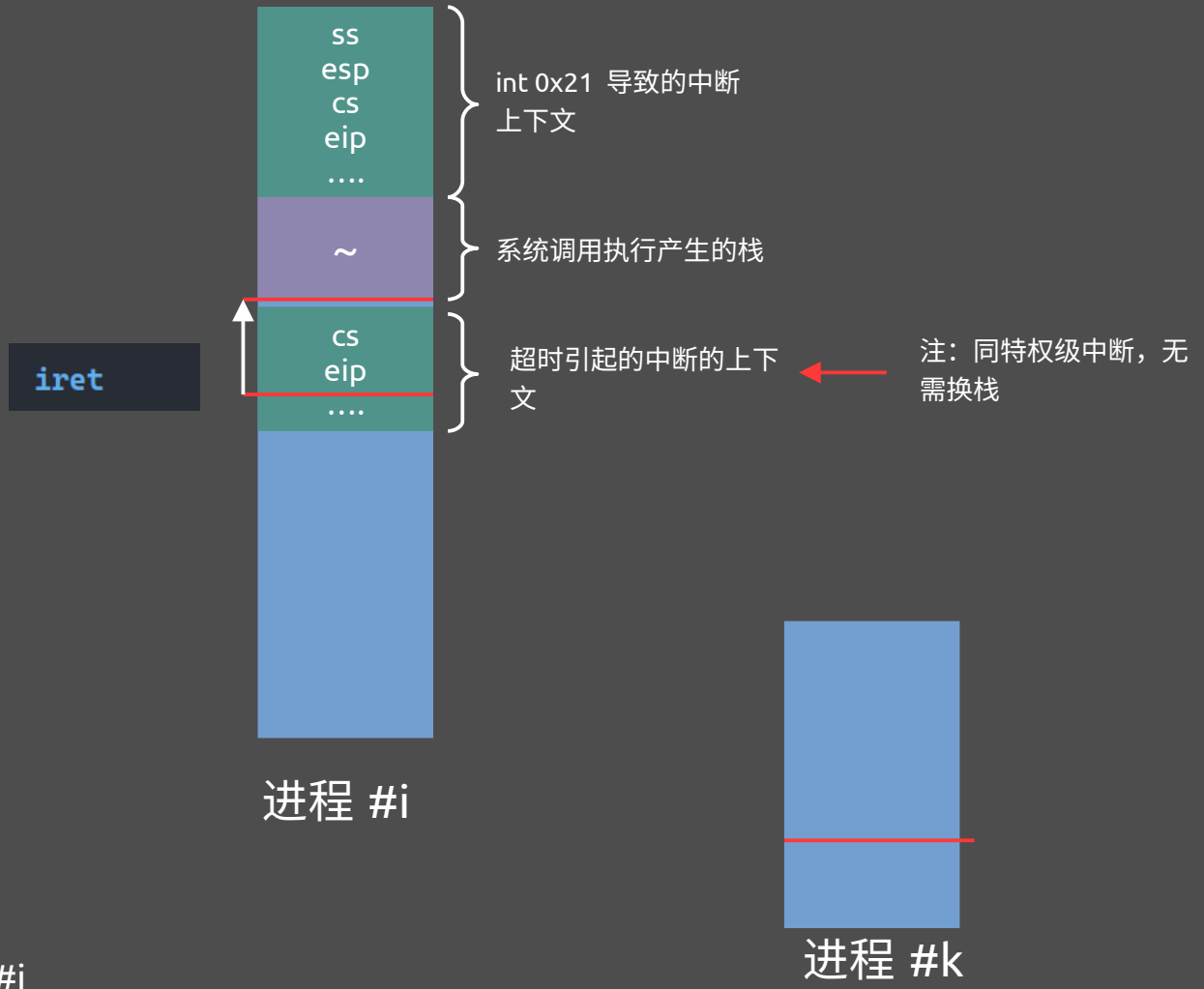
进程 #i 切往进程 #j

可重入的系统调用



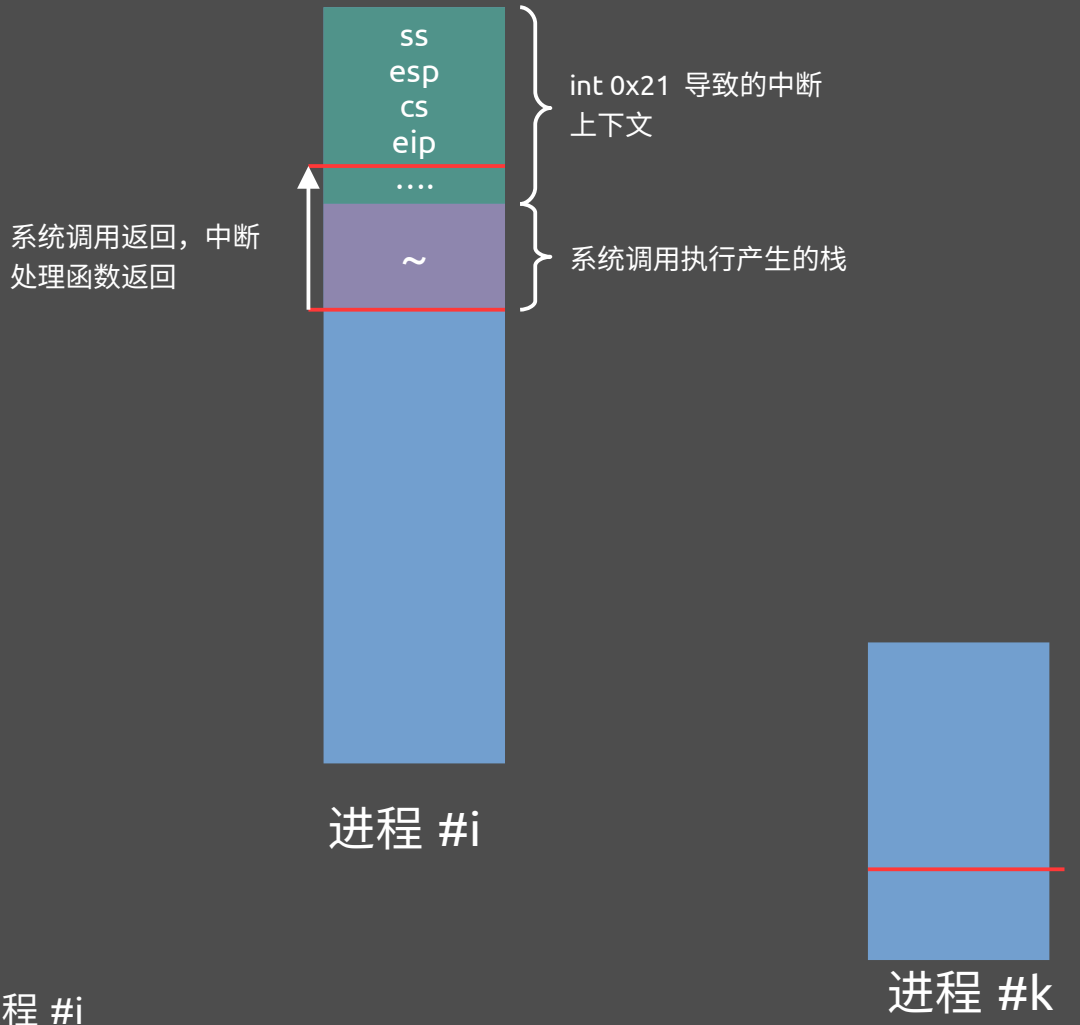
一段时间后，进程 #k 切回进程 #i

可重入的系统调用



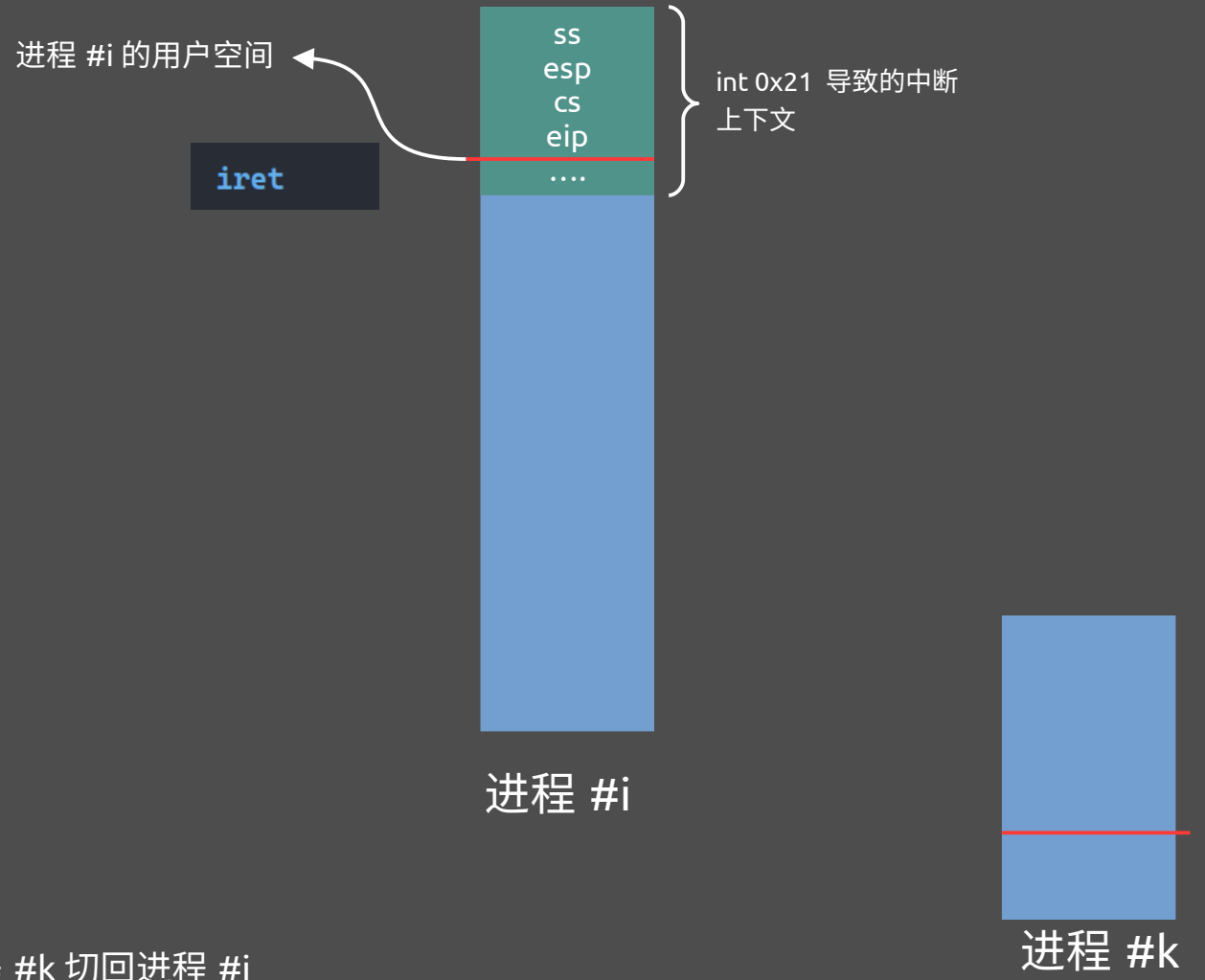
一段时间后，进程 #k 切回进程 #i

可重入的系统调用



一段时间后，进程 #k 切回进程 #i

可重入的系统调用



一段时间后，进程 #k 切回进程 #i

我们需要可重入调用吗?

我们来分析一下优缺点……

好的地方:

1. 在执行系统调用时，依然允许中断的处理
2. 极大简化一些系统调用的实现，比如 wait(2), pause(2) 。
3. 前面看到的：更好的保证进程之间的公平。

不好的地方:

1. 需要非常谨慎的决定在何处划红线。
2. 需要处理同优先级中断的情况。

竞态条件……

我们会在实现信号时看到这一点……

全部都搞成可重入?

```
_set_idt_intr_entry(LUNAIX_SYS_CALL, 0x08, _asm_isr33, 3);
```

v.s.

~~`_set_idt_trap_entry(LUNAIX_SYS_CALL, 0x08, _asm_isr33, 3);`~~

在 LunaixOS 中，为了防止竞态条件的出现，我们还是对系统调用禁用中断，而只在需要时开启。

比如 LunaixOS 的 wait 系统调用的实现

```
    cpu_enable_interrupt();
repeat:
    llist_for_each(proc, n, &__current->children, siblings)
    {
        if (!~wpid || proc->pid == wpid) {
            if (proc->state == PROC_TERMNAT && !options) {
                status_flags |= PROCTERM;
                goto done;
            }
            if (proc->state == PROC_STOPPED && (options & WUNTRACED)) {
                status_flags |= PROCSTOP;
                goto done;
            }
        }
    }
    if ((options & WNOHANG)) {
        return 0;
    }
    // 放弃当前的运行机会
    sched_yield();
    goto repeat;
```

What's Next ?

在这一期中，我们了解到了中断与系统调用的关系，以及实现细节。

也可以发现，基于中断的系统调用非常的低效，因为每一次调用都需要做一次上下文的保存和恢复。

既然我们已经实现了系统调用的框架，那么在下一期中，我们会实现几个 POSIX 中的系统调用。而这也是我们 LunaixOS 兼容 POSIX 的第一步！

```
c010606f: 55                push   %ebp
c0106070: 89 e5             mov    %esp,%ebp
c0106072: 56                push   %esi
c0106073: 53                push   %ebx
c0106074: 83 ec 0c          sub    $0xc,%esp
c0106077: 68 44 a2 12 c0    push   $0xc012a244
c010607c: e8 ed 1c 00 00    call   c0107d6e <printf>
c0106081: be 00 60 10 00    mov    $0x106000,%esi
c0106086: c1 ee 0c          shr    $0xc,%esi
c0106089: 83 c4 08          add    $0x8,%esp
c010608c: 56                push   %esi
c010608d: 68 6c a2 12 c0    push   $0xc012a26c
c0106092: e8 d7 1c 00 00    call   c0107d6e <printf>
c0106097: 83 c4 10          add    $0x10,%esp
c010609a: bb 00 00 00 00    mov    $0x0,%ebx
c010609f: eb 14             jmp    c01060b5 <_kernel_post_init+0x46>
c01060a1: 89 d8             mov    %ebx,%eax
c01060a3: c1 e0 0c          shl    $0xc,%eax
c01060a6: 83 ec 0c          sub    $0xc,%esp
c01060a9: 50                push   %eax
c01060aa: e8 9f 0a 00 00    call   c0106b4e <vmm_unmap_page>
c01060af: 83 c3 01          add    $0x1,%ebx
c01060b2: 83 c4 10          add    $0x10,%esp
c01060b5: 39 f3             cmp    %esi,%ebx
c01060b7: 72 e8             jb     c01060a1 <_kernel_post_init+0x32>
c01060b9: e8 54 ...         call   <kalloc_init>
c01060be: 85 c0 ...
c01060c0: 74 07 ...                <_kernel_post_init+0x5a>
c01060c2: 8d 65 f8 ...             0x-8(%ebp),%esp
c01060c5: 5b                pop    %ebx
c01060c6: 5e                pop    %esi
c01060c7: 5d                pop    %ebp
c01060c8: c3                ret
c01060c9: 83 ec 04          sub    $0x4,%esp
c01060cc: 6a 40             push   $0x40
c01060ce: 68 af a0 12 c0    push   $0xc012a0af
```

LunaixOS

从零开始

自制操作系统

多进程

一些 Unix 系统调用的具体实现

EP 10-6

本期简介

1. 讨论 LunaixOS 当前支持的系统调用，以及其实现细节。

2. 兼容性测试：我们将会用这些系统调用，写一个简单的程序，并对比和分析 LunaixOS 和 Linux 的运行结果

LunaixOS 的系统调用

截止到多进程的实现， LunaixOS 支持的 POSIX 兼容的系统调用：

getpid	sbrk	wait	yield
getppid	brk	waitpid	
getpgid	fork		
setpgid	sleep		
_exit			

<lunaix/lunistd.h>

以及一些非 POSIX 的：

我们主要把注意力放在
wait(2)， sleep(3)， pgid 和 _exit(2)

_exit(2) 的实现

```
__DEFINE_LXSYSCALL1(void, exit, int, status)
{
    terminate_proc(status);
}
```

```
void
terminate_proc(int exit_code)
{
    __current->state = PROC_TERMNAT;
    __current->exit_code = exit_code;

    schedule();
}
```

sleep(3) 的实现

功能：让进程休眠指定的秒数，返回秒数。如果在休眠未完成时被再次调用（如在信号送达时），则返回剩余秒数。

大致思路：

1. 每次 sleep 调用为进程绑定一个计时器，并将进程状态设置为阻塞。

2. 当计时器结束时，在回调中将进程状态改回为已暂停。

```
You, 13 hours ago | 1 author (You)
struct proc_info {
    pid_t pid;
    struct proc_info* parent;
    isr_param intr_ctx;
    struct proc_mm mm;
    void* page_table;
    time_t created;
    uint8_t state;
    int32_t exit_code;
    int32_t k_status;
    struct lx_timer* timer;
};
```

sleep(3) 的实现

```
__DEFINE_LXSYSCALL1(unsigned int, sleep, unsigned int, seconds)
{
    if (!seconds) {
        return 0;
    }

    if (__current->timer) {
        return __current->timer->counter / timer_context()->running_frequency;
    }

    struct lx_timer* timer =
      timer_run_second(seconds, proc_timer_callback, __current, 0);
    __current->timer = timer;
    __current->intr_ctx.registers.eax = seconds;
    __current->state = PROC_BLOCKED;
    schedule();
}
```

```
static void
proc_timer_callback(struct proc_info* proc)
{
    proc->timer = NULL;
    proc->state = PROC_STOPPED;
}
```

sleep(3)：另一种思路?

计时器方法非常低效，每次 APIC 产生 tick 时都需要检查计时器状态（ 2048Hz ，大约每 488 **微秒**一次）。

回调函数只是更改状态 ⇒ 在最好的情况下，进程的只会在下一次调度时才算真正唤醒（至少 300 **毫秒**的延迟）。

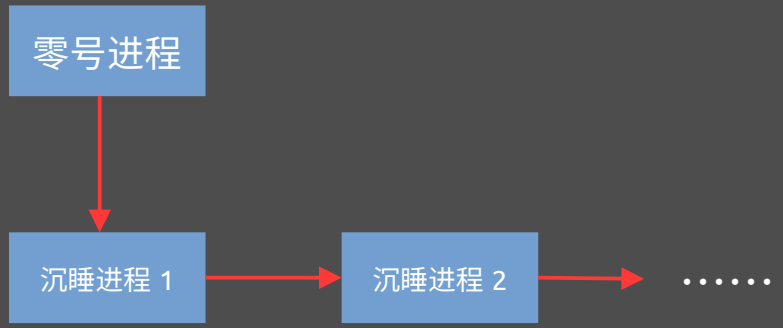
非常的划不来!

改进：不使用 LunaixOS 的计时器序列，而在 PCB 里面做一个简化版的计数器。其检查间隔为调度间隔。

sleep(3)：另一种思路

```
You, 18 minutes ago | 1 author (You)
struct
{
    struct llist_header sleepers;
    time_t wakeup_time;
    [redacted]
} sleep;
```

在 PCB 里面




```
78  void
79  check_sleepers()
80  {
81      struct proc_info* leader = &sched_ctx._procs[0];
82      struct proc_info *pos, *n;
83      time_t now = clock_systime();
84      llist_for_each(pos, n, &leader->sleep.sleepers, sleep.sleepers)
85      {
86          if (PROC_TERMINATED(pos->state)) {
87              goto del;
88          }
89
90          time_t wtime = pos->sleep.wakeup_time;
91          [redacted]
92
93          if (wtime && now ≥ wtime) {
94              pos->sleep.wakeup_time = 0;
95              pos->state = PS_STOPPED;
96          }
97
98  ›       [redacted]
102
103         if (!wtime [redacted]) {
104         del:
105             llist_delete(&pos->sleep.sleepers);
106         }
107     }
108 }
```

每次调度前检查是否有需要唤醒的进程

```
__DEFINE_LXSYSCALL1(unsigned int, sleep, unsigned int, seconds)
{
    if (!seconds) {
        return 0;
    }

    if (__current->sleep.wakeup_time) {
        return (__current->sleep.wakeup_time - clock_systime()) / 1000U;
    }

    __current->sleep.wakeup_time = clock_systime() + seconds * 1000;
    llist_append(&sched_ctx._procs[0].sleep.sleepers,
                 &__current->sleep.sleepers);

    __current->intr_ctx.registers.eax = seconds;
    __current->state = PS_BLOCKED;
    schedule();
}
```

sleep(3) 系统调用的新实现

进程组和他的系统调用

进程们可以形成一个组，从而实现统一管理：

1. waitpid 可以等待进程组中的某个子进程返回。
2. 递送到进程组的信号，会同时递送给其中的每个进程。

相关系统调用：

`setpgid(2)`

`getpgid(2)`

进程组和他的系统调用

```c
struct proc_info
{
    pid_t pid;
    struct proc_info* parent;
    isr_param intr_ctx;
    struct llist_header siblings;
    struct llist_header children;
    struct llist_header grp_member;
    struct proc_mm mm;
    void* page_table;
    time_t created;
    uint8_t state;
    int32_t exit_code;
    int32_t k_status;
    pid_t pgid;
    struct lx_timer* timer;
};
```

pgid 标识了这个进程所属的组别，
为组长的 pid

```c
__DEFINE_LXSYSCALL2(int, setpgid, pid_t, pid, pid_t, pgid)
{
    struct proc_info* proc = pid ? get_process(pid) : __current;

    if (!proc) {
        __current->k_status = LXINVL;
        return -1;
    }

    pgid = pgid ? pgid : proc->pid;

    struct proc_info* gruppenfuhrer = get_process(pgid);

    if (!gruppenfuhrer || proc->pgid == proc->pid) {
        __current->k_status = LXINVL;
        return -1;
    }

    llist_delete(&proc->grp_member);
    llist_append(&gruppenfuhrer->grp_member, &proc->grp_member);

    proc->pgid = pgid;
    return 0;
}
```

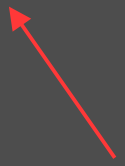
setpgid 不能改变
组长的 pgid

wait(2) 的实现

wait： 让调用进程等待**任意**一个子进程退出。
waitpid： 让调用进程等待**指定**的子进程退出

实现思路：

1. 遍历进程表检查所有符合条件的进程（父进程 pid= 当前 pid）的状态。
2. 如果存在已退出子进程，则返回。
3. 否则回到第 1 步继续检查。

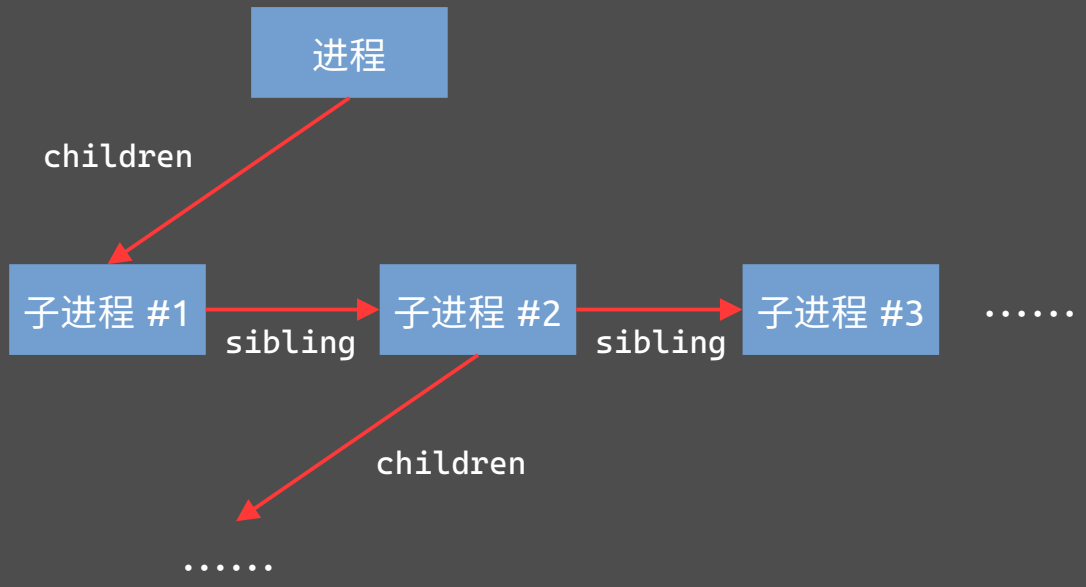


需要系统调用可重入可中断。否则调度器无法进行调度，而只能卡在 wait 这里。

⇒ 系统死锁！

还有更好的办法吗?

通过实际构建一个进程树来缩小检查范围。




```
struct proc_info
{
    pid_t pid;
    struct proc_info* parent;
    isr_param intr_ctx;
    struct llist_header siblings;
    struct llist_header children;
    struct llist_header grp_member;
    struct proc_mm mm;
    void* page_table;
    time_t created;
    uint8_t state;
    int32_t exit_code;
    int32_t k_status;
    pid_t pgid;
    struct lx_timer* timer;
};
```

wait(2) 的实现

```
    wpid = wpid ? wpid : -__current->pgid;
    cpu_enable_interrupt();
repeat:
    llist_for_each(proc, n, &__current->children, siblings)
    {
        if (!~wpid || proc->pid == wpid || proc->pgid == -wpid) {
            if (proc->state == PROC_TERMNAT && !options) {
                status_flags |= PROCTERM;
                goto done;
            }
            if (proc->state == PROC_STOPPED && (options & WUNTRACED)) {
                status_flags |= PROCSTOP;
                goto done;
            }
        }
    }
    if ((options & WNOHANG)) {
        return 0;
    }
    // 放弃当前的运行机会
    sched_yield();
    goto repeat;
```

问题：为什么我们不直接调用 schedule()？

我们直接放弃当前的时间切片，迫使调度器在下次时钟中断产生时，进行调度，以最大化执行效率

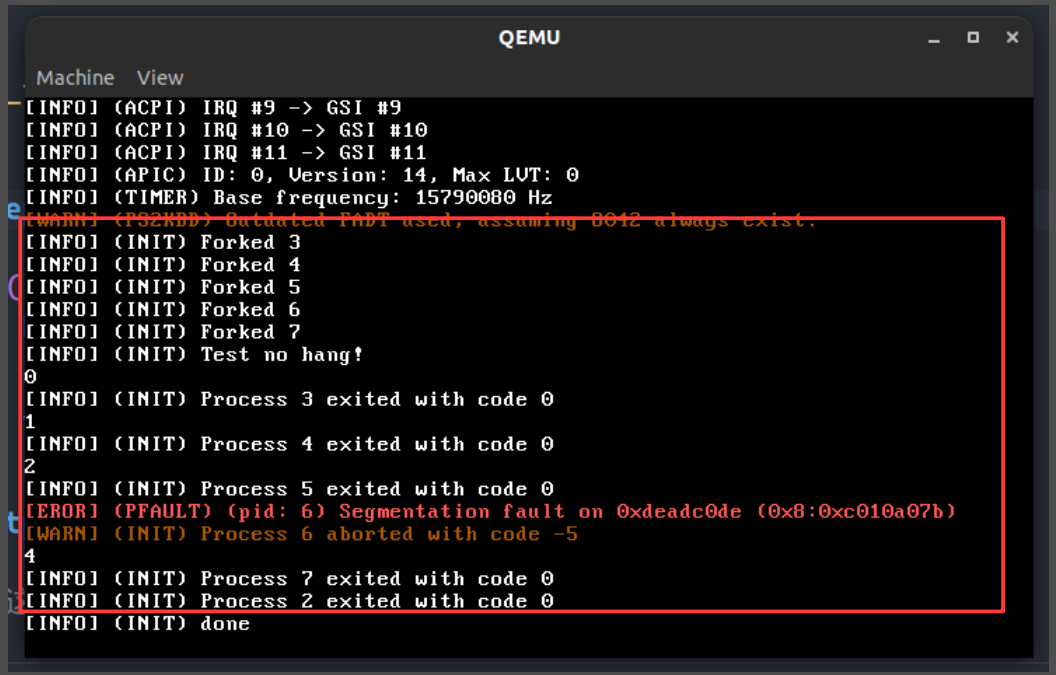
兼容性测试

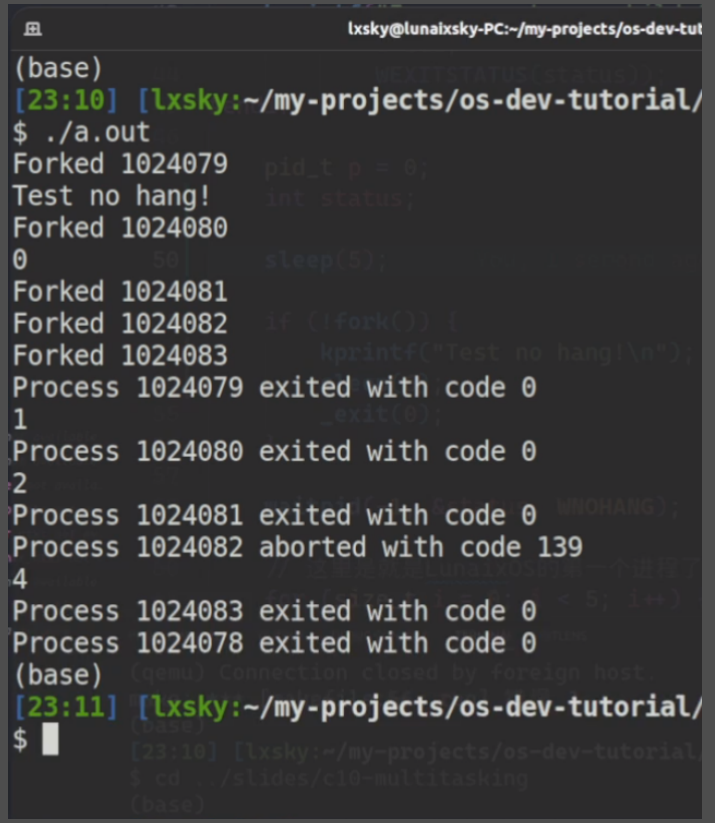

```c
pid_t p = 0;

if (!fork()) {
    kprintf("Test no hang!\n");
    sleep(6);
    _exit(0);
}

waitpid(-1, &status, WNOHANG);

for (size_t i = 0; i < 5; i++) {
    pid_t pid = 0;
    if (!(pid = fork())) {
        sleep(i);
        if (i == 3) {
            i = *(int*)0xdeadc0de; // seg fault!
        }
        tty_put_char('0' + i);
        tty_put_char('\n');
        _exit(0);
    }
    kprintf(KINFO "Forked %d\n", pid);
}

while ((p = wait(&status)) ≥ 0) {
    short code = WEXITSTATUS(status);
    if (WIFEXITED(status)) {
        kprintf(KINFO "Process %d exited with code %d\n", p, code);
    } else {
        kprintf(KWARN "Process %d aborted with code %d\n", p, code);
    }
}
```

V.S.


```c
pid_t p = 0;
int status;

if (!fork()) {
    printf("Test no hang!\n");
    sleep(6);
    _exit(0);
}

waitpid(-1, &status, WNOHANG);

pid_t pid = 0;
for (size_t i = 0; i < 5; i++) {
    if (!(pid = fork())) {
        sleep(i);
        if (i == 3) {
            i = *(int*)0xdeadc0de; // seg fault!
        }
        printf("%d", i);
        printf("\n");
        _exit(0);
    }
    printf("Forked %d\n", pid);
}

while ((p = wait(&status)) ≥ 0) {
    short code = WEXITSTATUS(status);
    if (WIFEXITED(status)) {
        printf("Process %d exited with code %d\n", p, status);
    } else {
        printf("Process %d aborted with code %d\n", p, status);
    }
}
```

兼容性测试：结果分析





观察得出，四点不一样：

- 进程 pid
- 打印顺序（调度顺序）
- 错误返回码
- 额外的报错信息

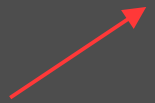
结论：所有系统调用工作正常并且有着一致行为！

多进程之后

我们操作系统的旅程其实才刚刚开始……

更多激动人心的内容即将到来！

1. 实现信号机制（Signal）
2. 实现多线程？【暂定】
3. PCI 总线与 SATA 驱动（via AHCI）
4. 文件系统
5. 程序加载与执行



序号不是顺序！