Lab4 introduction

2009-04-15

Lab4时间安排

- **Lab4时间：4月15日至5月5日**
- 第一周
  - **PartA** 实现调度算法及创建新的**environment**
- 第二周
  - **PartB** 实现用户态**page fault** 处理和**copy-on-write fork**函数
- 第三周
  - **PartC** 实现可抢占调度及**IPC**
  - 代码调试和文档

Lab4任务清单

- Round-robin调度算法
- 实现创建environment所需的系统调用
- 实现用户态的page fault处理函数
- 实现Copy_on_Write的fork函数
- 在时钟中断中调用调度函数，实现可抢占的调度
- 实现进程间消息通信(IPC)

Lab4实习要求

- **Exercise1~11**
  - 必做
- **Questions**
  - 必做，写入文档
- **Challenges**
  - 选做

Lab4代码树： kern

kern

- init.c：注意新添加的代码
- pmap.c*: 内存管理
- syscall.c***：系统调用
- trap.c****：与处理trap相关的各种函数
- picirq.c*：启动硬件时钟中断
- sched.c：round-robin进程调度

Lab4代码树： inc

- inc
  - mmu.h*： 注意VPN、VPD宏
  - memlayout.h**： 虚拟内存分布图
  - env.h*： struct Env新增字段
  -
  -

Lab4代码树： lib

- lib
  - entry.S*：注意vpt和vpd的设置
  - pfentry.S***：用户态page fault处理入口
  - pgfault.c*：设置page fault handler
  - fork.c****：实现cow进程创建
  - ipc.c****：实现进程间通信

Lab4代码树： user

- user
  - dumbfork.c*：一个古老的fork，建议阅读
  - faultalloc.c*：对理解用户态page fualt处理极有帮助
  - forktree.c：演示进程创建
  -
  -

Lab4数据结构

```
struct Env {
    struct Trapframe env_tf;    // Saved registers
    LIST_ENTRY(Env) env_link;   // Free list link pointers
    envid_t env_id;           // Unique environment identifier
    envid_t env_parent_id;      // env_id of this env's parent
    unsigned env_status;        // Status of the environment
    uint32_t env_runs;        // Number of times environment has run

    // Address space
    pde_t *env_pgdir;         // Kernel virtual address of page dir
    physaddr_t env_cr3;       // Physi                age dir

    // Exception handling
    void *env_pgfault_upcall;   // page fault upcall entry point

    // Lab 4 IPC
    bool env_ipc_recving;       //              eceiving
    void *env_ipc_dstva;        // va at which to map received page
    uint32_t env_ipc_value;     // data value sent to us
    envid_t env_ipc_from;       // envid of the sender
    int env_ipc_perm;         // perm of page mapping received
};
```

用户态page fault处理入口

进程间通信

Lab4数据结构

- **trap-time information on user exception stack**

```
struct UTrapframe {
    /* information about the fault */
    uint32_t utf_fault_va;  /* va for T_PGF
    uint32_t utf_err;
    /* trap-time return state */
    struct PushRegs utf_regs;
    uintptr_t utf_eip;
    uint32_t utf_eflags;
    /* the trap-time stack to return to */
    uintptr_t utf_esp;
};
```

Part A

- **Round-Robin Scheduling**
- **System Calls for Environment Creation**

Part A-1: Round-Robin Scheduling

- 进程调度
  - 要求在**kern/sched.c**里的**sched_yield()**函数中实现**round-robin**进程调度
- 关键点
  - **curenv**全局变量
    - 在运行第一个用户进程之前，**curenv**的值为**NULL**
  - 调度过程

调度过程

- **envs[1]、envs[2]… envs[NENV-1]**形成一个逻辑上的环状结构
- 如果**curenv==NULL**，就从**envs[1]**开始检查；否则就从**curenv**指向的**Env**结构的下一个**envs**数组元素开始检查。从环上寻找第一个状态为**ENV_RUNNABLE**的进程
- 如果找到，则让该进程占用**cpu**
- 否则，就运行**envs[0]**对应的进程**(**即**idle**进程**)**

调度实例

- 实例1：只有envs[0], envs[1], envs[2]的状态是ENV_RUNNABLE，且curenv指向envs[2]
  - sched_yield将选择envs[1]对应进程占用cpu
- 实例2：只有envs[0]，envs[2]的状态是ENV_RUNNABLE，且curenv指向envs[2]
  - Sched_yield将选择envs[2]对应进程占用cpu
- 注意：envs[0]只有在不存在其他ENV_RUNNABLE状态的进程时，才会占用cpu

PartA-2: System Calls for Environment Creation

- 需要实现的系统调用
  - **sys_exofork**
  - **sys_env_set_status**
  - **sys_page_alloc**
  - **sys_page_map**
  - **sys_page_unmap**
- 注意
  - 以上系统调用大部分是对**pmap.c**中函数进行封装
  - 主要工作是进行权限检查

Part B

- User-level page fault handling

- Implementing Copy-on-Write Fork

Part B-1: User-level page fault handling

- JOS中的以下情况会触发14号中断(Page-Fault)
  - 页目录项或页表项的P(present)标志位没有设置，即要访问的页面不存在
  - 在用户态试图访问只有内核才能访问的页（即：PTE_U没有设置）
  - 试图向一个只读页中写入数据

发生 page fault时系统运行状态

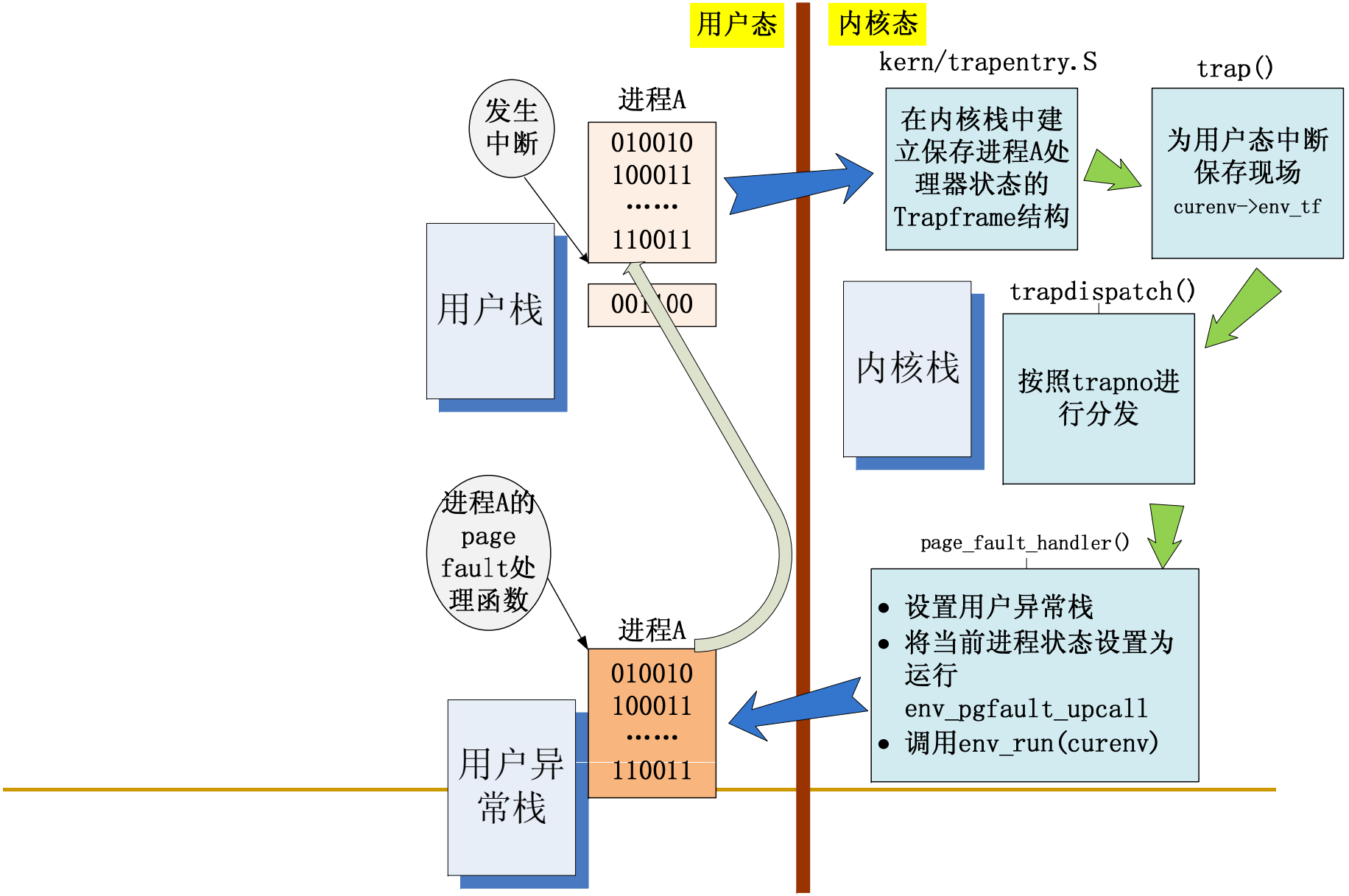
- 用户态
  - 用户进程正常运行时发生**page fault**
  - 用户进程在进行**page fault**处理时又发生**page fault**
- 内核态
  - （**Lab 3**中已经处理）

- 注意：
  - 讲义中讨论的**page fault**处理过程是针对“用户进程正常运行时发生**page fault**”的情况

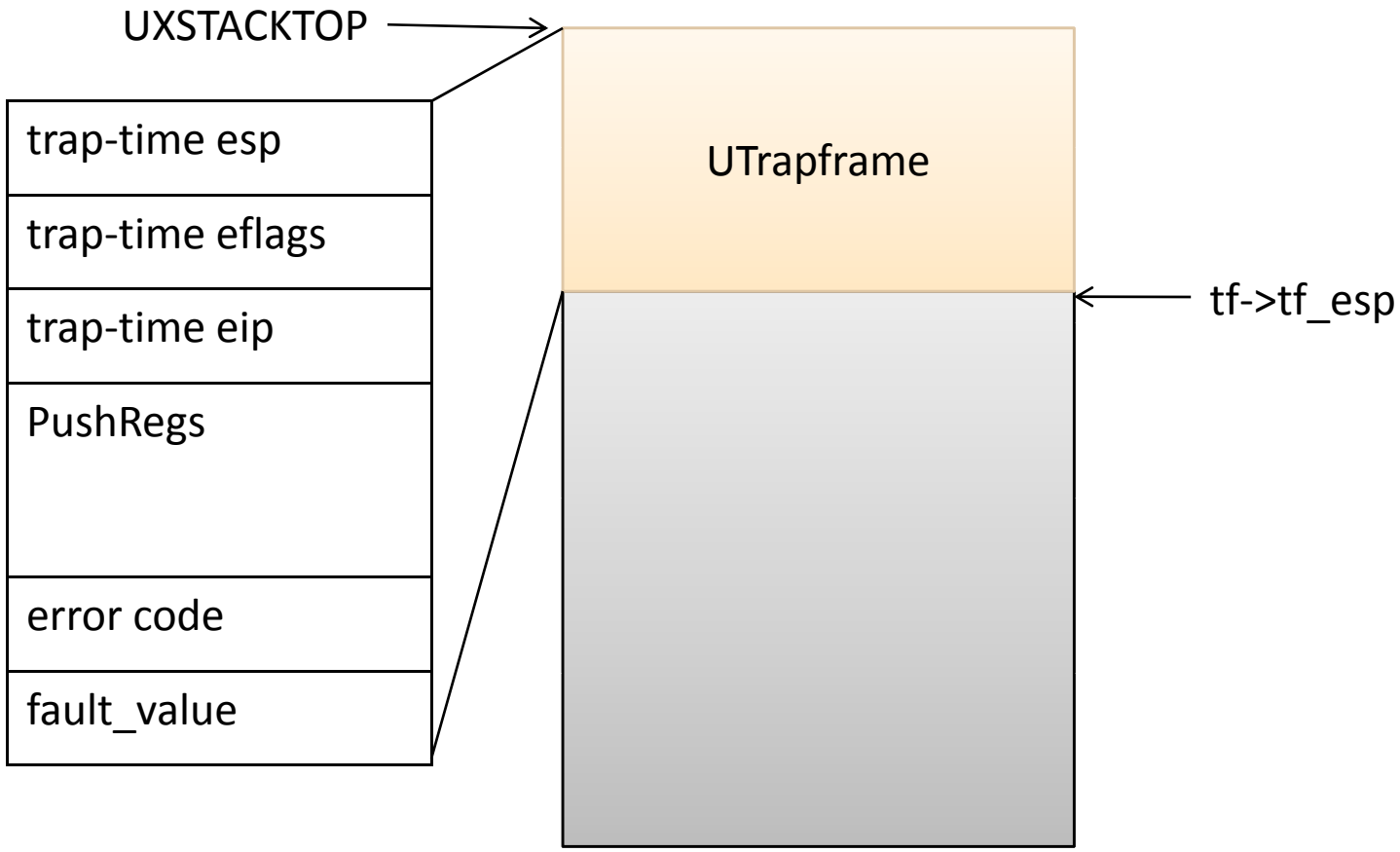
JOS中 page fault 的处理

- 与JOS中其他中断处理程序不同，
  用户态 page fault 由用户进程设置的用户态处理函数(handler)进行处理
- 用户态处理函数运行时使用用户异常栈

JOS中page fault处理过程



用户异常栈设置


UXSTACKTOP
trap-time esp
trap-time eflags
trap-time eip
PushRegs
error code
fault_value
UTrapframe
tf->tf_esp

用户态page fault处理入口点

- 用户态page fault处理的入口点(entry point)
  - struct Env中的env_pgfault_upcall字段
- 工作流程：
  - 调用用户进程设定的page fault处理函数
  - 返回到发生page fault之前的状态继续执行
    - 根据栈顶的UTrapframe将寄存器的值设置为发生中断时的值
    - 在用户栈中写入发生page fault时的eip值
    - 切换堆栈到用户栈
    - ret

用户态page fault处理函数设置

- **sys_env_set_pgfault_upcall(envid, func)**
  - 设置进程的**env_pgfault_upcall**字段
- **进程通过lib/pgfault.c中的set_pgfault_handler()函数来设置用户态page fault处理函数；同时将env_pgfault_upcall设置为lib/pfentry.S中的_pgfault_upcall**

区分

- 内核栈中的**Trapframe**
  - 发生中断时由硬件压入一部分数据
  - **trapentry.S**中设置另一部分数据
  - 作用：保存发生中断时的进程状态
- **kern**中**page_fault_handler()**中的参数**tf**
  - **tf = &curenv->env_tf;**
  - 修改**tf**就相当于修改当前进程的运行状态
  - 思考：是何时设置的
- 用户异常栈中的**Utrapframe**
  - 保存从异常处理函数返回到进程正常运行所需的信息

Part B-2: Implementing Copy-on-Write Fork

- **dumbfork**
  - 一个古老的**fork**，采用复制地址空间的内容的方法创建子进程
  - **user/dumbfork.c**
- **fork**
  - 实现了**copy-on-write**的**fork**函数
  - **lib/fork.c**

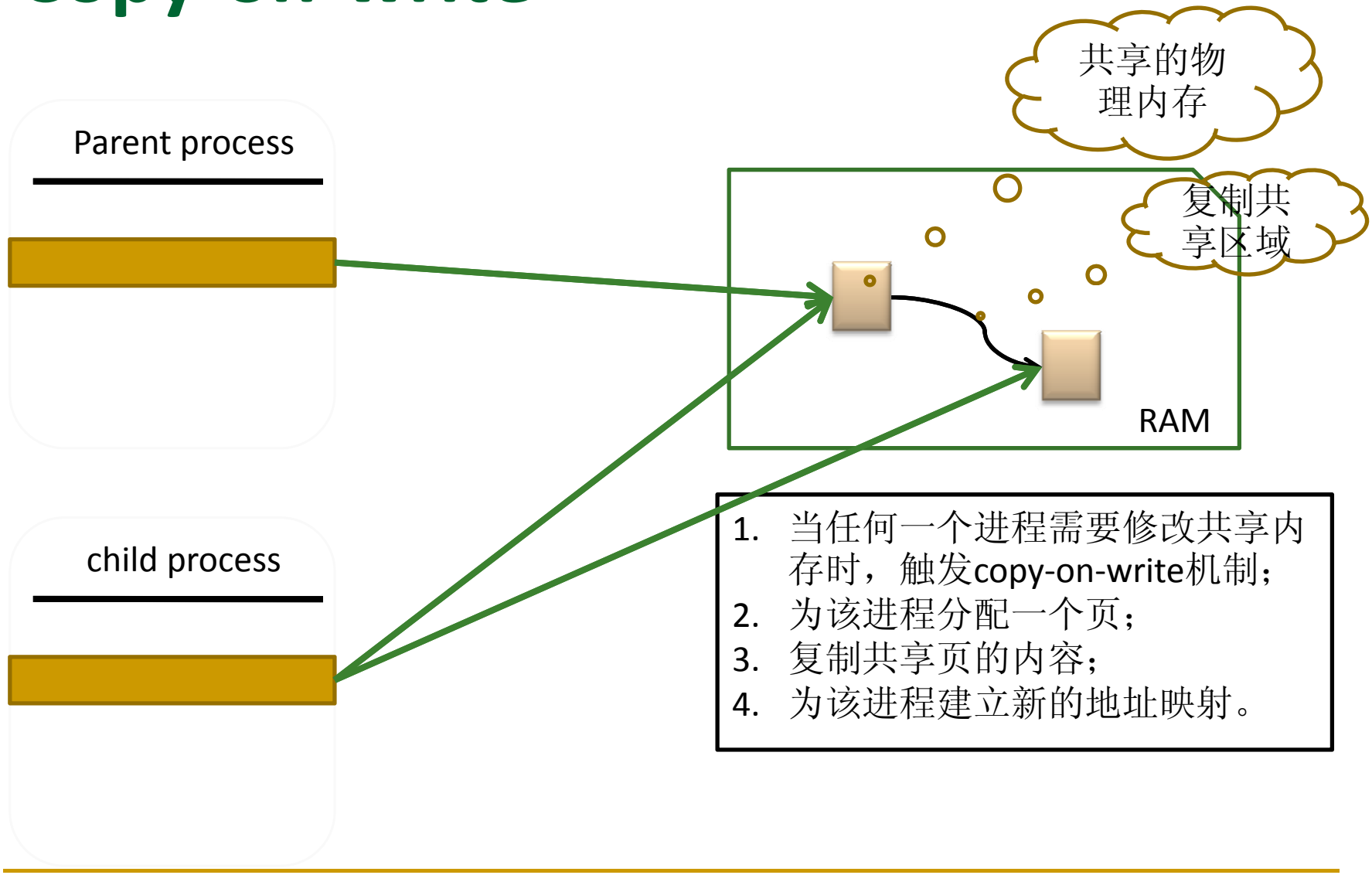
dumbfork流程

1. sys_exofork
   - env_alloc
   - eax (0)
   - Status (ENV_NOT_RUNNABLE)
2. 使用duppage为子进程分配内存，并把父进程地址空间的内容复制到子进程内存
3. 复制正在使用的栈到子进程
4. 标志子进程为ENV_RUNNABLE

Copy-on-Write（写时复制）

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory
- If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied

Copy-on-write



fork()的原理

- 利用一些系统调用来实现一个用户空间的、写时复制的库函数**fork**
- 用户创建进程相关的系统调用:
  - **sys_exofork**
  - **sys_env_set_status**
  - **sys_page_alloc**
  - **sys_page_map**
  - **sys_page_unmap**

fork()的流程

1. set_pgfault_handler
2. sys_exofork
   - env_alloc
   - eax (0)
   - Status (ENV_NOT_RUNNABLE)
3. 把writable或copy-on-write的页映射为copy-on-write (使用duppage函数)
4. 为子进程分配exception stack
   思考：为什么异常栈不能用COW？
5. 为子进程设置user-level page_fault_handler
6. 标志子进程为ENV_RUNNABLE

vpt & vpd—the great purpose

- lib/entry.S

```
.globl vpt
.set vpt, UVPT
.globl vpd
.set vpd, (UVPT+(UVPT>>12)*4)
```

- vpd[PDX(addr)]
  - 获得addr对应的页目录项
- vpt[VPN(addr)]
  - 获得addr对应的页表项

vpt[VPN(addr)]

1. 这个地址转换为

   VPT + PDX(addr)|PTX(addr)|00

2. **VPT的低22位为0，所以这个32位的地址实际是**

PDX(VPT)	PDX(addr)	PTX(addr)	00
31------22	21------12	11------2	1-0

3. 使用**PDX(VPT)**在页目录在查找，得到一个特殊的页表**(SPT)——**页目录本身 **(**自映射的作用**)**
4. 使用**PDX(addr)**在**SPT(**也就是页目录**)**里找到一个特殊的页**(SPage)——**实际上是页表
5. 使用**PTX(addr)|00**得到页表项

fork() vs dumb_fork()

- **dumb_fork**
  - 父进程复制整个地址空间的内容到子进程
  - 创建新进程的时间大部分用在复制
- **fork**
  - 父进程只复制映射到子进程
  - 当某一个进程需要修改共享页面时，才分配页并复制该页内容
- **fork优点**
  - 创建新进程的效率提高

Part C

- Clock Interrupts and Preemption
- Inter-Process communication(IPC)

Part C: Clock Interrupts and Preemption

定时器中断有什么用？

- 到目前为止，如果一个正在运行的用户进程不主动放弃**cpu**，它将永远占用**cpu**
- 定时器中断**(Timer interrupt)**
  - 周期性产生
  - 可以通过将**EFLAGS**寄存器的**IF**位清**0**来屏蔽
  - 内核可以在定时器中断的中断处理程序中选择其他进程占用**cpu**

JOS中的定时器中断的屏蔽问题

- 用户态下，要保证定时器中断处于开启的状态
  - 将进程**Env**结构中存储的**eflags**值的**IF**位设置成**1**
  - 当调用**env_run**运行一个用户进程时，该进程**Env**结构中的**eflags**值会在执行**iret**指令时，被**cpu**加载到**EFLAGS**寄存器中，从而保证在用户态开启中断
- 内核态下，要屏蔽外部中断
  - 使用**cli (clear IF)**指令

Part C: Inter-Process communication (IPC)

- 每个进程都有自己独立的虚拟地址空间
  - 好处：为每个进程都呈现一种独占**cpu**的“假象”
  - 坏处：进程之间传递消息不方便

- 进程间通信
  - 实现了进程之间传递消息的机制

JOS中的IPC

- 发送方将传送一个32位值给接收方
- 发送方可能给接收方“传送”一个页面
  - 条件：发送方指定要“发送”一个页面，且接收方指定要“接收”一个页面
  - 实现：让接收方共享发送方“发送”的页面，即为接收方建立到对应物理页面的映射

Env结构中IPC相关的域(1/2)

- **Env_ipc_recving**
  - 指定该进程是否处于等待其他进程向它通过ipc传送消息的状态
- **Env_ipc_dstva**
  - 指定要被该进程“接收”的页面被映射到的虚拟地址
- **Env_ipc_value**
  - 存储接收到的32位值

Env结构中IPC相关的域(2/2)

- **Env_ipc_from**
  - 指定这次IPC的发送者的Env id
- **Env_ipc_perm**
  - 当页面传送发生时，这个域存发送者许可给接收者的对被传送的页面的权限

IPC相关的系统调用

- **Sys_ipc_recv**
  - 接收者调用
- **Sys_ipc_try_send**
  - 发送者调用

Sys_ipc_recv

- 将当前进程的**env_ipc_recving**设置成**1**
- 设置**env_ipc_dstva**
- 设置当前进程状态为**ENV_NOT_RUNNABLE**
- 选择其他进程占用处理器

Sys_ipc_try_send的基本处理流程