MOS操作系统实验

lab4 系统调用和fork

姜博 gongbell@gmail.com

内容提要

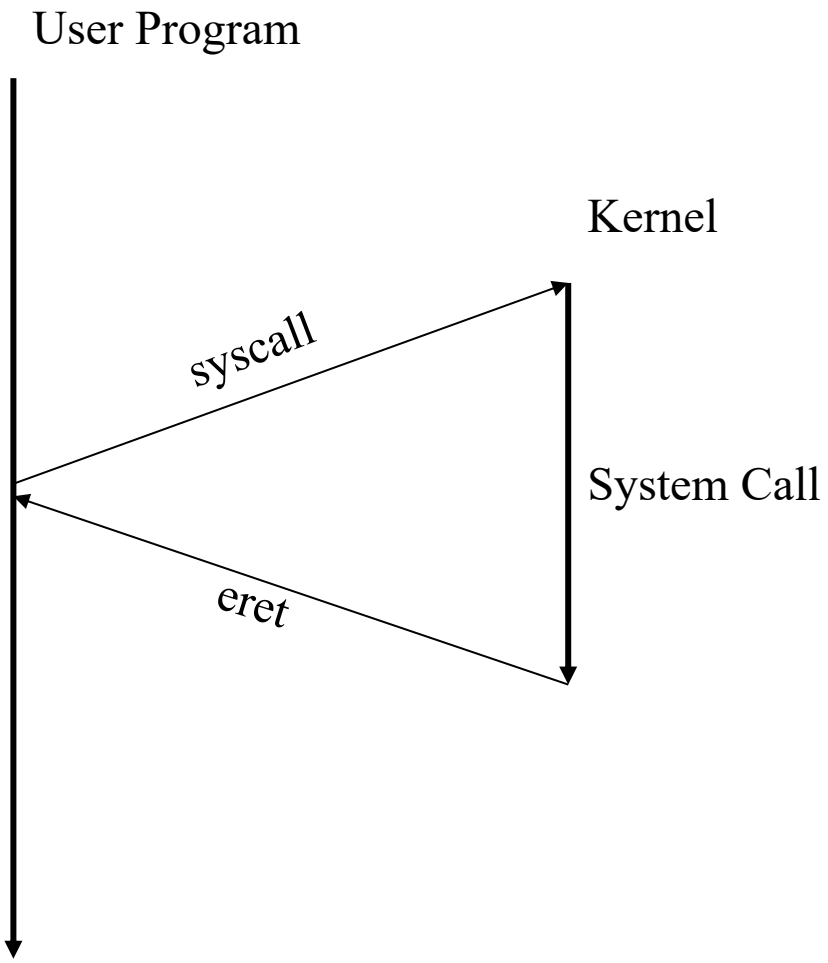
- 实验概述
- 实验内容
  - 完成系统调用的机制
  - 完成几个基本系统调用
  - 实现进程通讯需要的系统调用
  - 实现fork函数和缺页中断的处理
- 测试结果

实验概述

- 掌握系统调用的概念及流程
- 实现进程间通讯机制
- 实现 fork 函数
- 掌握缺页中断的处理流程

实验内容——系统调用的机制

- MIPS的异常模型
- 从用户态到内核态（陷入异常）
- 从内核态回到用户态（恢复现场）



writef 的调用树

- user/printf.c : void writef(char *fmt, ...)
- user/print.c : void user_lp_Print(……)
- user/printf.c : void user_myoutput(void *arg, const char *s, int l)
- user/syscall_lib.c : void syscall_putchar(char ch)
- user/syscall_wrap.S : msyscall

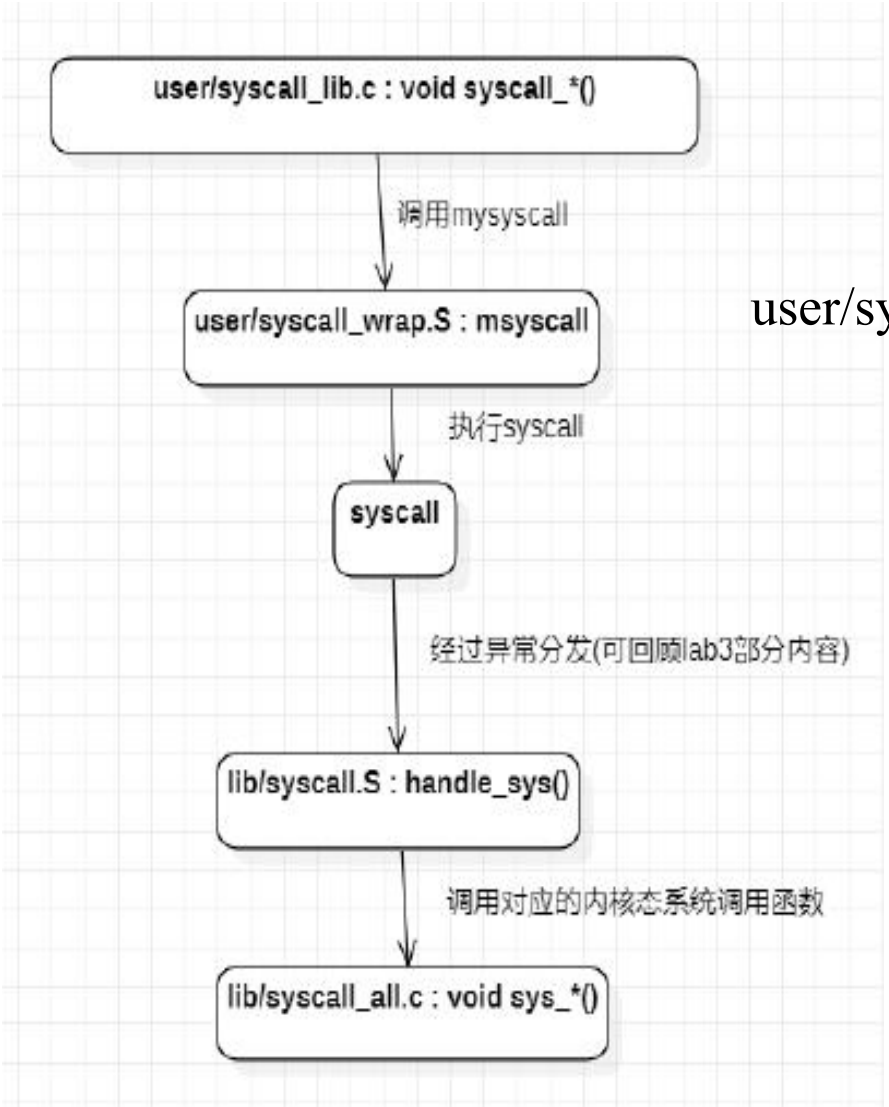
用户态

- lib/syscall.S : handle_sys

内核态

- lib/syscall_all.c : void sys_putchar(int sysno, int c, int a2, int a3, int a4, int a5)
- drivers/gxconsole/console.c : void printcharc(char ch)

系统调用的流程图示



user/syscall_lib.c：用户态的系统调用接口

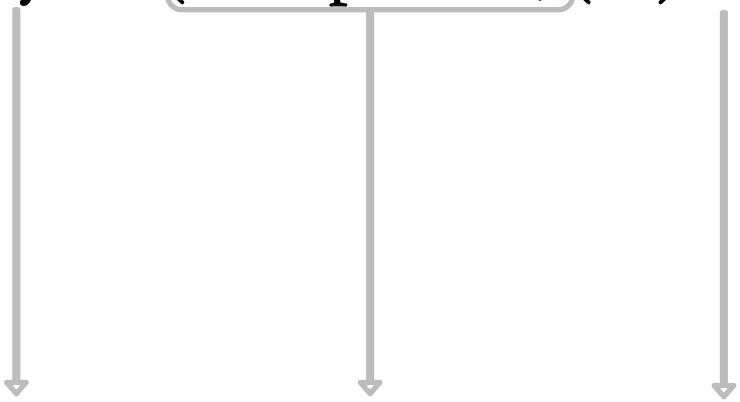
user/syscall_wrap.S：执行特权指令syscall的汇编

lib/syscall.S：内核的系统调用中断入口

lib/syscall_all.c：具体的系统调用的实现

用户态的系统调用接口

```
void syscall_putchar(char ch) {
    msyscall(SYS_putchar, (int)ch, 0, 0, 0, 0);
}
```



汇编函数
user/syscall_wrap.S

系统调用号
include/unistd.h

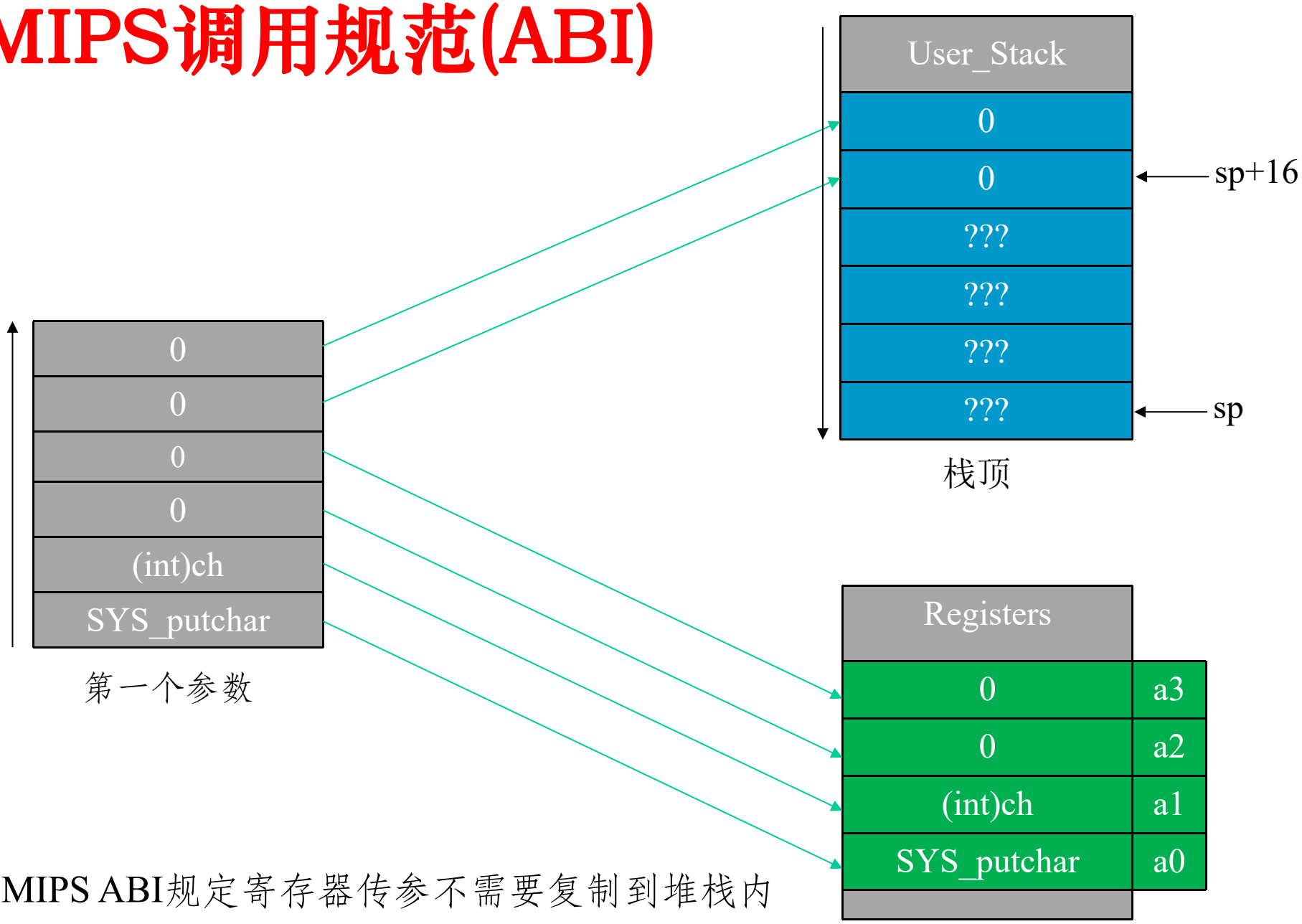
参数，补齐成 6 个参数

执行特权指令syscall

Exercise 4.1：填写user/syscall_wrap.S 中的msyscall 函数，使得用户部分的系统调用机制可以正常工作。

- LEAF(msyscall)
- // TODO: execute a `syscall` instruction and return from msyscall
- END(msyscall)

MIPS调用规范(ABI)



※MIPS ABI规定寄存器传参不需要复制到堆栈内

MIPS调用规范(ABI) 续

- 在 syscall_putchar 中调用 msyscall 的反汇编

```
00000000 <syscall_putchar>:
   0:   27bdffe0        addiu   sp,sp,-32
   4:   afbf0018        sw      ra,24(sp)
   8:   00042e00        sll     a1,a0,0x18
   c:   00052e03        sra     a1,a1,0x18
  10:   afa00010        sw      zero,16(sp)
  14:   afa00014        sw      zero,20(sp)
  18:   24042537        li      a0,9527
  1c:   00003021        move    a2,zero
  20:   0c000000        jal     0 <syscall_putchar>
  24:   00003821        move    a3,zero
  28:   8fbf0018        lw      ra,24(sp)
  2c:   03e00008        jr      ra
  30:   27bd0020        addiu   sp,sp,32
```

设置(int)ch到$a1（类型char转换为int）

将第5个和第6个参数存储到栈帧

设置SYS_putchar（9527）到$a0

设置$a2

未链接的符号（msyscall）

设置$a3

内核的系统调用中断入口

- 因为特权指令syscall，处理器陷入内核态
- 根据异常向量跳转到handle_sys函数

handle_sys 函数

Exercise 4.2： 按照lib/syscall.S 中的提示，完成handle_sys 函数，使得内核部分的系统调用机制可以正常工作。

```
NESTED(handle_sys,TF_SIZE, sp)
    SAVE_ALL                            // Macro used to save trapframe
    CLI                                 // Clean Interrupt Mask
    nop
    .set at                             // Resume use of $at
    // TODO: Fetch EPC from Trapframe, calculate a proper value and store it back to trapframe.
    // TODO: Copy the syscall number into $a0.
    addiu   a0, a0, -__SYSCALL_BASE     // a0 <- relative syscall number
    sll     t0, a0, 2                   // t0 <- relative syscall number times 4
    la      t1, sys_call_table          // t1 <- syscall table base
    addu    t1, t1, t0                  // t1 <- table entry of specific syscall
    lw      t2, 0(t1)                   // t2 <- function entry of specific syscall
    lw      t0, TF_REG29(sp)            // t0 <- user's stack pointer
    lw      t3, 16(t0)                  // t3 <- the 5th argument of msyscall
    lw      t4, 20(t0)                  // t4 <- the 6th argument of msyscall
```

保存现场，屏蔽中断

通过系统调用号找到具体内核系统调用函数的入口

通过用户栈指针找回第5~6个参数

handle_sys 函数

在内核栈分配参数空间，并复制第5~6个参数；前4个参数在哪？

```
    // TODO: Allocate a space of six arguments on current kernel stack and copy the six arguments to
proper location
    jalr    t2                          // Invoke sys_* function
    nop
    // TODO: Resume current kernel stack
    sw      v0, TF_REG2(sp)             // Store return value of function sys_* (in $v0) into trapframe
    j       ret_from_exception          // Return from exeception
    nop
END(handle_sys)

sys_call_table:                         // Syscall Table
.align 2
    .word sys_putchar
    .word sys_getenvid
    .word sys_yield
    .word sys_env_destroy
```

跳转到内核系统调用函数

恢复内核的栈

将返回值保存在Trapframe的$v0寄存器

从异常中返回

系统调用表（由系统调用号来下标索引）

系统调用的实现

内核系统调用	作用
sys_putchar	输出字符（控制台驱动）
sys_getenvid	取得自身的进程ID
sys_yield	放弃处理机
sys_env_destroy	结束销毁进程
sys_set_pgfault_handler	缺页中断相关
sys_mem_alloc	物理页面分配
sys_mem_map	虚拟页面映射
sys_mem_unmap	虚拟页面去映射
sys_env_alloc	创建一个进程（fork）
sys_set_env_status	设置子进程的运行状态
sys_set_trapframe	弃用
sys_panic	造成kernel panic（调试用途）
sys_ipc_can_send	进程通讯相关
sys_ipc_recv	进程通讯相关
sys_cgetc	输入字符（控制台驱动）

系统调用的实现

Exercise 4.3：实现lib/syscall_all.c 中的int sys_mem_alloc(int sysno,u_int envid,u_int va, u_int perm) 函数。

这个系统调用的功能是分配内存，用户程序可以通过这个系统调用给该程序所允许的虚拟内存空间内存显式地分配实际的物理内存。
可能用到的函数：page_alloc，page_insert。

Exercise 4.4：实现lib/syscall_all.c 中的int sys_mem_map(int sysno,u_int srcid,u_int srcva, u_int dstid, u_dstva, u_int perm) 函数。

这个系统调用的功能是将源进程地址空间中的相应内存映射到目标进程的相应地址空间的相应虚拟内存中去。
可能用到的函数：page_alloc，page_insert，page_lookup。

系统调用的实现

Exercise 4.5： 实现lib/syscall_all.c 中的int sys_mem_unmap(int sysno,u_int envid,u_int va) 函数。

这个系统调用的功能是解除某个进程地址空间虚拟内存和物理内存之间的映射关系。可能用到的函数：page_remove。

Exercise 4.6： 实现lib/syscall_all.c 中的void sys_yield(void) 函数。

这个系统调用的功能主要用于实现用户进程对CPU的放弃。

进程间通讯 IPC

在进程控制块 PCB 中设置若干的域

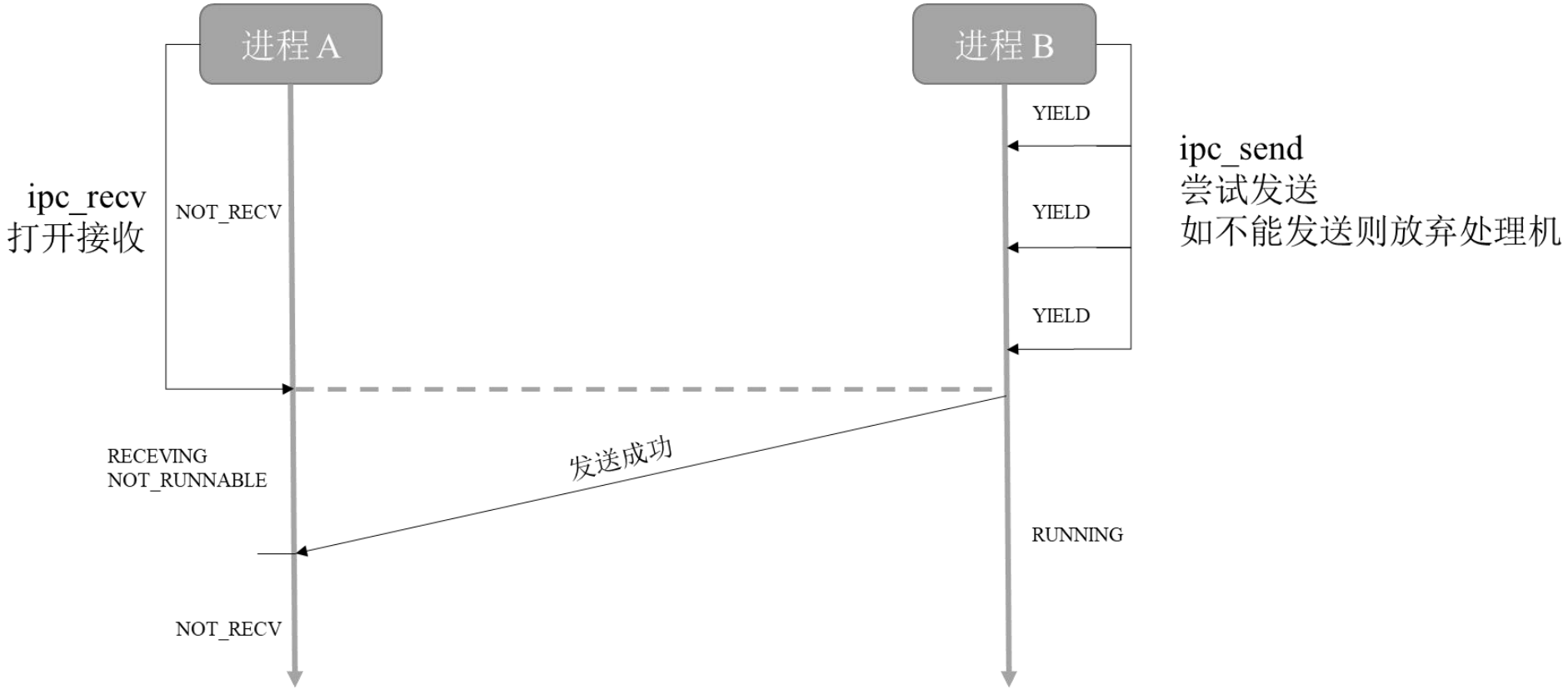
- env_ipc_value 传递的“值”
- env_ipc_from 发送者的进程ID
- env_ipc_recving 自身接收状态
- env_ipc_dstva 页面映射地址
- env_ipc_perm 页面映射权限

IPC 的核心系统调用

- 接收 **sys_ipc_recv**
- 发送 **sys_ipc_can_send**

※在Lab4的测试中，并不涉及页面的传递。但在Lab5的文件系统服务中依赖于正确的页面传递的IPC实现。

进程间通讯 IPC 续


进程 A
进程 B
YIELD
ipc_send
尝试发送
如不能发送则放弃处理机
ipc_recv
打开接收
NOT_RECV
YIELD
YIELD
RECEVING
NOT_RUNNABLE
发送成功
RUNNING
NOT_RECV

IPC 的用户接口（user/ipc.c）

```
void ipc_send(u_int whom, u_int val, u_int srcva, u_int perm) {
    int r;
    while ((r = syscall_ipc_can_send(whom, val, srcva, perm)) ==
-E_IPC_NOT_RECV) {
        syscall_yield();
    }
    if (r == 0) {
        return;
    }
    user_panic("error in ipc_send: %d", r);
}
```

IPC 的用户接口（user/ipc.c）

```
u_int ipc_recv(u_int *whom, u_int dstva, u_int *perm) {
    syscall_ipc_recv(dstva);
    if (whom) {
        *whom = env->env_ipc_from;
    }
    if (perm) {
        *perm = env->env_ipc_perm;
    }
    return env->env_ipc_value;
}
```

IPC 相关的系统调用

Exercise 4.7：

实现lib/syscall_all.c 中的void sys_ipc_recv(int sysno,u_int dstva)函数和int sys_ipc_can_send(int sysno,u_int envid, u_int value, u_int srcva,u_int perm) 函数。

注：由于在我们的用户程序中，会大量使用srcva 为0 的调用来表示不需要传递物理页面，因此在编写相关函数时也需要注意此种情况

Fork

- 这是一个非原子的用户态的 fork 函数。所以主要工作是在 user/fork.c 中完成；
- Fork主要流程：
  - 父进程：
    - 调用syscall_env_alloc创建子进程env，并设置相关参数
    - duppage复制页表内容，并设置相关标志（如PTE_COW）
    - 为子进程分配异常处理栈，设置缺页处理函数入口
  - 子进程：
    - 设置用户env指针
    - 运行时触发缺页中断，在内核态保存现场，返回到用户态进行缺页处理
    - 用户态pgfault处理缺页中断，拷贝页面
    - 用户态恢复现场，返回执行

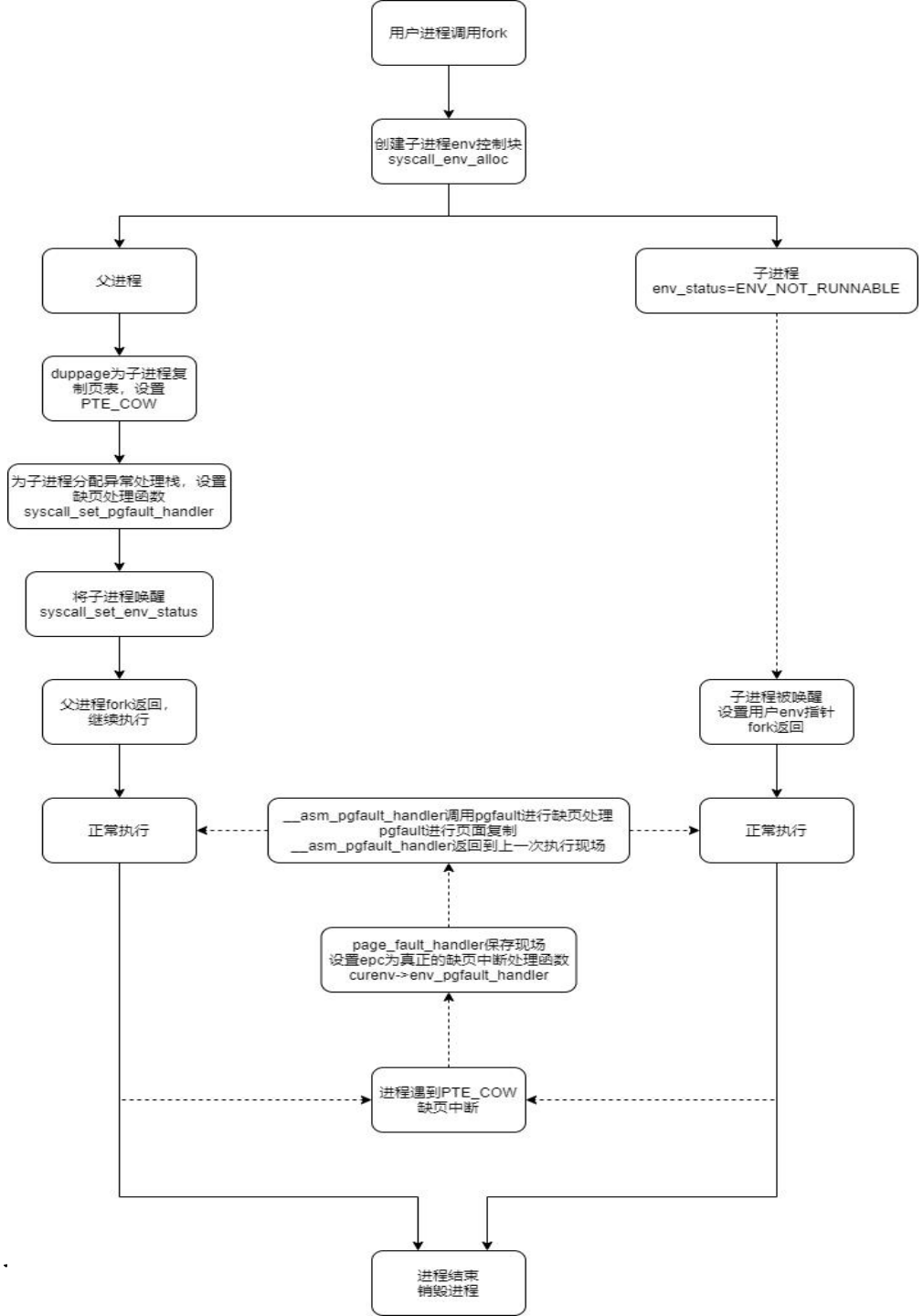
写时复制 & 缺页中断处理

- 父进程在duppage函数为子进程页表设置标识：
  - 只读页面： 按照相同权限（只读）映射给子进程即可
  - 共享页面： 即具有PTE_LIBRARY标记的页面，这类页面需要保持共享的可写的状态
  - 写时复制页面： 即具有PTE_COW标记的页面，这类页面是上一次的fork的duppage的结果
  - 可写页面： 需要给父进程和子进程的页表项都加上PTE_COW标记
- 父进程为子进程分配好异常处理栈，地址空间为[UXSTACKTOP-BY2PG, UXSTACKTOP)
- 父进程为子进程设置好缺页处理函数入口（trap.c中的page_fault_handler）

写时复制 & 缺页中断处理

- 子进程触发缺页中断陷入内核态，执行page_fault_handler函数（trap.c中）
- page_fault_handler函数保存现场，设置异常处理栈，设置epc为真正的缺页处理函数（__asm_pgfault_handler）
- __asm_pgfault_handler调用pgfault函数，进行缺页处理，拷贝页面。
- __asm_pgfault_handler恢复现场，子进程返回执行

Fork流程图


用户进程调用fork
创建子进程env控制块
syscall_env_alloc
父进程
子进程
env_status=ENV_NOT_RUNNABLE
duppage为子进程复制页表，设置PTE_COW
为子进程分配异常处理栈，设置缺页处理函数
syscall_set_pgfault_handler
将子进程唤醒
syscall_set_env_status
父进程fork返回，继续执行
子进程被唤醒
设置用户env指针
fork返回
正常执行
__asm_pgfault_handler调用pgfault进行缺页处理
pgfault进行页面复制
__asm_pgfault_handler返回到上一次执行现场
正常执行
page_fault_handler保存现场
设置epc为真正的缺页中断处理函数
curenv->env_pgfault_handler
进程遇到PTE_COW
缺页中断
进程结束
销毁进程

Fork的实现

Exercise 4.8： 请根据指导书步骤提示以及代码中的注释提示，填写 lib/syscall_all.c 中的 sys_env_alloc 函数。

该函数比较简单，即分配一个子进程env控制块，并设定一定初始值，例如env_status，env_pri，env_tf等。

Exercise 4.9： 按照指导书的提示，填写 user/fork.c 中的fork 函数中关于sys_env_alloc的部分和“子进程”执行的部分

Fork中包括“父进程部分”和“子进程部分”，其中子进程只需要对用户态的env指针进行更新即可。

Fork的实现

Exercise 4.10： 结合代码注释以及上述提示，填写 user/fork.c 中的 duppage 函数。

Fork需要复制父进程的页表给子进程，duppage函数负责复制单个页面映射，以及页面标志设置：

- 只读页面：按照相同权限（只读）映射给子进程即可
- 共享页面：即具有PTE_LIBRARY标记的页面，这类页面需要保持共享的可写的状态
- 写时复制页面：即具有PTE_COW标记的页面，这类页面是上一次的fork的duppage的结果
- 可写页面：需要给父进程和子进程的页表项都加上PTE_COW标记

Fork的实现

Exercise 4.11： 根据指导书提示以及代码注释，完成 lib/traps.c 中的 page_fault_handler 函数，设置好异常处理栈以及epc寄存器的值。

该函数比较简单，主要工作为：保存现场，设置异常栈sp寄存器的值，设置epc寄存器的值，使得返回用户态时能够进入缺页处理函数。

Exercise 4.12： 完成 lib/syscall_all.c的sys_set_pgfault_handler 函数。

该函数比较简单，设置好子进程异常处理栈顶和异常处理函数即可。

Fork的实现

Exercise 4.13： 填写 user/fork.c 中的 pgfault函数

Pgfault是子进程缺页处理的重点，主要工作为：分配临时页面，复制页面内容到临时页面上

可能需要的函数：syscall_mem_alloc，syscall_mem_map，syscall_mem_unmap

Exercise 4.14： 填写 lib/syscall_all.c 中的sys_set_env_status 函数

该函数比较简单，设置好env_status，以及env从调度队列的插入或删除。

Fork的实现

Exercise 4.15 填写 user/fork.c 中的fork 函数中关于“父进程”执行的部分。

调用上述已经实现的duppage，syscall_set_pgfault_handler，syscall_set_env_status来完成这个过程。具体fork流程见前边讲述，根据你的理解完成整个fork流程

课下自行测试须知

- 你可以模仿fktest.c的方式构造用户进程
  - 首先在user目录下新建xxx.c
  - 在xxx.c中#include “lib.h”
  - 在user/Makefile中的编译目标加上xxx.x和xxx.b
  - 在init/init.c中用ENV_CREATE或者ENV_CREATE_PRIORITY创建用户进程user_xxx
- 用户进程执行过程：
  - 入口： entry.S、libos.c
  - 主函数： void umain()

课下测试结果

- 当你完成所有的Lab4的内容时，你才能使用我们提供的两个测试用的用户程序fktest（fork测试）、pingpong（fork以及IPC的测试）。
- fktest会连续fork两次，所以会一共产生3个进程。它们分别输出不同的文本。
- pingpong在进程fork之后互相发送数字，数字从递增到10结束。
- 如果你只完成了第一部分，想自行测试syscall功能，可以自行构造测试样例进行测试

提交评测结果

```
remote: [ DEFAULT COMPILE OK and your total score is 5/100 now. ]
remote: [ [TEST 1.1] SYSCALL test begin. ]
```

- 完成第一部分syscall和ipc后：

```
remote: [ [TEST 1.1] SYSCALL COMPILE OK ]
remote: [ PASSED:15 ]
remote: [ TOTAL:15 ]
remote: [ You got 10 in [TEST 1.1] SYSCALL test and your total score is 15 now. ]
remote: [ [TEST 1.2] IPC test begin. ]
```

```
remote: [ IPC COMPILE OK ]
remote: [ PASSED:7 ]
remote: [ TOTAL:7 ]
remote: [ You got 35 in [TEST 1.2] IPC test and your total score is 50 now. ]
remote: [ [TEST 2.1] FORK test begin. ]
```

- 完成第二部分fork后：

```
remote: [ FORK COMPILE OK and your total score is 50/100 now. ]
remote: [ PASSED:8 ]
remote: [ TOTAL:8 ]
remote: [ You got 16 in [TEST 2.1] FORK test and your total score is 66 now. ]
remote: [ [TEST 2.2] FORK-IPC test begin. ]
```

```
remote: [ FORK-IPC COMPILE OK and your total score is 66/100 now. ]
remote: [ PASSED:8 ]
remote: [ TOTAL:8 ]
remote: [ You got 8 in [TEST 2.2.1] FORK-IPC-WEAK test and your total score is 74 now. ]
remote: [ PASSED:13 ]
remote: [ TOTAL:13 ]
remote: [ You got 26 in [TEST 2.2.2] FORK-IPC-STRONG test and your total score is 100 now. ]
remote: [ You got 100 (of 100) this time. Sat Apr 18 23:51:33 CST 2020 ]
```

```
remote: [ Congratulations! You have passed the current lab. ]
```

lab4-extra测试范围说明

- lab4-extra 的范围是 系统调用和进程间通信
- 即需要完成Exercise4.1-4.7
- 但是课下分发的代码并没有能够在这一阶段能够直接运行的用户程序，请自行通过可能的手段（如编写测试用的用户程序）进行测试。