Week10 用户进程

一、 实验概述

本次实验中，我们将完成用户进程的创建，并使相应的程序代码在用户特权级执行。执行过程中，用户通过系统调用实现具体功能。

因为我们的操作系统中没有编译器，这里我们的选择是把一个编译好的用户程序（可执行文件）运行起来。

二、实验目的

1. 掌握用户进程的创建过程
2. 掌握用户进程特权级切换以及执行过程
3. 掌握系统调用过程

三、实验项目整体框架概述

// Week10

```
├── kern
│   ├── debug
│   ├── driver
│   ├── fs
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   │   ├── readline.c
│   │   └── stdio.c
│   ├── mm
│   ├── process
│   │   ├── entry.S
│   │   ├── proc.c //修改和增添了部分内容
│   │   ├── proc.h
│   │   └── switch.S
│   ├── schedule
│   │   ├── sched.c
│   │   └── sched.h
│   ├── sync
│   │   └── sync.h
│   ├── syscall
│   │   ├── syscall.c// 处理对应的系统调用
│   │   └── syscall.h
│   └── trap
│       ├── trap.c
│       ├── trapentry.S
```

```
│   └── trap.h
├── libs
│   ├── atomic.h
│   ├── defs.h
│   ├── elf.h //elf文件的相关内容
│   ├── error.h
│   ├── hash.c
│   ├── list.h
│   ├── printfmt.c
│   ├── rand.c
│   ├── riscv.h
│   ├── sbi.h
│   ├── stdarg.h
│   ├── stdio.h
│   ├── stdlib.h
│   ├── string.c
│   ├── string.h
│   └── unistd.h
├── Makefile
├── tools
│   ├── function.mk
│   ├── kernel.ld
│   └── user.ld//用户程序的链接文件
└── user用户程序
    ├── hello.c
    └── libs
        ├── initcode.S
        ├── panic.c
        ├── stdio.c
        ├── syscall.c
        ├── syscall.h
        ├── ulib.c
        ├── ulib.h
        └── umain.c
```

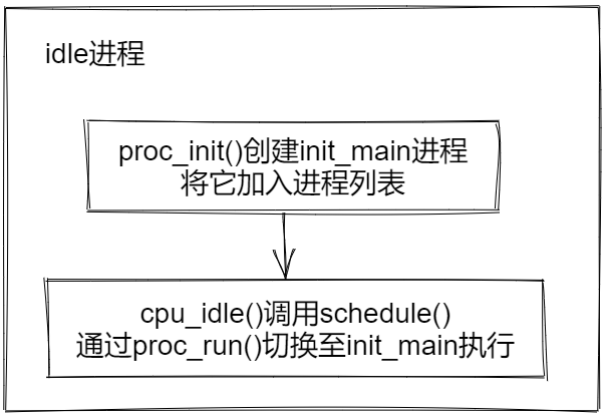
四、实验内容

1. 创建用户进程
2. 用户进程通过exec()切换至用户特权级，并执行程序（可执行文件）
3. 用户程序通过系统调用打印信息

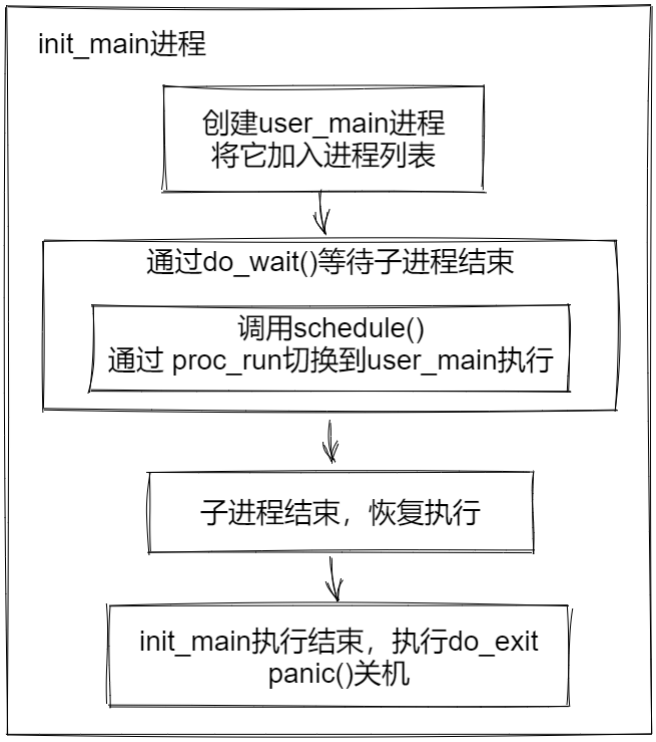
五、实验过程概述及相关知识点

在内核线程中创建一个用户进程

1）idle进程创建init进程，用于管理之后产生的子进程们

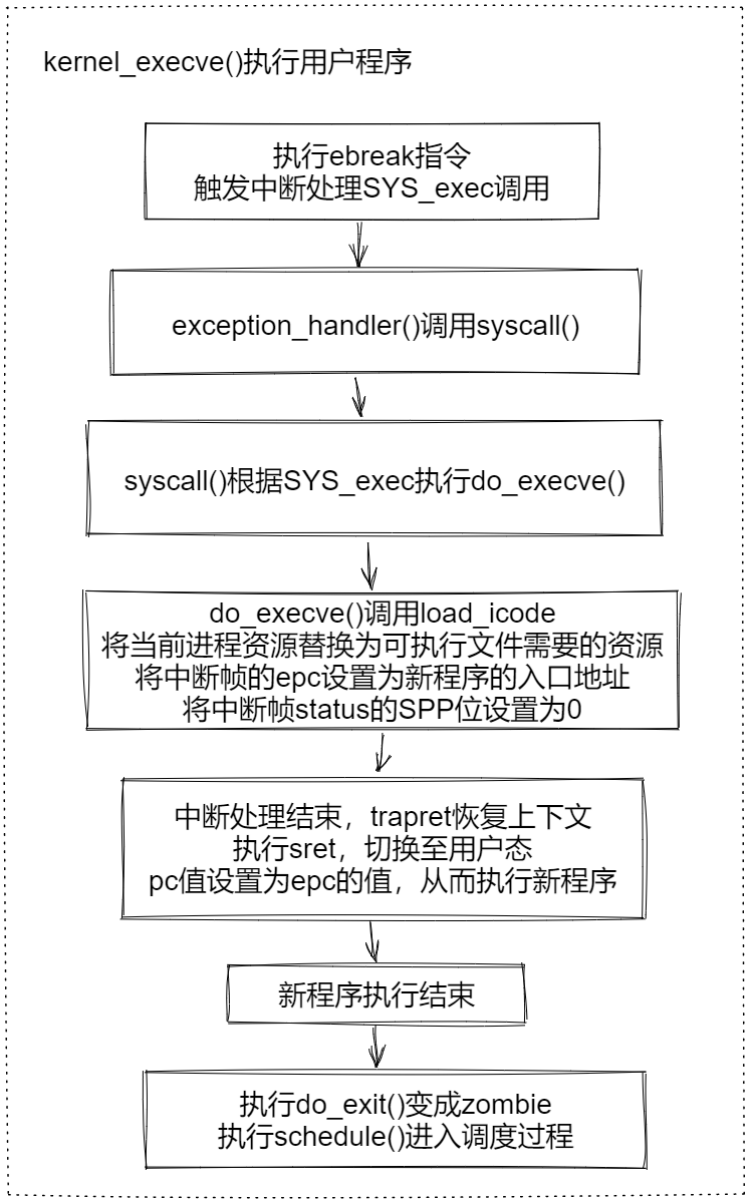


2）init进程创建用户进程user_main，注意此时仅分配了进程资源，但尚未执行exec()，也没有切换到用户态。



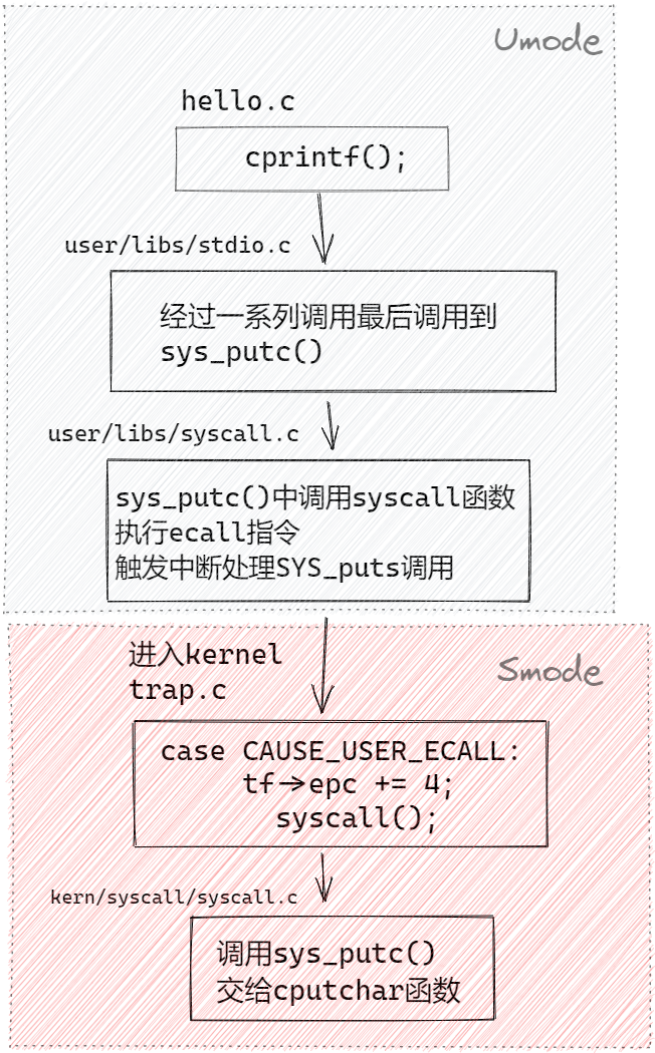
执行用户进程

user_main执行时调用exec()，将当前进程资源替换为新程序。同时，通过更改status的SPP位使得中断返回时返回至Umode；并通过将epc的值设置为新程序入口，使得中断返回时的sret指令将pc的值设为epc的值，从而中断返回后跳转执行新程序。



> **SPP**
>
> sstatus中的SPP域用于保存trap发生时的权限模式（即处于什么态），trap结束后sret指令会根据SPP的值恢复权限模式。0为Umode，1为Smode。

系统调用的处理流程



系统调用

操作系统应当提供给用户程序一些接口，让用户程序使用操作系统提供的服务。这些接口就是**系统调用**。用户程序在用户态运行(U mode), 系统调用在内核态执行(S mode)。这里有一个CPU的特权级切换的过程, 要用到`ecall`指令从U mode进入S mode。想想我们之前用`ecall`做过什么？在S mode调用OpenSBI提供的M mode接口。当时我们用`ecall`进入了M mode, 剩下的事情就交给OpenSBI来完成，然后我们收到OpenSBI返回的结果。

现在我们用`ecall`从U mode进入S mode之后，对应的处理需要我们编写内核系统调用的代码来完成。

另外，我们总不能让用户程序里直接调用`ecall`。通常我们会把这样的系统调用操作封装成一个个的函数，作为"标准库"提供给用户使用。例如在linux里，写一个C程序时使用`printf()`函数进行输出, 实际上是要进行`write()`的系统调用，通过内核把输出打印到命令行或其他地方。

对于用户进程的管理，有四个系统调用比较重要。

`sys_fork()`：把当前的进程复制一份，创建一个子进程，原先的进程是父进程。接下来两个进程都会收到`sys_fork()`的返回值，如果返回0说明当前位于子进程中，返回一个非0的值（子进程的PID）说明当前位于父进程中。然后就可以根据返回值的不同，在两个进程里进行不同的处理。

`sys_exec()`：在当前的进程下，停止原先正在运行的程序，开始执行一个新程序。PID不变，但是内存空间要重新分配，执行的机器代码发生了改变。我们可以用`fork()`和`exec()`配合，在当前程序不停止的情况下，开始执行另一个程序。

`sys_exit()`：退出当前的进程。

> `sys_wait()`：挂起当前的进程，等到特定条件满足的时候再继续执行。

六、实验内容

第一步. 用户进程的创建

week9的手册中，我们在`proc_init()`函数里初始化进程的时候, 认为启动时运行的ucore程序, 是一个内核进程("第0个"内核进程)，并将其初始化为`idleproc`进程。然后我们新建了一个内核进程执行`init_main()`函数。

我们比较week9和week10的`init_main()`有何不同。

```
// kern/process/proc.c (week9)
static int init_main(void *arg) {
    cprintf("this initproc, pid = %d, name = \"%s\"\n", current->pid,
get_proc_name(current));
    cprintf("%s.\n", (const char *)arg);
    return 0;
}

// kern/process/proc.c (week10)
static int init_main(void *arg) {
    size_t nr_free_pages_store = nr_free_pages();
    size_t kernel_allocated_store = kallocated();

    int pid = kernel_thread(user_main, NULL, 0);
    if (pid <= 0) {
        panic("create user_main failed.\n");
    }

    while (do_wait(0, NULL) == 0) {
        schedule();
    }

    cprintf("all user-mode processes have quit.\n");
    assert(initproc->cptr == NULL && initproc->yptr == NULL && initproc->optr ==
NULL);
    assert(nr_process == 2);
    assert(list_next(&proc_list) == &(initproc->list_link));
    assert(list_prev(&proc_list) == &(initproc->list_link));

    cprintf("init check memory pass.\n");
    return 0;
}
```

注意到，week10新建了一个进程执行函数`user_main()`，这个进程里我们将通过exec()实现用户进程的执行。在这一步中，我们为新的进程创建了proc_struct，并将其加入进程链表，但此时我们仅进行了进程的创建（fork，即复制了父进程的副本），用户进程资源的分配是在之后进程执行是通过exec完成的。

第二步. 用户进程的执行

接下来，init_main作为管理者，调用do_wait()等待子进程的结束。等待的过程中通过schedule()调度user_main执行。

我们来看下user_main()做了些什么：

```
// kern/process/proc.c
#define __KERNEL_EXECVE(name, binary, size) ({                          \
            cprintf("kernel_execve: pid = %d, name = \"%s\".\n",        \
                    current->pid, name);                                \
            kernel_execve(name, binary, (size_t)(size));                \
        })

#define KERNEL_EXECVE(x) ({                                             \
            extern unsigned char _binary_obj___user_##x##_out_start[],  \
                _binary_obj___user_##x##_out_size[];                    \
            __KERNEL_EXECVE(#x, _binary_obj___user_##x##_out_start,     \
                            _binary_obj___user_##x##_out_size);         \
        })


// user_main - kernel thread used to exec a user program
static int
user_main(void *arg) {
    KERNEL_EXECVE(hello);
    panic("user_main execve failed.\n");
}
```

`_binary_obj___user_##x##_out_start`和`_binary_obj___user_##x##_out_size`都是编译的时候自动生成的符号。注意这里的`##x##`，按照C语言宏的语法，会直接把x的变量名代替进去。

于是，我们在`user_main()`所做的，就是执行了

```
kern_execve("hello",
_binary_obj___user_hello_out_start,_binary_obj___hello_exit_out_size)
```

这么一个函数。

如果你熟悉`exec()`函数，或许已经猜到这里我们做了什么。

实际上，就是加载了存储在这个位置的程序`hello`并在`user_main`这个进程里开始执行。这时`user_main`就从内核进程变成了用户进程。

第三步. 用户进程通过exec执行用户程序

接下来，让我们了解user_main是如何通过exec执行用户程序的。

在内核中我们实现了`do_execve()`函数:

```
// kern/process/proc.c
// do_execve - call exit_mmap(mm)&put_pgdir(mm) to reclaim memory space of
current process
//           - call load_icode to setup new memory space accroding binary prog.
int do_execve(const char *name, size_t len, unsigned char *binary, size_t size)
{
    struct mm_struct *mm = current->mm;
```

```
    if (!user_mem_check(mm, (uintptr_t)name, len, 0)) { //检查name的内存空间能否被访
问
        return -E_INVAL;
    }
    if (len > PROC_NAME_LEN) { //进程名字的长度有上限 PROC_NAME_LEN，在proc.h定义
        len = PROC_NAME_LEN;
    }
    char local_name[PROC_NAME_LEN + 1];
    memset(local_name, 0, sizeof(local_name));
    memcpy(local_name, name, len);

    if (mm != NULL) {
        cputs("mm != NULL");
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);//把进程当前占用的内存释放，之后重新分配内存
        }
        current->mm = NULL;
    }
    //把新的程序加载到当前进程里的工作都在load_icode()函数里完成
    int ret;
    if ((ret = load_icode(binary, size)) != 0) {
        goto execve_exit;//返回不为0，则加载失败
    }
    set_proc_name(current, local_name);
    return 0;

execve_exit:
    do_exit(ret);
    panic("already exit: %e.\n", ret);
}
```

在`do_execve()`函数中，我们调用`load_icode()`为用户进程分配新的资源，包括mm、页表、用户栈等等，更重的是下面两行代码：

```
tf->epc = elf->e_entry;
tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
```

第一行，将进程中断帧的epc寄存器内写入了可执行文件的程序入口。

第二行，将sstatus的SPP域置0，使得中断处理结束sret返回时返回至Umode。

将tf内容改写后，中断帧的上下文发生了变化，当中断结束返回（sret）时，会切换上下文把epc的值写入pc，从而下一步执行可执行文件的程序；并且sret还会根据SPP的值（它认为是触发中断时的原始权限），返回到原本的权限，此时SPP为0，所以会从S态切换到用户态。

这样我们就通过**中断返回**实现了用户进程的切换和执行。

等等？？？？

我们触发中断了吗？没有啊

那怎么办？

回想一下，我们目前想要做的是让user_main()调用kernel_execve()实现exec的功能。而内核中我们是实现的是do_execve()。kernel_execve()执行时是处于Smode的，因此他有足够的权限调用do_execve()执行对应的功能，但是直接执行只是更改了中断帧，如果没有发生中断返回（sret指令）那么就不会恢复上下文，不会切换至Umode，并跳转至epc继续执行。

这里，我们可以人为的触发中断：

```
// kernel_execve - do SYS_exec syscall to exec a user program called by
user_main kernel_thread
static int
kernel_execve(const char *name, unsigned char *binary, size_t size) {
    int64_t ret=0, len = strlen(name);
    asm volatile(
        "li a0, %1\n"
        "lw a1, %2\n"
        "lw a2, %3\n"
        "lw a3, %4\n"
        "lw a4, %5\n"
        "li a7, 10\n"
        "ebreak\n"
        "sw a0, %0\n"
        : "=m"(ret)
        : "i"(SYS_exec), "m"(name), "m"(len), "m"(binary), "m"(size)
        : "memory");
    //cprintf("ret = %d\n", ret);
    return ret;
}
```

我们通过ebreak指令进入trap处理，再通过trap.c中的处理函数转发至do_execve()执行，这样do_execve()就是在中断中更改了中断帧，可以使得trap返回时完成我们想要实现的效果：切换至用户态，执行用户程序。

第四步. 用户程序通过cprintf调用系统调用输出信息

本次实验的用户程序在`user/hello.c`中实现，代码如下：

```
#include <stdio.h>
#include <ulib.h>

int main(void) {
    cprintf("Hello world!!.\n");
    cprintf("I am process %d.\n", getpid());
    cprintf("hello pass.\n");
    return 0;
}
```

hello应用程序只是输出一些字符串，并通过系统调用`sys_getpid`（在`getpid`函数中调用）输出代表hello应用程序执行的用户进程的进程标识--`pid`。

在用户程序里使用的`cprintf()`也是在`user/libs/stdio.c`重新实现的，和之前比最大的区别是，打印字符的时候需要经过系统调用`sys_putc()`，而不能直接调用`sbi_console_putchar()`。这是自然的，因为只有在Supervisor Mode才能通过`ecall`调用Machine Mode的OpenSBI接口，而在用户态(U Mode)就不能直接使用M mode的接口，而是要通过系统调用。

如何能够找到hello应用程序

这需要分析ucore和hello是如何编译的。然后在本实验源码目录下执行`make V=`，在最下面可得到如下输出：

```
  ...
+ cc user/hello.c
riscv64-unknown-elf-gcc -Iuser/ -mcmodel=medany -O2 -std=gnu99 -Wno-unused
-fno-builtin -Wall -nostdinc  -fno-stack-protector -ffunction-sections -
fdata-sections -Ilibs/ -Iuser/include/ -Iuser/libs/ -c user/hello.c -o
obj/user/hello.o
+ cc user/libs/panic.c
riscv64-unknown-elf-gcc -Iuser/libs/ -mcmodel=medany -O2 -std=gnu99 -Wno-
unused -fno-builtin -Wall -nostdinc  -fno-stack-protector -ffunction-
sections -fdata-sections -Ilibs/ -Iuser/include/ -Iuser/libs/ -c
user/libs/panic.c -o obj/user/libs/panic.o
+ cc user/libs/syscall.c
riscv64-unknown-elf-gcc -Iuser/libs/ -mcmodel=medany -O2 -std=gnu99 -Wno-
unused -fno-builtin -Wall -nostdinc  -fno-stack-protector -ffunction-
sections -fdata-sections -Ilibs/ -Iuser/include/ -Iuser/libs/ -c
user/libs/syscall.c -o obj/user/libs/syscall.o
+ cc user/libs/ulib.c
riscv64-unknown-elf-gcc -Iuser/libs/ -mcmodel=medany -O2 -std=gnu99 -Wno-
unused -fno-builtin -Wall -nostdinc  -fno-stack-protector -ffunction-
sections -fdata-sections -Ilibs/ -Iuser/include/ -Iuser/libs/ -c
user/libs/ulib.c -o obj/user/libs/ulib.o
+ cc user/libs/initcode.S
riscv64-unknown-elf-gcc -Iuser/libs/ -mcmodel=medany -O2 -std=gnu99 -Wno-
unused -fno-builtin -Wall -nostdinc  -fno-stack-protector -ffunction-
sections -fdata-sections -Ilibs/ -Iuser/include/ -Iuser/libs/ -c
user/libs/initcode.S -o obj/user/libs/initcode.o
+ cc user/libs/stdio.c
riscv64-unknown-elf-gcc -Iuser/libs/ -mcmodel=medany -O2 -std=gnu99 -Wno-
unused -fno-builtin -Wall -nostdinc  -fno-stack-protector -ffunction-
sections -fdata-sections -Ilibs/ -Iuser/include/ -Iuser/libs/ -c
user/libs/stdio.c -o obj/user/libs/stdio.o
+ cc user/libs/umain.c
riscv64-unknown-elf-gcc -Iuser/libs/ -mcmodel=medany -O2 -std=gnu99 -Wno-
unused -fno-builtin -Wall -nostdinc  -fno-stack-protector -ffunction-
sections -fdata-sections -Ilibs/ -Iuser/include/ -Iuser/libs/ -c
user/libs/umain.c -o obj/user/libs/umain.o
riscv64-unknown-elf-ld -m elf64lriscv -nostdlib --gc-sections -T
tools/user.ld -o obj/__user_hello.out  obj/user/libs/panic.o
obj/user/libs/syscall.o obj/user/libs/ulib.o obj/user/libs/initcode.o
obj/user/libs/stdio.o obj/user/libs/umain.o  obj/libs/string.o
obj/libs/printfmt.o obj/libs/hash.o obj/libs/rand.o obj/user/hello.o
+ ld bin/kernel
```

```
riscv64-unknown-elf-ld -m elf64lriscv -nostdlib --gc-sections -T
tools/kernel.ld -o bin/kernel  obj/kern/init/entry.o obj/kern/init/init.o
obj/kern/libs/stdio.o obj/kern/libs/readline.o obj/kern/debug/panic.o
obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/driver/ide.o
obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/picirq.o
obj/kern/driver/intr.o obj/kern/trap/trap.o obj/kern/trap/trapentry.o
obj/kern/mm/pmm.o obj/kern/mm/vmm.o obj/kern/mm/swap.o
obj/kern/mm/kmalloc.o obj/kern/mm/swap_fifo.o obj/kern/mm/default_pmm.o
obj/kern/fs/swapfs.o obj/kern/process/entry.o obj/kern/process/switch.o
obj/kern/process/proc.o obj/kern/schedule/sched.o
obj/kern/syscall/syscall.o  obj/libs/string.o obj/libs/printfmt.o
obj/libs/hash.o obj/libs/rand.o --format=binary  obj/__user_hello.out --
format=default
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.bin
```

从中可以看出，hello应用程序不仅仅是`hello.c`，还包含了支持hello应用程序的用户态库：

- `user/libs/initcode.S`：所有应用程序的起始用户态执行地址“`_start`”，调用`umain`函数。
- `user/libs/umain.c`：实现了`umain`函数，这是所有应用程序执行的第一个C函数，它将调用应用程序的`main`函数，并在`main`函数结束后调用`exit`函数，而`exit`函数最终将调用`sys_exit`系统调用，让操作系统回收进程资源。
- `user/libs/ulib.[ch]`：实现了最小的C函数库，除了一些与系统调用无关的函数，其他函数是对访问系统调用的包装。
- `user/libs/syscall.[ch]`：用户层发出系统调用的具体实现。
- `user/libs/stdio.c`：实现`cprintf`函数，通过系统调用`sys_putc`来完成字符输出。
- `user/libs/panic.c`：实现`__panic/__warn`函数，通过系统调用`sys_exit`完成用户进程退出。

除了这些用户态库函数实现外，还有一些`libs/*.[ch]`是操作系统内核和应用程序共用的函数实现。这些用户库函数其实在本质上与UNIX系统中的标准libc没有区别，只是实现得很简单，但hello应用程序的正确执行离不开这些库函数。

在make的最后一步执行了一个ld命令，把hello应用程序的执行码`obj/__user_hello.out`连接在了`ucore kernel`的末尾。且ld命令会在kernel中会把`__user_hello.out`的位置和大小记录在全局变量`_binary_obj___user_hello_out_start`和`_binary_obj___user_hello_out_size`中，这样这个hello用户程序就能够和ucore内核一起被OpenSBI加载到内存里中，并且通过这两个全局变量定位hello用户程序执行码的起始位置和大小。在后面的与文件系统相关的实验后，ucore会提供一个简单的文件系统，那时所有的用户程序就都不再用这种方法进行加载了，而可以用大家熟悉的文件方式进行加载了。

用户进程的虚拟地址空间

在tools/user.ld描述了用户程序的用户虚拟空间的执行入口虚拟地址：

```
SECTIONS {
 /* Load programs at this address: "." means the current address */
 . = 0x800020;
```

在tools/kernel.ld描述了操作系统的内核虚拟空间的起始入口虚拟地址：

```
BASE_ADDRESS = 0xFFFFFFFFC0200000;

SECTIONS
{
 /* Load the kernel at this address: "." means the current address */
 . = BASE_ADDRESS;
```

> 这样ucore把用户进程的虚拟地址空间分了两块，一块与内核线程一样，是所有用户进程都共享的内核虚拟地址空间，映射到同样的物理内存空间中，这样在物理内存中只需放置一份内核代码，使得用户进程从用户态进入核心态时，内核代码可以统一应对不同的内核程序；另外一块是用户虚拟地址空间，虽然虚拟地址范围一样，但映射到不同且没有交集的物理内存空间中。这样当ucore把用户进程的执行代码（即应用程序的执行代码）和数据（即应用程序的全局变量等）放到用户虚拟地址空间中时，确保了各个进程不会“非法”访问到其他进程的物理内存空间。

那么，如何在用户程序调用系统调用`sys_putc()`呢?

我们可以在用户态为程序提供一个调用的接口，真正的处理都在内核态进行。

系统调用转发

首先我们在头文件里定义一些系统调用的编号。

```
// libs/unistd.h
#ifndef __LIBS_UNISTD_H__
#define __LIBS_UNISTD_H__

#define T_SYSCALL           0x80

/* syscall number */
#define SYS_exit            1
#define SYS_fork            2
#define SYS_wait            3
#define SYS_exec            4
#define SYS_clone           5
#define SYS_yield           10
#define SYS_sleep           11
#define SYS_kill            12
#define SYS_gettime         17
#define SYS_getpid          18
#define SYS_brk             19
#define SYS_mmap            20
#define SYS_munmap          21
#define SYS_shmem           22
#define SYS_putc            30
#define SYS_pgdir           31

/* SYS_fork flags */
#define CLONE_VM            0x00000100  // set if VM shared between processes
#define CLONE_THREAD        0x00000200  // thread group

#endif /* !__LIBS_UNISTD_H__ */
```

我们注意系统调用需要在S mode执行，因此我们需要进行模式的切换。在用户态进行系统调用的核心操作是，通过内联汇编进行`ecall`环境调用。这将产生一个trap, 进入S mode进行异常处理。

```
// user/libs/syscall.c
```

```
#include <defs.h>
#include <unistd.h>
#include <stdarg.h>
#include <syscall.h>
#define MAX_ARGS            5
static inline int syscall(int num, ...) {
    //va_list, va_start, va_arg都是C语言处理参数个数不定的函数的宏
    //在stdarg.h里定义
    va_list ap; //ap: 参数列表(此时未初始化)
    va_start(ap, num); //初始化参数列表, 从num开始
    //First, va_start initializes the list of variable arguments as a va_list.
    uint64_t a[MAX_ARGS];
    int i, ret;
    for (i = 0; i < MAX_ARGS; i ++) { //把参数依次取出
           /*Subsequent executions of va_arg yield the values of the additional
arguments
           in the same order as passed to the function.*/
        a[i] = va_arg(ap, uint64_t);
    }
    va_end(ap); //Finally, va_end shall be executed before the function returns.
    asm volatile (
        "ld a0, %1\n"
        "ld a1, %2\n"
        "ld a2, %3\n"
        "ld a3, %4\n"
        "ld a4, %5\n"
        "ld a5, %6\n"
        "ecall\n"
        "sd a0, %0"
        : "=m" (ret)
        : "m"(num), "m"(a[0]), "m"(a[1]), "m"(a[2]), "m"(a[3]), "m"(a[4])
        :"memory");
    //num存到a0寄存器， a[0]存到a1寄存器
    //ecall的返回值存到ret
    return ret;
}
int sys_exit(int error_code) { return syscall(SYS_exit, error_code); }
int sys_fork(void) { return syscall(SYS_fork); }
int sys_wait(int pid, int *store) { return syscall(SYS_wait, pid, store); }
int sys_yield(void) { return syscall(SYS_yield);}
int sys_kill(int pid) { return syscall(SYS_kill, pid); }
int sys_getpid(void) { return syscall(SYS_getpid); }
int sys_putc(int c) { return syscall(SYS_putc, c); }
```

我们下面看看trap.c是如何转发这个系统调用的。

```
// kern/trap/trap.c
void exception_handler(struct trapframe *tf) {
    int ret;
    switch (tf->cause) { //通过中断帧里 scause寄存器的数值，判断出当前是来自USER_ECALL
的异常
        case CAUSE_USER_ECALL:
            //cprintf("Environment call from U-mode\n");
            tf->epc += 4;
            //sepc寄存器是产生异常的指令的位置，在异常处理结束后，会回到sepc的位置继续执行
            //对于ecall, 我们希望sepc寄存器要指向产生异常的指令(ecall)的下一条指令
            //否则就会回到ecall执行再执行一次ecall, 无限循环
```

```
            syscall();// 进行系统调用处理
            break;
        /*other cases .... */
    }
}
// kern/syscall/syscall.c
#include <unistd.h>
#include <proc.h>
#include <syscall.h>
#include <trap.h>
#include <stdio.h>
#include <pmm.h>
#include <assert.h>
//这里把系统调用进一步转发给proc.c的do_exit(), do_fork()等函数
static int sys_exit(uint64_t arg[]) {
    int error_code = (int)arg[0];
    return do_exit(error_code);
}
static int sys_fork(uint64_t arg[]) {
    struct trapframe *tf = current->tf;
    uintptr_t stack = tf->gpr.sp;
    return do_fork(0, stack, tf);
}
static int sys_wait(uint64_t arg[]) {
    int pid = (int)arg[0];
    int *store = (int *)arg[1];
    return do_wait(pid, store);
}
static int sys_exec(uint64_t arg[]) {
    const char *name = (const char *)arg[0];
    size_t len = (size_t)arg[1];
    unsigned char *binary = (unsigned char *)arg[2];
    size_t size = (size_t)arg[3];
    //用户态调用的exec(), 归根结底是do_execve()
    return do_execve(name, len, binary, size);
}
static int sys_yield(uint64_t arg[]) {
    return do_yield();
}
static int sys_kill(uint64_t arg[]) {
    int pid = (int)arg[0];
    return do_kill(pid);
}
static int sys_getpid(uint64_t arg[]) {
    return current->pid;
}
static int sys_putc(uint64_t arg[]) {
    int c = (int)arg[0];
    cputchar(c);
    return 0;
}
//这里定义了函数指针的数组syscalls, 把每个系统调用编号的下标上初始化为对应的函数指针
static int (*syscalls[])(uint64_t arg[]) = {
    [SYS_exit]              sys_exit,
    [SYS_fork]              sys_fork,
    [SYS_wait]              sys_wait,
    [SYS_exec]              sys_exec,
    [SYS_yield]             sys_yield,
```

```
    [SYS_kill]                  sys_kill,
    [SYS_getpid]                sys_getpid,
    [SYS_putc]                  sys_putc,
};

#define NUM_SYSCALLS        ((sizeof(syscalls)) / (sizeof(syscalls[0])))

void syscall(void) {
    struct trapframe *tf = current->tf;
    uint64_t arg[5];
    int num = tf->gpr.a0;//a0寄存器保存了系统调用编号
    if (num >= 0 && num < NUM_SYSCALLS) {//防止syscalls[num]下标越界
        if (syscalls[num] != NULL) {
            arg[0] = tf->gpr.a1;
            arg[1] = tf->gpr.a2;
            arg[2] = tf->gpr.a3;
            arg[3] = tf->gpr.a4;
            arg[4] = tf->gpr.a5;
            tf->gpr.a0 = syscalls[num](arg);
            //把寄存器里的参数取出来，转发给系统调用编号对应的函数进行处理
            return ;
        }
    }
    //如果执行到这里，说明传入的系统调用编号还没有被实现，就崩掉了。
    print_trapframe(tf);
    panic("undefined syscall %d, pid = %d, name = %s.\n",
            num, current->pid, current->name);
}
```

这样我们就在U mode通过ecall触发trap进入S mode，在trap中实现了系统调用的执行。

第五步. 了解用户进程的退出和等待

退出

在进程执行完工作后，需要退出，释放资源，正我们在`do_execve`函数末尾看到的一样，退出进程是调用`do_exit`函数来实现的。

```
execve_exit:
    do_exit(ret);
    panic("already exit: %e.\n", ret);
```

```
// do_exit - called by sys_exit
//   1. call exit_mmap & put_pgdir & mm_destroy to free the almost all memory space of process
//   2. set process' state as PROC_ZOMBIE, then call wakeup_proc(parent) to ask parent reclaim itself.
//   3. call scheduler to switch to other process
int
do_exit(int error_code) {
    if (current == idleproc) {
        panic("idleproc exit.\n");
    }
    if (current == initproc) {
```

```
        panic("initproc exit.\n");
    }
    struct mm_struct *mm = current->mm;
    if (mm != NULL) {
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
    current->state = PROC_ZOMBIE;
    current->exit_code = error_code;
    bool intr_flag;
    struct proc_struct *proc;
    local_intr_save(intr_flag);
    {
        proc = current->parent;
        if (proc->wait_state == WT_CHILD) {
            wakeup_proc(proc);
        }
        while (current->cptr != NULL) {
            proc = current->cptr;
            current->cptr = proc->optr;

            proc->yptr = NULL;
            if ((proc->optr = initproc->cptr) != NULL) {
                initproc->cptr->yptr = proc;
            }
            proc->parent = initproc;
            initproc->cptr = proc;
            if (proc->state == PROC_ZOMBIE) {
                if (initproc->wait_state == WT_CHILD) {
                    wakeup_proc(initproc);
                }
            }
        }
    }
    local_intr_restore(intr_flag);
    schedule();
    panic("do_exit will not return!! %d.\n", current->pid);
}
```

1.如果是内核线程则不需要回收空间

2.如果是用户进程，就开始回收，首先执行`lcr3(boot_cr3);`切换到内核的页表上，这样用户进程就只能在内核的虚拟地址空间上执行，因为内核权限高。如果当前进程的被调用数减一后等于0，那么就没有其他进程在使用了，就可以进行回收，先回收内存资源，调用`exit_mmap`函数释放`mm`中的`vma`描述的进程合法空间中实际分配的内存，然后把对应的页表项内容清空，最后把页表项和页目录表清空。然后调用`put_pgdir`函数释放页目录表所占用的内存。最后调用`mm_destroy`释放`vma`与`mm`的内存。把`mm`置为NULL，表示与当前进程相关的用户虚拟内存空间和对应的内存管 理成员变量所占的内核虚拟内存空间已经回收完毕；

3.设置进程的状态为`PROC_ZOMBIE`表示该进程要死了，等待父进程来回收资源，回收内核栈和进程控制块。当前进程的退出码为`error_code`表示该进程已经不能被调度。

4.如果当前进程的父进程处于等待子进程的状态，则唤醒父进程让父进程回收资源。

5.如果该进程还有子进程，那么就指向第一个孩子，把后面的孩子全部置为空，然后把孩子过继给内核线程`initproc`，把子进程插入到`initproc`的孩子链表中，如果某个子进程的状态时要死的状态，并且`initproc`的状态时等待孩子的状态，则唤醒`initproc`来回收子进程的资源。

6.然后开启中断，执行`schedule`函数，选择新的进程执行

等待

那么父进程如何完成对子进程的最后回收工作呢？这要求父进程要执行`wait`用户函数或`wait_pid`用户函数，这两个函数的区别是，`wait`函数等待任意子进程的结束通知，而`wait_pid`函数等待进程id号为pid的子进程结束通知。这两个函数最终访问`sys_wait`系统调用接口让ucore来完成对子进程的最后回收工作，即回收子进程的内核栈和进程控制块所占内存空间，具体流程如下：

```
// do_wait - wait one OR any children with PROC_ZOMBIE state, and free memory
space of kernel stack
//         - proc struct of this child.
// NOTE: only after do_wait function, all resources of the child proces are
free.
int
do_wait(int pid, int *code_store) {
    struct mm_struct *mm = current->mm;
    if (code_store != NULL) {
        if (!user_mem_check(mm, (uintptr_t)code_store, sizeof(int), 1)) {
            return -E_INVAL;
        }
    }

    struct proc_struct *proc;
    bool intr_flag, haskid;
repeat:
    haskid = 0;
    if (pid != 0) {
        proc = find_proc(pid);
        if (proc != NULL && proc->parent == current) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found;
            }
        }
    }
    else {
        proc = current->cptr;
        for (; proc != NULL; proc = proc->optr) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found;
            }
        }
    }
    if (haskid) {
        current->state = PROC_SLEEPING;
        current->wait_state = WT_CHILD;
        schedule();
        if (current->flags & PF_EXITING) {
            do_exit(-E_KILLED);
```

```
        }
        goto repeat;
    }
    return -E_BAD_PROC;

found:
    if (proc == idleproc || proc == initproc) {
        panic("wait idleproc or initproc.\n");
    }
    if (code_store != NULL) {
        *code_store = proc->exit_code;
    }
    local_intr_save(intr_flag);
    {
        unhash_proc(proc);
        remove_links(proc);
    }
    local_intr_restore(intr_flag);
    put_kstack(proc);
    kfree(proc);
    return 0;
}
```

1.首先进行检查

2.若`pid`等于0，就去找对应的孩子进程，否则就任意的一个快死的孩子进程。如果此子进程的执行状态不为`PROC_ZOMBIE`，表明此子进程还没有退出，则当前进程只 好设置自己的执行状态为`PROC_SLEEPING`，睡眠原因为`WT_CHILD`（即等待子进程退 出），调用`schedule()`函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复该步骤执行；

3.如果此子进程的执行状态为`PROC_ZOMBIE`，表明此子进程处于退出状态，需要当前进程 （即子进程的父进程）完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列`proc_list`和`hash_list`中删除，并释放子进程的内核堆栈和进程控制块。自此，子进程才彻底 地结束了它的执行过程，消除了它所占用的所有资源。

七、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. 进程的创建
2. exec执行用户进程
3. 系统调用的转发和实现
4. 进程的退出和等待

八、下一实验简单介绍

下一次实验，我们将进行进程调度。