Lab5

流程概述

给应用程序一个用户态运行环境

- 建立用户代码/数据段
- 创建内核线程（为创建用户进程做好准备）

  调用`kernel_thread(init_main, NULL, 0);`建立内核线程
- 创建用户进程的外壳

  调用`alloc_proc`建立用户进程的外壳
- 填写用户进程具体内容

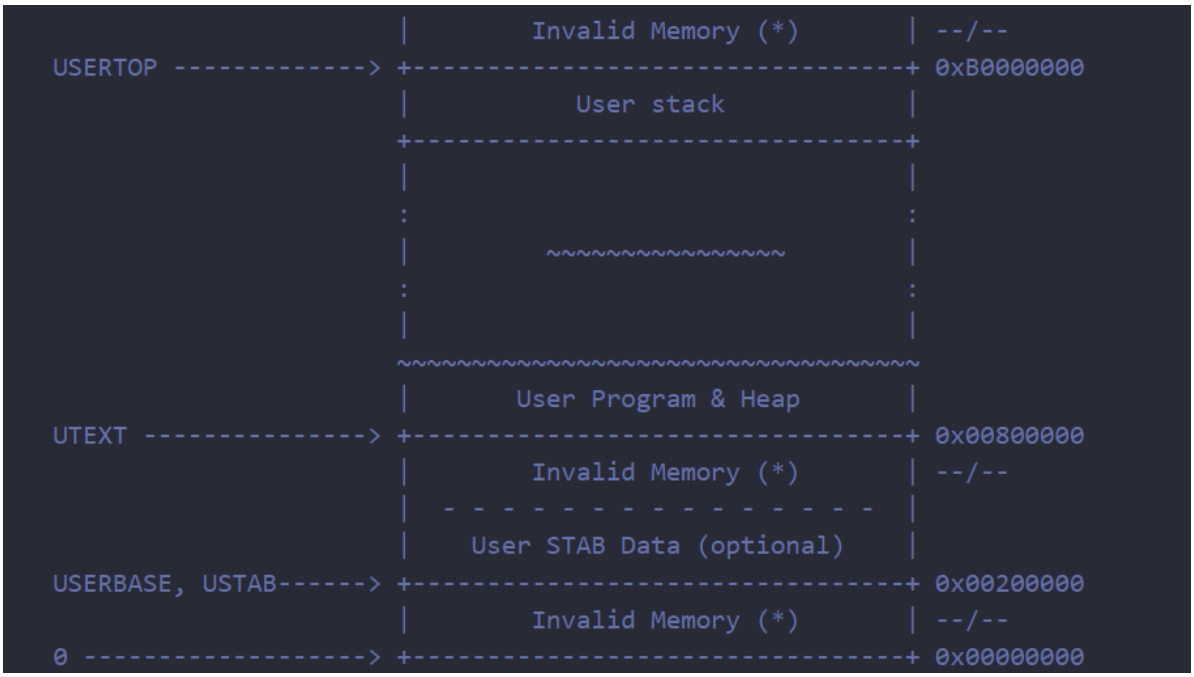
  `kernel_execve()`函数，使用内联汇编调用syscall：`SYS_exec`加载用户进程的内容
- 执行用户进程
- 完成系统调用
- 结束用户进程

  调用`do_wait()`和`do_kill`函数释放进程的空间

进程的内存布局

内存管理

建立用户态虚拟内存



用户态进程虚拟内存布局如上图

其中进程可以使用的部分为`User stack` 和 `User Program & Heap`，建立进程的堆和栈。`User STAB Data (optional)`区域存放调试信息 ,其余区域为非法区域，访问该区域则产生page fault，这样设置便于限制进程的访问空间

进程管理

U态和S态的转换

> SPP： 指示前一个特权级别是否是S态。对于从内核模式返回到用户SPP模式的情况，这个位应该被设置为0，表示当前是从S态返回。

通过设置SPP的位来实现U态和s态的切换：

进行系统调用之后，我们在trap返回的时候调用了`SRET`指令，这时只要`sstatus`寄存器的SPP二进制位为0，就会切换到U态，SPP存储的是“进入trap之前来自什么特权级”，也就是说我们这里ebreak之后SPP的数值为1，`SRET`之后会回到S态，在内核态执行用户程序。所以load_icode()函数在构造新进程的时候，会把`SSTATUS_SPP`设置为0，使得`SRET`的时候能回到U态。

执行ELF格式二进制代码

`do_execve`函数

- 检查用户空间范围的合法性
- 清空用户态的页表等内存空间
- 调用`lode_icode`函数加载进程执行程序内容

`load_icode`函数

- 创建一个新的内存管理空间，创建新的页表
- 先找到elf格式的执行程序，找到header，再根据header找到各代码段的地址，建立VMA，设置好各段的读写属性
- 拷贝要执行的代码的内容到进程空间
- 清空BSS段的内容
- 建立进程的用户态的堆栈空间，设置好映射关系
- 把页表起始地址从内核地址换到新建立的页表的起始地址
- 设置好trapframe，实现从S态跳转到U态（设置`tf->gpr.sp，tf->epc，tf->status`三个寄存器的值）

进程复制

建立子进程

`do_fork`函数

- 分配新的`proc_struct`
- 为进程分配`kernel stack`
- `copy_mm()`为新进程创建新虚拟空间(根据`clone_flag`来区别，如果`clone_flag & CLONE_VM`为真，则意味着新进程将共享父进程的虚拟内存空间（即地址空间）。否则，新进程将获得自己独立的虚拟内存空间。)
- 设置`trapframe & context`，拷贝父进程的`trapframe`到新进程，修改部分值（eax = 0系统调用的返回值，esp ，eip = forkret），使用`copy_thread()`完成上述工作
- 添加进程结构体到`proc_list`
- 使用`wakeup_proc()`唤醒新进程
- 父进程的系统调用返回值（父进程`do_fork`返回子进程的pid，子进程do_fork返回的是0）

练习

练习1: 加载应用程序并执行（需要编码）

do_execve函数调用`load_icode`（位于kern/process/proc.c中）来加载并解析一个处于内存中的ELF执行文件格式的应用程序。你需要补充`load_icode`的第6步，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好`proc_struct`结构中的成员变量trapframe中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的trapframe内容。

请在实验报告中简要说明你的设计实现过程。

- 请简要描述这个用户态进程被ucore选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过。

`load_icode`的功能要求：

1. `tf->gpr.sp`：这个字段应该设置为用户堆栈顶部的地址（即sp寄存器的值）。在用户模式下运行时，进程会使用这个堆栈。
2. `tf->epc`：这个字段应该设置为用户程序的入口点（即sepc寄存器的值）。当进程从内核模式返回到用户模式时，它将在这个地址开始执行。

3. `tf->status`：这个字段应该设置为适合用户程序的状态寄存器值（即sstatus寄存器的值）。特别需要注意的是`spp`和`spie`位：
    - `SPP`（Supervisor Previous Privilege）：指示前一个特权级别是否是S态。对于从内核模式返回到用户模式的情况，这个位应该被设置为0，表示当前是从S态返回。
    - `SPIE`（Supervisor Previous Interrupt Enable）：指示前一个特权级别中中断是否启用。这个位应该根据你的需求来设置，如果你希望在用户模式下允许中断，那么就将其设置为1；否则，设置为0以禁止中断。

实现：

```
//预定义宏，用户栈顶部位置
tf->gpr.sp=USTACKTOP;

//elf->e_entry是程序入口地址,定义在elf.h中
tf->epc=elf->e_entry;
//SPP: Previous Privilege Mode,0 for user mode,1 for supervisor mode
//SPIE: Supervisor Previous Interrupt Enable,1 for enable,0 for disable
tf->status=(sstatus|SSTATUS_SPIE)&(~SSTATUS_SPP);
//首先，使用按位或操作符 | 将sstatus和SSTATUS_SPIE组合起来。用于提示前一个特权级别中中断是否启用。
//接下来，使用按位与操作符 & 和按位取反操作符 ~ 来清除sstatus中的SPP位。这样可以确保在返回到用户模式时，前一个特权级别被标记为用户模式。
```

用户态进程进入running态到第一条指令执行的过程：

1. 为内存管理的数据结构mm分配空间并进行空间初始化
2. 创建用户进程内存空间创建页表，将mm的pgdir设置为页目录的虚地址
3. 调用`lode_icode()`函数，在用户进程内存建立BSS，为用户进程栈分配空间
4. 写入内存管理结构mm中的内容
5. 清空原来的中断帧，建立新的中断帧，并恢复各寄存器的值
6. CPU执行用户态进程第一条指令

练习2: 父进程复制自己的内存空间给子进程（需要编码）

创建子进程的函数`do_fork`在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。具体是通过`copy_range`函数（位于kern/mm/pmm.c中）实现的，请补充`copy_range`的实现，确保能够正确执行。

请在实验报告中简要说明你的设计实现过程。

fork系统调用生成子进程的过程：

- 父进程调用fork系统调用，进入中断处理机制
- 调用`syscall`函数，交给`sys_fork`函数处理
- 调用`do_fork()`函数，调用`alloc_proc()`函数初始化一个`proc_struct`结构体，即用户进程的外壳

- 遍历父进程所有合法虚拟内存空间，并将内容复制到子进程的内存空间
- 分配子进程的内核栈，设置子进程的trapframe和上下文
- 将父子进程的关系进行设置，并且加入到`proc_list`当中
- 唤醒子进程并返回子进程的PID

`copy_range`的功能要求：

- 对于每个页，首先调用 `get_pte` 函数在**源进程**的页表中找到对应的 PTE。如果找不到，说明该页没有映射，将 `start` 调整为下一个页的起始地址，然后继续下一轮循环
- 调用 `get_pte` 函数在**目标进程**的页表中找到对应的 PTE。如果 PTE 不存在，说明需要为进程分配一个新的页表，并获取到对应的 PTE。
- 如果**源进程** 的 PTE 标记为有效（`PTE_V`），则表示该页已经映射到物理页框上。接下来，分配一个新的物理页框给**目的进程**
- 找到`src_kvaddr`：**源进程**的内核虚拟地址，找到`dst_kvaddr`：**目的进程**的内核虚拟地址，从`rc_kvaddr`到`dst_kvaddr`进行内存复制，大小为PGSIZE
- 建立**目的进程**物理地址与线性地址start的映射

实现：

所补充部分是内存的复制和建立目的进程物理地址和线性地址start的映射

```
memcpy(page2kva(npage), page2kva(page), PGSIZE);
//使用memcpy函数复制原进程的页面内容到npage当中，调用page2kva获取页面的内核虚拟地址

if ((ret = page_insert(to, npage, start, perm)) != 0) {
    return ret;
}
//page_insert建立npage到线性地址start的映射
```

- 如何设计实现`Copy on Write`机制？给出概要设计，鼓励给出详细设计。

> Copy-on-write（简称COW）的基本概念是指如果有多个使用者对一个资源A（比如内存块）进行读操作，则每个使用者只需获得一个指向同一个资源A的指针，就可以该资源了。若某使用者需要对这个资源A进行写操作，系统会对该资源进行拷贝操作，从而使得该“写操作”使用者获得一个该资源A的“私有”拷贝—资源B，可对资源B进行写操作。该“写操作”使用者对资源B的改变对于其他的使用者而言是不可见的，因为其他使用者看到的还是资源A。

COW设计思路

主要目的是实现进程执行fork系统调用进行复制的时候，父进程暂时共享相同的物理内存页，而当其中一个进程需要对内存进行修改的时候，再额外创建一个自己私有的物理内存页，将共享的内容复制过去，然后在自己的内存页中进行修改；

所以要实现两个部分：

- fork操作的时候不直接复制内存。比如copy_range函数内部，不实际进行内存的复制，而是将子进程和父进程的虚拟页映射上同一个物理页面，然后在分别在这两个进程的虚拟页对应的PTE部分将这个页置成是不可写的，同时利用PTE中的保留位将这个页设置成共享的页面，这样的话如果应用程序试图写某一个共享页就会产生页访问异常，从而可以将控制权交给操作系统进行处理

- 出现了内存页访问异常的时候，会将共享的内存页复制一份，然后在新的内存页进行修改。比如在page_fault中添加一个fault的处理情况——当前的异常是由于尝试写了某一个共享页面引起的。处理方式为额外申请分配一个物理页面，然后将当前的共享页的内容复制过去，建立出错的线性地址与新创建的物理页面的映射关系，将PTE设置设置成非共享的，然后查询原先共享的页面是否还有其他进程在共享使用，如果没有了，就修改PTE，把共享标记修改为写标记，就可以实现正常的写操作了。

练习3: 阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现（不需要编码）

请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析。并回答如下问题：

- 请分析fork/exec/wait/exit的执行流程。重点关注哪些操作是在用户态完成，哪些是在内核态完成？内核态与用户态程序是如何交错执行的？内核态执行结果是如何返回给用户程序的？
- 请给出ucore中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）。（字符方式画即可）

函数分析

`fork()`：在用户态调用，调用`sys_fork()`的syscall时进入内核态，调用了`do_fork()`函数来创建新的进程。`do_fork()`函数是一个内核级函数，它负责分配新的进程资源，并将其放入内存中。然后，它会设置当前进程的上下文，使其能够执行新的程序，为子进程创建用户栈、内核栈等。并将控制权交给新的进程。如果创建或执行过程中出现错误，那么它会立即返回-1表示失败。 当`do_fork()`函数完成其任务后，它会返回到`sys_fork()`函数，并返回创建的进程的PID用户程序。

`exec()`：在用户态调用，调用`sys_exec()`的syscall时进入内核态，`sys_exec()`函数调用了`do_execve()`函数来执行新的程序。`do_execve()`函数是一个内核级函数，它负责加载指定的二进制代码，并将其放入内存中。然后，它会设置当前进程的上下文，使其能够执行新的程序，并将控制权交给新的程序。如果加载或执行过程中出现错误，那么它会立即返回-1表示失败。 当`do_execve()`函数完成其任务后，它会返回到`sys_exec()`函数，并将结果返回给用户程序

`wait()`：在用户态调用，该函数会循环查看子进程的状态。调用`sys_wait()`系统调用来执行等待子进程的操作。`sys_wait()`系统调用是一个内核级函数，它负责检查指定的子进程是否已经结束。如果子进程已经结束，那么它会释放该进程所占用的所有资源，并将该进程的状态码或错误代码保存在内存区域中。然后，它会将这个状态码或错误代码复制到用户程序提供的存储位置，并返回0表示成功。如果子进程还没有结束，那么它会立即返回-1表示失败。 当`sys_wait()`系统调用完成其任务后，它会返回到`wait()`函数，并将结果返回给用户程序

`exit()`：在用户态调用，用于结束当前进程，释放掉一部分内存（还有一部分可能由父进程进行释放），并将该进程标记为ZMOBIE状态。如果它的父进程处于等待子进程退出的状态，则唤醒父进程，将子进程交给initproc处理，并进行进程调度。

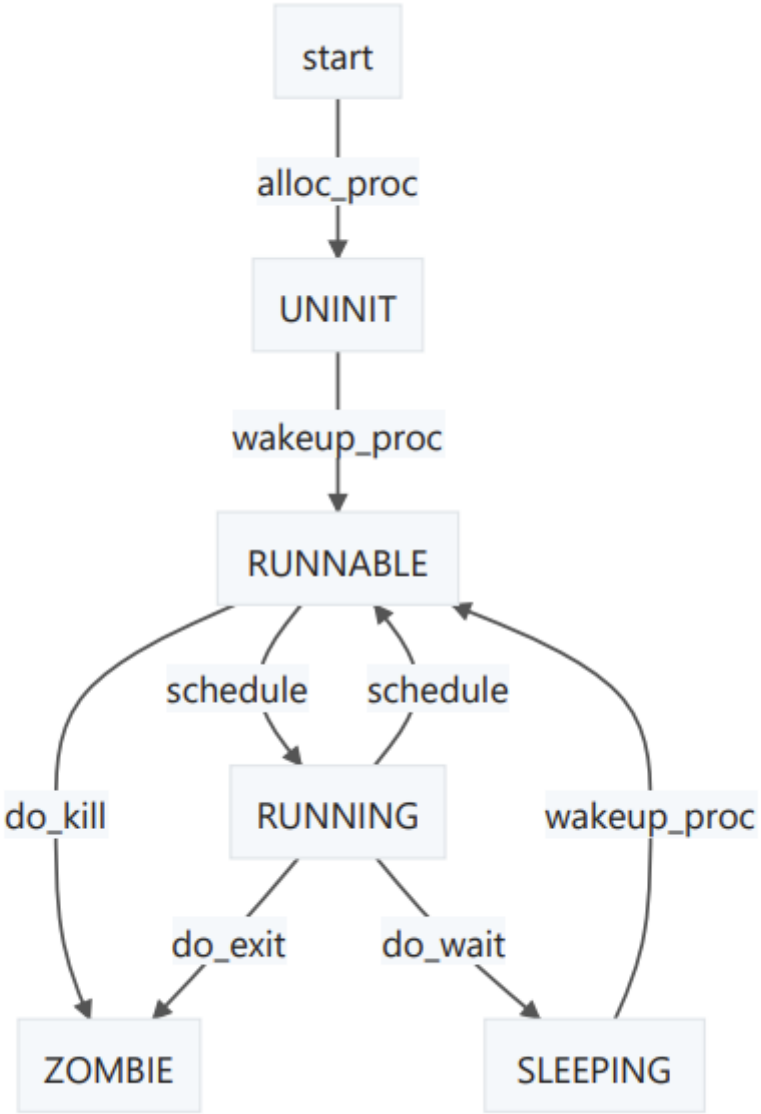
内核态与用户态程序是如何交错执行的?

当用户态进程调用一个系统调用时，它会将控制权交给内核，并传递系统调用的类型值给内核。然后，内核会在内核态下执行相应的操作，并将结果返回给进程。在这个过程中，进程是无法执行任何其他操作的。

内核态执行结果是如何返回给用户程序的?

内核将系统调用的结果（通常是状态码或错误代码）保存在一个寄存器中，并切换到用户态。当进程重新获得控制权时，它可以读取寄存器中的结果，并根据需要进行处理。

用户态进程的执行状态生命周期图



扩展练习 Challenge

1. 实现 Copy on Write （COW）机制

   给出实现源码,测试用例和设计报告（包括在cow情况下的各种状态转换（类似有限状态自动机）的说明）。

   这个扩展练习涉及到本实验和上一个实验“虚拟内存管理”。在ucore操作系统中，当一个用户父进程创建自己的子进程时，父进程会把其申请的用户空间设置为只读，子进程可共享父进程占用的用户内存空间中的页面（这就是一个共享的资源）。当其中任何一个进程修改此用户内存空间中的某页面时，ucore会通过page fault异常获知该操作，并完成拷贝内存页面，使得两个进程都有各自的内存页面。这样一个进程所做的修改不会被另外一个进程可见了。请在ucore中实现这样的COW机制。

   由于COW实现比较复杂，容易引入bug，请参考 https://dirtycow.ninja/ 看看能否在ucore的COW实现中模拟这个错误和解决方案。需要有解释。

   这是一个big challenge.

2. 说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么？

1

实现方案，具体见COW设计文档

2

在ucore中，用户程序是在创建`init_main`内核线程的时候，通过`exec()`系统调用被加载到内存中的。具体的流程为：

> kernel_thread(user_main, NULL, 0)->KERNEL_EXECVE2->kernel_execve->ebreak进入中断->syscall()->sys_exec->do_execve

在`KERNEL_EXECVE`中设置了两个全局变量`_binary_obj___user_hello_out_start`和`_binary_obj___user_hello_out_size`用于记录用户程序执行码的起始位置和大小。执行make后，在make的最后一步执行了一个ld命令，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 bootloader 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。

下面从`de_execve`函数调用的`load_icode`函数进行用户程序加载到内存的流程介绍：

- 为用户进程创建一个新的mm和页表
- 获取elf的文件头和程序头
- 遍历每个程序段头，创建新的vma把每个程序段的内容进行保存（其中把BSS段的内容载入到数据段的背后，由于BSS 段是一个特殊的段，它通常包含程序中未初始化的全局变量。这些变量未被初始化，故不需要把文件中的内容复制到内存，所以将BSS段的内容全部清零）
- 为进程分配内存，并将程序段内容复制到进程的内存空间
- 建立用户栈，设置当前进程的`mm`、`cr3`和`trapframe`，保存程序入口地址，用户栈顶地址和特权级别等信息

ucore的操作系统内核没有虚拟内存管理机制，所有的程序都必须一次性地加载到物理内存中，现代操作系统通常使用更复杂的技术来加载和运行用户程序。例如，它们可能会使用动态链接技术，将程序的不同部分（如代码、数据和库）分别加载到内存中，并在运行时进行连接。现代操作系统还通常使用虚拟内存管理机制，允许程序超过物理内存的大小，并在需要时将页面换入或换出物理内存。区别产生的原因在于：ucore的加载方式简单易懂，适合于教学目的。而现代操作系统的加载方式则更加复杂和灵活，适用于实际生产环境中的各种需求。