

- Lab 6: RV64 fork系统调用
  - \*\*1 \*\*实验目的
  - 2 实验内容及要求
  - 3 实验步骤
    - 3.1 搭建实验环境
    - 3.2 创建进程
      - 3.2.1 批处理系统和进程
      - 3.2.2 fork 系统调用
      - 3.2.3 exec 系统调用
      - 3.2.4 wait 系统调用
      - 3.2.5 exit 系统调用
    - 3.3 代码实现 (100%)
      - 3.3.1 其他代码调整 (已完成)
      - 3.3.2 实现 fork 系统调用 (50%)
      - 3.3.2 实现 exec 系统调用 (20%)
      - 3.3.3 实现 wait 系统调用 (20%)
      - 3.3.4 实现 exit 系统调用 (10%)
      - 3.3.5 用户侧实现 (已完成)
    - 3.4 编译和测试
      - 3.4.1 进入 Shell
      - 3.4.2 测试

## Lab 6: RV64 fork系统调用

---

### \*\*1 \*\*实验目的

---

了解 Linux 系统中 fork 系统调用的实现方法，并在之前实验基础上实现 fork 相关的系统调用（fork, exec, wait, exit）。

### 2 实验内容及要求

---

- 了解 Linux 系统中 fork 系统调用的实现方法
- 实现 fork, exec, wait, exit 系统调用

请各小组独立完成实验，任何抄袭行为都将使本次实验判为0分。

请跟随实验步骤完成实验并根据文档中的要求记录实验过程，最后删除文档末尾的附录部分，并命名为“学号1\_姓名1\_学号2\_姓名2\_lab6.pdf”，你的代码请打包并命名为“学号1\_姓名1\_学号2\_姓名2\_lab6”，文件上传至学在浙大平台。

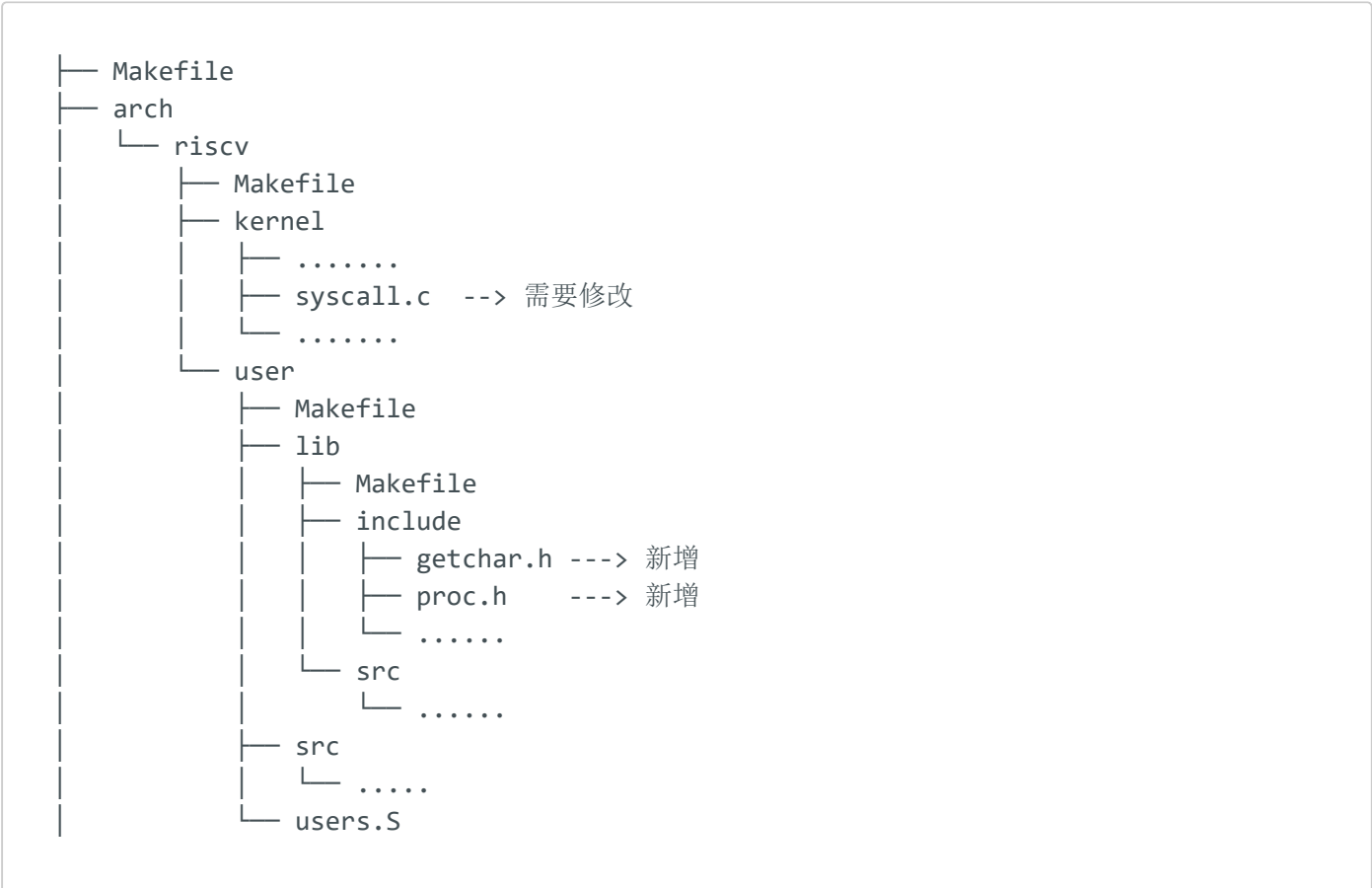
本实验以双人组队的方式进行，\*\*仅需一人提交实验，\*\*默认平均分配两人的得分（若原始打分为X，则可分配分数为2X，平均分配后每人得到X分）。如果有特殊情况请单独向助教反应，出示两人对于分数的钉钉聊天记录截图。单人完成实验者的得分为原始得分。

姓名	学号	分工	分配分数
陈申尧	3220101837	fork和exec	50%
安瑞和	3220106416	exit和wait	50%

### 3 实验步骤

#### 3.1 搭建实验环境

本实验提供的代码框架结构如图，你可以点击 [lab6.zip](#) 进行下载。首先，请下载相关代码，并移动至你所建立的本地映射文件夹中（即 lab0 中创建的 os\_experiment 文件夹）。



```
└─ include
   └─ .....
```

## 3.2 创建进程

### 3.2.1 批处理系统和进程

我们此前的实验已经为进程实现了许多应有的功能：上下文切换，地址空间，虚拟内存映射，动态内存分配管理等。惟一不足的地方在于，我们的进程总是在内核初始化的时候就定好了，无法在运行时新建进程。这事实上更像是批处理系统，每次导入一批任务，全部执行后再导入下一批任务。

因此，为了进程的灵活性，本实验将实现进程模型的几个重要系统调用：创建进程，执行新进程，等待进程结束，退出进程。有了这些系统调用，我们的 OS 进程管理就会更加灵活。

### 3.2.2 fork 系统调用

Linux 系统启动时会创建一些特定进程，负责特定功能。例如 0 号 Idle 进程，1 号 init 进程等。这些进程都是内核硬编码的进程【即执行代码，如何进入该进程都在源代码中已写好，而不是一般的从硬盘加载一个程序运行】，其他所有的进程都是通过 fork 系统调用实现的。

```
// 功能：当前进程 fork 出来一个子进程。
// 返回值：对于子进程返回 0，对于当前进程则返回子进程的 PID 。
// syscall ID: 220
int fork();
```

假设 A 进程发起了 fork 系统调用，那么之后就会进入 A 进程的内核栈，并开始执行 fork 函数，fork 函数会创建一个新的进程 B，并完整的拷贝 A 进程的用户栈，代码段，数据段，动态内存分配的空间等。而且我们知道 A 进程返回到用户态程序的时候，执行的一定是 ecall 指令的下一行，因此进程 B 同样可以做好设置，使得 B 进程返回用户态程序时也在相同的位置开始执行。这样，A，B 两个进程就好像是从 ecall 开始分裂出来的两个一模一样的进程。

注意，拷贝 A 进程的时候，我们用的是完全一致的虚拟地址空间，例如 A 进程 ecall 指令的下一行是 0x1000100。那么 B 进程回到用户态程序时也是回到 0x1000100 地址处。两者的虚拟地址完全一致，只是虚拟地址映射后的物理地址有所不同。【用户栈，

动态内存分配的空间还需要使用 `alloc_page` 等新建物理空间，并拷贝 A 进程对应物理地址下的数据，之后建立映射】

A 进程执行时候使用的地址都是虚拟地址，因此只要 B 进程保证虚拟地址和 A 进程完全一致，就确保了 A，B 进程完全一致。

当然，A 进程和 B 进程还是有一些区别的。我们把调用 `fork` 函数的进程称为父进程，创建出来的进程称为子进程，`fork` 函数执行的结果在父进程中是「子进程的PID」，在子进程中是「0」。这个效果很容易实现，我们只需要把内核栈上保存的 `a0` 寄存器的值修改即可。父进程的内核栈保存的 `a0` 寄存器值为「子进程的PID」，子进程则是「0」。这样当切换到子进程的时候，从内核栈上恢复寄存器后，就达到了 `fork` 函数返回不同值的效果了。

### 3.2.3 exec 系统调用

有了 `fork` 函数能确保我们可以创建进程，然而这些进程都执行相同的代码。这样的效果还远远不够，为此，还需要 `exec` 系统调用。`exec` 系统调用的功能就是将当前进程的地址空间清空并加载一个新的可执行程序。可以理解为回收空间，重新建立映射，然后「重头执行」。这样就可以执行别的进程了。

有了上述两个系统调用，我们就可以用如下代码启动一个新进程。

```
int pid = fork();
if (pid == 0) {
    // 子进程执行新程序
    exec("new program");
} else {
    // 父进程的代码
}
```

### 3.2.4 wait 系统调用

`wait` 系统调用是一种父子进程的通信手段，也是回收空间时的一种手段。`fork` 和 `exec` 的配合调用，会使得我们所有的进程都含有一个「父子关系」，即组织成一颗树的形式。有时候我们创建一个新进程出来，可能就是希望新进程完成某件事情并得到结果。`wait` 系统调用就起到了这个作用，他的作用是等待某一个子进程执行完成后再继续执行（阻塞版实现），然后回收该子进程资源。

为什么需要使用 `wait` 系统调用来回收资源？假设我们正在执行 B 进程（父进程是 A 进程），那么 B 进程执行完成后回到 B 进程的内核栈，开始回收资源，这个时候我们会发现无法回收 B 进程的内核栈，因为回收了就有可能导致其他的内核函数无法正常执行

（比如我们想要切换到别的进程去，这个时候内核栈都被回收了，就没有栈给函数用了）。因此我们需要把回收 B 进程内核栈的任务交给 A 进程去做。

### 3.2.5 exit 系统调用

`exit` 系统调用的功能就是退出程序，回收空间，并且返回一个退出代码。例如在 C 语言中有如下代码：

```
#include <stdlib.h>

int main() {
    if (...) {
        // 偶尔也会主动调用 exit 退出程序
        exit(1);
    }
    return 0;
}
```

一般来说，0代表正常返回，其他数字则代表异常返回，不同数字有不同含义等。

只看代码我们可以得知 `main` 函数结束时整个进程执行结束，然而我们需要在进程结束时回到内核态来回收该进程空间。为此，我们可以在用户态程序结束时候（隐式的）调用 `exit` 系统调用来实现这个效果。

在本实验中，我们为每个用户态程序开头加入了如下代码段：

```
_start:
    call main # a0 是 main 函数的返回值
    call exit # a0 作为 exit 函数的参数
```

这样就实现了 `main` 函数结束的时候调用 `exit` 函数的效果。而无需 `main` 函数中手动调用。

## 3.3 代码实现（100%）

### 3.3.1 其他代码调整 (已完成)

按照 3.2 节描述，我们需要修改部分代码来适配 `fork` 系统调用。

首先，我们需要在 `task_manager.c` 文件中的 `task_init` 函数中只为 0 号进程分配空间，并安排他执行 `test0.c` 的程序（我们写好的一个 `shell` 程序，这相当于我们提到的

硬编码程序)。

```
// initialize tasks, set member variables
void task_init(void) {
    // only init the first process
    struct task_struct* new_task = (struct task_struct*)(VIRTUAL_ADDR(alloc_page()));
    .....
    .....
    new_task->pid = 0;
    .....
    create_mapping(.....);
    .....
    .....
    printf("[PID = %d] Process Create Successfully!\n", task[0]->pid);
}
```

接下来，我们移动了内核态下处理 syscall 调用后，设置 a0, a1, sepc 寄存器的部分。如下

```
else if (cause == 0x8) {
    uint64_t* sp_ptr = (uint64_t*)(sp);
    uint64_t syscall_num = sp_ptr[11];
    uint64_t arg0 = sp_ptr[4], arg1 = sp_ptr[5], arg2 = sp_ptr[6], arg3 =
    sp_ptr[7], arg4 = sp_ptr[8], arg5 = sp_ptr[9];

    syscall(syscall_num, arg0, arg1, arg2, arg3, arg4, arg5, sp);

    // =====
    // | sp_ptr[4] = ret.a0; |
    // | sp_ptr[5] = ret.a1; |
    // | sp_ptr[16] += 4;    |
    // =====
}
```

我们把上述方框部分移动到了 syscall 函数内部，这样特定的系统调用就可以根据功能来修改这些寄存器的值了。

### 3.3.2 实现 fork 系统调用 (50%)

本实验中，我们需要实现第 220 号系统调用 fork。和之前一样，我们仍然在 syscall.c 文件中统一实现，其函数原型如下：

```
int fork(void);
```

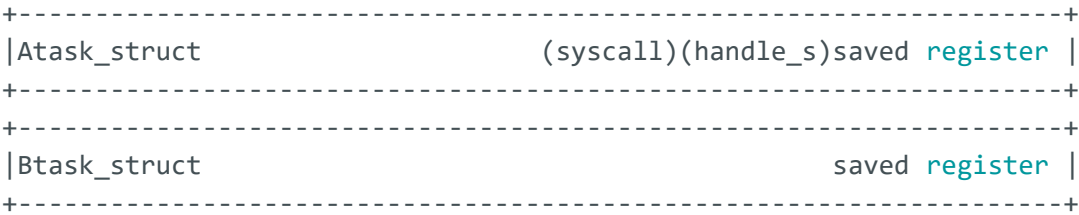
其执行流程如下：

1. 创建新进程，设置 counter, priority, pid 等内容（counter 可任意设置，但不需要不为 0, pid 需要保证不重复）。注意存放在我们的 task 数组中。
2. 创建页表，设置 task->satp
  1. 拷贝当前进程的用户进程代码地址，建立映射（当前进程运行的用户进程代码起始地址保存在 task->mm.user\_program\_start 处，用户进程代码占用一页大小）
  2. 创建内核地址映射（和 Lab 3 一致，包含 0xffff.... 的虚拟内存映射和等值映射）
  3. 创建 UART 的硬件地址映射（和 Lab 3 一致）
3. 分配一页用来做新进程的用户栈，拷贝当前进程的用户栈内容到新进程中，设置好新进程的用户栈指针(task->sscrath)。另外，将用户栈物理地址保存到 task->mm.user\_stack 中，方便之后的回收。
4. 拷贝动态内存管理部分 task->mm.vm，拷贝动态内存分配的空间内容，并且建立对应的映射。
5. 正确设置当前进程内核栈上的 a0, sepc 值
6. 拷贝内核栈内容（只需要拷贝 trap\_s 的内核栈）
7. 正确设置新进程内核栈上的 a0, sepc 值，以及 task->thread.sp， task->thread.ra。

为了让 fork 出来的新进程能够在被调度的时候正确执行。我们需要在第 6 步保存 trap\_s 的内核栈内容。（实际上就是保存了陷入系统调用时候用户态的所有寄存器内容，【注意这里是所有，和之前实验有区别】）之后，我们在第 7 步时，直接设置 ra 为 trap\_s 的后半段内容，也就是从恢复寄存器的部分开始 (trap\_s\_bottom)，这样就能保证重现调度到 fork 出来的程序的时候正常运行。

之后，fork 系统调用执行完毕，父进程继续执行。直到下次触发进程切换到子进程后，才真正的运行子进程。

A 是当前进程，B 进程是 fork 新建的进程



我们只复制这些部分，其他部分不用复制 sp, ra = trap\_s\_bottom

下一次调度到 B 进程的时候



```

+-----+
|Btask_struct                                saved register |
+-----+
                                     ↑
                                     sp, ra = trap_s_bottom

```

直接从 `trap_s_bottom` 开始，就可以恢复原先用户态进程下的所有寄存器了

请将你编写好的代码填入下方代码框中。

```

// syscall.c: 64

.....

case SYS_FORK: {
    // DONE:
    // 1. create new task and set counter, priority and pid (use our task
array)
    // 2. create root page table, set current process's satp
    // 2.1 copy current process's user program address, create mapping for
user program
    // 2.2 create mapping for kernel address
    // 2.3 create mapping for UART address
    // 3. create user stack, copy current process's user stack and save user
stack sp to new_task->sscratch
    // 4. copy mm struct and create mapping
    // 5. set current process a0 = new task pid, sepc += 4
    // 6. copy kernel stack (only need trap_s' stack)
    // 7. set new process a0 = 0, and ra = trap_s_bottom, sp = register number
* 8

    int i = 0;
    for (i = 0; i < NR_TASKS; i++)
        if (!task[i])
            break;
    if (i >= NR_TASKS)
        break;

    struct task_struct* new_task = (struct task_struct*)
(VIRTUAL_ADDR(alloc_page()));
    new_task->state = current->state;
    new_task->counter = current->counter;
    new_task->priority = current->priority;
    new_task->blocked = current->blocked;
    new_task->pid = i;
    task[i] = new_task;
    task[i]->thread.sp = (uint64_t)task[i] + PAGE_SIZE; // 内核栈的栈底

    uint64_t task_addr = current->mm.user_program_start;
    // DONE: 完成用户栈的分配，并创建页表项，将用户栈映射到实际的物理地址
    // 1. 为用户栈分配物理页面，使用alloc_page函数
    // 2. 为用户进程分配根页表，使用alloc_page函数
    // 3. 将task[i]->sscratch指定为虚拟空间下的栈地址，即0x1001000 + PAGE_SIZE（注
意栈是从高地址到低地址使用的）

```



```

// 4. 正确设置task[i]->satp, 注意设置ASID
// 5. 将用户栈映射到实际的物理地址, 使用create_mapping函数
// 6. 将用户程序映射到虚拟地址空间, 使用create_mapping函数
// 7. 将虚拟地址 0xffffffc000000000 开始的 16 MB 空间映射到起始物理地址为
0x80000000 的 16MB 地址空间, 注意此时 &rodata_start、... 得到的是虚拟地址还是物理地址?
我们需要的是什么地址?
// 8. 对内核起始地址 0x80000000 的16MB空间做等值映射 (将虚拟地址 0x80000000 开
始的 16 MB 空间映射到起始物理地址为 0x80000000 的 16MB 空间), PTE_V | PTE_R | PTE_W |
PTE_X 为映射的读写权限。
// 9. 修改对内核空间不同 section 所在页属性的设置, 完成对不同section的保护, 其中
text段的权限为 r-x, rodata 段为 r--, 其他段为 rw-, 注意上述两个映射都需要做保护。
// 10. 将必要的硬件地址 (如 0x10000000 为起始地址的 UART ) 进行等值映射 ( 可以映
射连续 1MB 大小 ), 无偏移, PTE_V | PTE_R 为映射的读写权限
uint64_t physical_stack = alloc_page();
uint64_t root_page_table = alloc_page();
task[i]->mm.user_stack = physical_stack;
task[i]->mm.user_program_start = task_addr;
task[i]->sscratch = (uint64_t)0x1001000 + PAGE_SIZE;
task[i]->satp = root_page_table >> 12 | 0x8000000000000000 | (((uint64_t)
(new_task->pid)) << 44);
create_mapping((uint64_t*)root_page_table, 0x1001000, physical_stack,
PAGE_SIZE, PTE_V | PTE_R | PTE_W | PTE_U);
create_mapping((uint64_t*)root_page_table, 0x1000000, task_addr, PAGE_SIZE,
PTE_V | PTE_R | PTE_X | PTE_U | PTE_W);
// 调用 create_mapping 函数将虚拟地址 0xffffffc000000000 开始的 16 MB 空间映射
到起始物理地址为 0x80000000 的 16MB 空间
create_mapping((uint64_t*)root_page_table, 0xffffffc000000000, 0x80000000,
16 * 1024 * 1024, PTE_V | PTE_R | PTE_W | PTE_X);
// 修改对内核空间不同 section 所在页属性的设置, 完成对不同section的保护, 其中
text段的权限为 r-x, rodata 段为 r--, 其他段为 rw-。
create_mapping((uint64_t*)root_page_table, 0xffffffc000000000, 0x80000000,
PHYSICAL_ADDR((uint64_t)&rodata_start) - 0x80000000, PTE_V | PTE_R | PTE_X);
create_mapping((uint64_t*)root_page_table, (uint64_t)&rodata_start,
PHYSICAL_ADDR((uint64_t)&rodata_start), (uint64_t)&data_start -
(uint64_t)&rodata_start, PTE_V | PTE_R);
create_mapping((uint64_t*)root_page_table, (uint64_t)&data_start,
PHYSICAL_ADDR((uint64_t)&data_start), (uint64_t)&end - (uint64_t)&data_start,
PTE_V | PTE_R | PTE_W);
// 对内核起始地址 0x80000000 的16MB空间做等值映射 (将虚拟地址 0x80000000 开始的
16 MB 空间映射到起始物理地址为 0x80000000 的 16MB 空间)
create_mapping((uint64_t*)root_page_table, 0x80000000, 0x80000000, 16 *
1024 * 1024, PTE_V | PTE_R | PTE_W | PTE_X);
// 修改对内核空间不同 section 所在页属性的设置, 完成对不同section的保护, 其中
text段的权限为 r-x, rodata 段为 r--, 其他段为 rw-。
create_mapping((uint64_t*)root_page_table, 0x80000000, 0x80000000,
PHYSICAL_ADDR((uint64_t)&rodata_start) - 0x80000000, PTE_V | PTE_R | PTE_X);
create_mapping((uint64_t*)root_page_table,
PHYSICAL_ADDR((uint64_t)&rodata_start), PHYSICAL_ADDR((uint64_t)&rodata_start),
(uint64_t)&data_start - (uint64_t)&rodata_start, PTE_V | PTE_R);
create_mapping((uint64_t*)root_page_table,
PHYSICAL_ADDR((uint64_t)&data_start), PHYSICAL_ADDR((uint64_t)&data_start),
(uint64_t)&end - (uint64_t)&data_start, PTE_V | PTE_R | PTE_W);
// 将必要的硬件地址 (如 0x10000000 为起始地址的 UART ) 进行等值映射 ( 可以映射连
续 1MB 大小 ), 无偏移, 3 为映射的读写权限
create_mapping((uint64_t*)root_page_table, 0x10000000, 0x10000000, 1 * 1024
* 1024, PTE_V | PTE_R | PTE_W | PTE_X);

```

```

// 复制用户栈
memcpy((void*)physical_stack, (void*)current->mm.user_stack, PAGE_SIZE);
task[i]->sscratch = read_csr(sscratch); // 父进程用户栈

// 复制mm
task[i]->mm.vm = kmalloc(sizeof(struct vm_area_struct));
INIT_LIST_HEAD(&(task[i]->mm.vm->vm_list));
uint64_t parent_page_table = (current->satp & ((1ULL << 44) - 1)) << 12;
struct vm_area_struct* vma;
list_for_each_entry(vma, &current->mm.vm->vm_list, vm_list) {
    struct vm_area_struct* newNode = kmalloc(sizeof(struct
vm_area_struct));
    memcpy(newNode, vma, sizeof(struct vm_area_struct));
    list_add(&(newNode->vm_list), &task[i]->mm.vm->vm_list);
    if (vma->mapped) {
        uint64_t pa = alloc_pages((vma->vm_end - vma->vm_start) /
PAGE_SIZE);
        uint64_t parent_pa = (get_pte(parent_page_table, vma->vm_start) >>
10) << 12;
        memcpy((void*)pa, (void*)parent_pa, vma->vm_end - vma->vm_start);
        create_mapping((uint64_t*)root_page_table, vma->vm_start, pa, vma-
>vm_end - vma->vm_start, vma->vm_flags);
    }
}

sp_ptr[4] = task[i]->pid;
sp_ptr[16] += 4;

task[i]->thread.sp -= 8 * 31;
memcpy((void*)task[i]->thread.sp, (void*)sp_ptr, 8 * 31);
((uint64_t*)task[i]->thread.sp)[4] = 0;
task[i]->thread.ra = (uint64_t)&trap_s_bottom;

// printf("[PID = %d] Process Create Successfully!\n", task[i]->pid);

break;
}

.....

```

### 3.3.2 实现 exec 系统调用 (20%)

接下来，我们需要实现 191 号系统调用 `exec`，该系统调用接收一个参数 `arg0`，表示的是需要执行的进程的名称。可以使用已经写好的 `get_program_address(arg0)` 函数获得要执行的新用户进程的代码地址。

`exec` 系统调用的函数原型如下：

```
void exec(const char * name);
```

其执行流程如下：

1. 清空当前进程的动态内存分配区域 `task->mm.vm`，包括映射和空间回收
2. 重置用户栈指针（`sscratch`）【我们复用用户栈】
3. 重新设置用户进程代码的虚拟地址映射【使用 `get_program_address` 获取新的用户进程代码的首地址】（注意修改了映射后需要使用 `sfence.vma` 刷新 TLB）
4. 重新设置 `sepc`

请将你编写好的代码填入下方代码框中。

```
// syscall.c: 79

.....

case SYS_EXEC: {
    // DONE:
    // 1. free current process vm_area_struct and it's mapping area
    // 2. reset user stack, user_program_start
    // 3. create mapping for new user program address
    // 4. set sepc = 0x1000000
    // 5. refresh TLB

    uint64_t root_page_table = (current->satp & ((1ULL << 44) - 1)) << 12;
    struct vm_area_struct* vma;
    list_for_each_entry(vma, &current->mm.vm->vm_list, vm_list) {
        if (vma->mapped == 1) {
            uint64_t pte = get_pte(root_page_table, vma->vm_start);
            free_pages((pte >> 10) << 12);
        }
        create_mapping(root_page_table, vma->vm_start, 0, (vma->vm_end - vma->vm_start), 0);
        list_del(&(vma->vm_list));
        kfree(vma);
    }

    // current->sscratch = (uint64_t)0x1001000 + PAGE_SIZE;
    write_csr(sscratch, (uint64_t)0x1001000 + PAGE_SIZE);
    uint64_t task_addr = get_program_address((char*)arg0);
    current->mm.user_program_start = task_addr;

    create_mapping(root_page_table, 0x1000000, task_addr, PAGE_SIZE, PTE_V |
PTE_R | PTE_X | PTE_U | PTE_W);

    sp_ptr[16] = 0x1000000;

    asm volatile ("sfence.vma");

    break;
}

.....
```

### 3.3.3 实现 wait 系统调用 (20%)

然后，我们需要实现 247 号系统调用 `wait`，该系统调用接收一个参数 `arg0`，表示的是要等待执行完的子进程的 `pid`。

`wait` 系统调用的函数原型如下：

```
void wait(int pid);
```

其执行流程如下：

1. 找到要等待的进程的 `pid`。
2. 如果没有，正确设置 `sepc` 的值并返回。
  1. 如果找到，但是 `counter = 0`，说明该进程执行完毕，回收他的内核栈。
3. 如果找到，则需要修改当前进程的优先级，使得要等待的子进程优先被调度，之后调用 `schedule` 函数主动切换进程。【之后再次被调度回来的时候需要循环检查，直到进入 2】

请将你编写好的代码填入下方代码框中。

```
// syscall.c: 98

.....

case SYS_WAIT: {
    // DONE:
    // 1. find the process which pid == arg0
    // 2. if not find
    //    2.1. sepc += 4, return
    // 3. if find and counter = 0
    //    3.1. free it's kernel stack and page table
    // 4. if find and counter != 0
    //    4.1. change current process's priority
    //    4.2. call schedule to run other process
    //    4.3. goto 1. check again

    if (task[arg0]) {
        while (task[arg0]->counter > 0) {
            current->priority = task[arg0]->priority + 1;
            schedule();
        }
        uint64_t* root_page_table = (uint64_t*)((task[arg0]->satp & ((1ULL <<
44) - 1)) << 12);
        uint64_t kernal_stack = (get_pte(root_page_table, task[arg0]) >> 10) <<
12;

        free_pages(root_page_table);
        free_pages(kernal_stack);
    }
}
```

```

        task[arg0] = NULL;
    }

    sp_ptr[16] += 4;

    break;
}

.....

```

### 3.3.4 实现 exit 系统调用 (10%)

最后，我们需要实现 60 号系统调用 `exit`，该系统调用接收一个参数 `arg0`，表示的是进程退出的返回值。

`exit` 系统调用的函数原型如下：

```
void exit(int code);
```

其执行流程如下：

1. 清空当前进程的动态内存分配区域 `task->mm.vm`，包括映射和空间回收。
2. 清空用户栈。
3. 清空页表。
4. 清空当前 `task_struct` 信息，设置 `counter = 0`，避免再次调度。
5. 调用 `schedule`，执行别的进程。

注意：这里不能直接回收内核栈，否则 `schedule` 调用就无栈可用了。这个问题在 3.2.4 节也有描述。其内核栈回收应该在 `wait` 系统调用时实现。

请将你编写好的代码填入下方代码框中。

```

// syscall.c: 88

.....

case SYS_EXIT: {
    // DONE:
    // 1. free current process vm_area_struct and it's mapping area
    // 2. free user stack
    // 3. clear current task, set current task->counter = 0
    // 4. call schedule

    uint64_t* root_page_table = (uint64_t*)((current->satp & ((1ULL << 44) -
1)) << 12);

```

```

struct vm_area_struct* vma;
list_for_each_entry(vma, &current->mm.vm->vm_list, vm_list) {
    if (vma->mapped == 1) {
        uint64_t pte = get_pte(root_page_table, vma->vm_start);
        free_pages((pte >> 10) << 12);
    }
    create_mapping(root_page_table, vma->vm_start, 0, (vma->vm_end - vma->vm_start), 0);
    list_del(&(vma->vm_list));
    kfree(vma);
}
// head node
kfree(current->mm.vm);
current->mm.vm = NULL;

free_pages(current->mm.user_stack);
current->mm.user_stack = 0;

// current->state = TASK_DEAD;
current->counter = 0;

schedule();

break;
}

.....

```

### 3.3.5 用户侧实现（已完成）

用户侧需要和之前的实验一样，写好对应的用户侧的系统调用。该部分比较简单，已完成。

## 3.4 编译和测试

### 3.4.1 进入 Shell

本次实验中，我们为大家提供了一个 `getchar` 系统调用，其功能是获取用户输入的字符，如果获取失败则返回 -1。基于此，我们在 `test1.c` 中实现了一个简单的 shell 程序，用于测试代码正确性。该 shell 程序仅包含以下指令：

- `ls`：展示当前可以执行的进程名称（测试程序仅实现了 `hello`, `malloc`, `print`, `guess`）
- `hello`：会输出 `hello world`，是最简单的 Debug 用程序。
- `malloc`：包含了一次 `mmap` 和 `munmap` 调用。
- `print`：包含了一次 `fork` 嵌套调用。

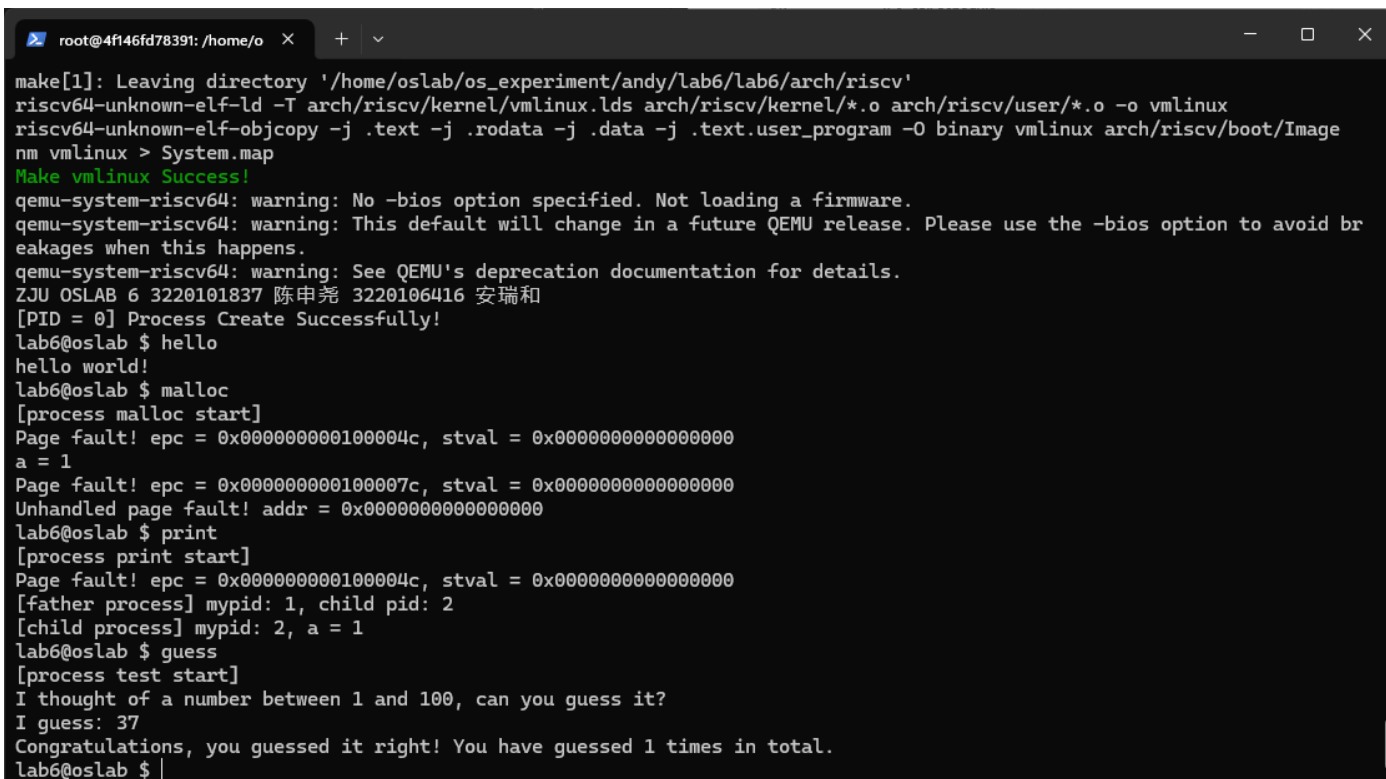
- `guess` : 一个简单的猜数字程序。

OS 还有一个很重要的功能，就是作为硬件和软件的桥梁，为上层应用屏蔽下层硬件实现。之前实验中用到的 `printf` 或是本次提供的 `getchar` 都是针对特定硬件来实现的。在 QEMU 模拟器中，我们使用的是 UART16550 等相关串口来实现输入输出。感兴趣的同学可自行查看相关实现。这种与特定硬件交互的程序，一般称为硬件设备驱动。编写驱动往往需要严格按照硬件设备设定的规范，同时还需要配合中断处理程序做操作等，编写琐碎，Debug 困难。因此这里直接提供了一份（只能说能用的）实现，仅供参考。

### 3.4.2 测试

仿照 Lab 5 进行编译及调试，对 `main.c` 做修改，确保输出你的学号与姓名。在项目最外层输入 `make run` 命令调用 Makefile 文件完成整个工程的编译及执行。

请在此附上你的调用4个程序的实验结果截图。



```
root@4f146fd78391: /home/o X + v
make[1]: Leaving directory '/home/oslab/os_experiment/andy/lab6/lab6/arch/riscv'
riscv64-unknown-elf-ld -T arch/riscv/kernel/vmlinux.lds arch/riscv/kernel/*.o arch/riscv/user/*.o -o vmlinux
riscv64-unknown-elf-objcopy -j .text -j .rodata -j .data -j .text.user_program -O binary vmlinux arch/riscv/boot/Image
nm vmlinux > System.map
Make vmlinux Success!
qemu-system-riscv64: warning: No -bios option specified. Not loading a firmware.
qemu-system-riscv64: warning: This default will change in a future QEMU release. Please use the -bios option to avoid br
eakages when this happens.
qemu-system-riscv64: warning: See QEMU's deprecation documentation for details.
ZJU OSLAB 6 3220101837 陈申尧 3220106416 安瑞和
[PID = 0] Process Create Successfully!
lab6@oslab $ hello
hello world!
lab6@oslab $ malloc
[process malloc start]
Page fault! epc = 0x00000000100004c, stval = 0x0000000000000000
a = 1
Page fault! epc = 0x00000000100007c, stval = 0x0000000000000000
Unhandled page fault! addr = 0x0000000000000000
lab6@oslab $ print
[process print start]
Page fault! epc = 0x00000000100004c, stval = 0x0000000000000000
[father process] mypid: 1, child pid: 2
[child process] mypid: 2, a = 1
lab6@oslab $ guess
[process test start]
I thought of a number between 1 and 100, can you guess it?
I guess: 37
Congratulations, you guessed it right! You have guessed 1 times in total.
lab6@oslab $
```