

验收成绩	报告成绩	总评成绩

武汉大学计算机学院

本科生实验报告

操作系统内核实验

专 业 名 称 ： 计算机科学与技术

课 程 名 称 ： 操作系统课程设计

指 导 教 师 ： 曾平 副教授

学 生 学 号 ：

学 生 姓 名 ：

二〇二三年六月

郑 重 声 明

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名：_____ 日期：_____

摘 要

操作系统内核实验要求通过完善一个运行在 RISC-V 体系结构上的操作系统内核，来加深对操作系统原理的理解。整个实验在类 Linux 环境下进行。需要构建一个类 Linux 环境，由于我们并没有 RISC-V 的硬件，所以要安装一个 spike 模拟器来模拟 RISC-V 硬件环境。

本次实验有三个部分，分别涉及系统调用、内存管理和进程管理。每个部分又有若干基础题目和挑战题目。每个部分的实验相互是有依赖关系的，后一个部分的实验题要在前一个部分的实验题做成功的基础上才能进行，内存管理要在系统调用正确实现的基础上才能开始，进程调度需要在内存管理成功实现的基础上才能运行。

关键词：操作系统内核；RISC-V；spike；系统调用

目 录

1 环境安装与配置

1.1 实验内容.....	5
1.2 实验工具与平台.....	5
1.3 实验原理.....	6

2 系统调用、异常与中断

2.1 实验内容.....	7
2.2 代码实验与分析.....	7
2.3 挑战实验.....	9
2.4 实验总结.....	13

3 内存管理

3.1 实验内容.....	14
3.2 代码实验与分析.....	14
3.3 挑战实验.....	17
3.4 实验总结.....	19

4 进程管理

4.1 实验内容.....	20
4.2 代码实验与分析.....	20
4.3 挑战实验.....	22
4.4 实验总结.....	24

5 总结与收获.....25

参考文献	26
------------	----

附录	27
----------	----

1 环境安装与配置

1.1 实验内容

本次实验需要在类 Linux 环境下进行，因此使用 macOS12.5 进行实验。相关工具有 RISC-V 交叉编译器及附带的主机编译器、工具等，以及 spike 模拟器。

本实验共分为三组，每组包含基础实验和挑战实验。

第一组实验重点涉及系统调用、异常和外部中断的知识；第二组实验重点涉及主存管理和虚拟内存方面的知识；第三组实验重点涉及进程管理方面的知识。部分实验间存在依赖性，后一个实验依赖于前一个实验的答案。每组的挑战实验只依赖于每组的最后一个实验。后续组的基础实验不依赖于前一组的挑战实验。

其中“系统调用、异常和外部中断”主要是对内核部分的分析和修改；“主存管理和虚拟内存”需要对地址空间和映射有一定的了解；“进程管理”涉及运行过程中的管理与调度。

1.2 实验工具与平台

1.2.1 RISC-V 交叉编译器

RISC-V 交叉编译器是与 Linux/macOS 自带的 GCC 编译器类似的一套工具软件集合，但 x86_64 或 arm 平台上 Linux 自带的 GCC 编译器会将源代码编译、链接成为适合在 x86_64 或 arm 平台上运行的二进制代码 (称为 native code)，而 RISC-V 交叉编译器则会将源代码编译、链接成为在 RISC-V 平台上运行的代码。后者 (RISC-V 交叉编译器生成的二进制代码) 是无法在 x86_64 或 arm 平台 (即 x86_64 或 arm 架构的 Ubuntu 环境下) 直接运行的，它的运行需要模拟器 (如 spike 模拟器) 的支持。一般情况下，我们称 x86_64 或 arm 架构的 Ubuntu 环境为 host，而在 host 上执行 spike 后所虚拟出来的 RISC-V 环境，则被称为 target。

本实验在 arm 架构的 Apple 芯片上进行，所以 host 平台为 arm 架构。

命令：riscv64-unknown-elf-gcc 可以预处理、编译、汇编已有的 C 语言代码；

命令：riscv64-unknown-elf-objdump 可以显示 headers 的内容；

命令：riscv64-unknown-elf-as 可以编译汇编源文件到目标文件；

1.2.2 Spike

命令格式：spike [host options] [target options]

常见参数：-m Provide MiB of target memory [default 2048] (提供 MiB 内存)

`--isa= RISC-V ISA string [default RV64IMAFDC]` (设置 ISA)

`spike` 是 RISC-V 仿真器,我们将需要使用它结合 `pke` 运行我们的二进制程序。

`spike` 默认提供的内存为 2048MiB, 使用 `-m` 选项可以指定内存大小 (单位是 MiB)。RISC-V 是一个可扩展指令集, `spike` 默认支持的 ISA 为 RV64IMAFDC, 可以通过 `--isa` 选项设置模拟出来的机器的 ISA。

1.2.3 git

`git` 是一个开源的分布式版本控制系统, 并可以对已存在的文件进行版本控制, 对远程仓库进行克隆。在本实验中, 非常方便地对不同任务之间的依赖进行良好的管理。

`Git` 是基于 `Linux` 内核开发的版本控制工具。与常用的版本控制工具 `CVS`, `Subversion` 等不同, 它采用了分布式版本库的方式, 不必服务器端软件支持, 使源代码的发布和交流极其方便。 `Git` 的速度很快, 这对于诸如 `Linux kernel` 这样的大项目来说自然很重要。 `Git` 最为出色的是它的合并跟踪 (merge tracing) 能力。[1]

1.3 实验原理

1.3.1 RISC-V 程序的编译和链接

按照前述 `riscv64-unknown-elf-gcc` 的 `-c` 命令, 使用交叉编译器对以上程序进行编译, 并在当前目录得到 `.o` 文件。这一文件中包含可浮动代码, 意味着该 ELF 文件中的符号并无指定的逻辑地址。最后, 用 `-o` 指令对生成的目标文件进行链接。

对比于 `.o` 文件, 可以发现 `helloworld` 文件是可执行文件而不是可浮动代码, 也就是说通过链接, 已经给源代码中的符号指定好了逻辑地址。

1.3.2 代理内核的构造

位于根目录的 `Makefile` 文件, 将 `user/`目录下通过编译出来的 `.o` 文件与 `util` 中编译和链接出来的静态库文件一起链接, 生成采用 RISC-V 指令集的可执行文件。同时, 链接过程采用 `user/user.ld` 脚本以指定生成的可执行文件中的符号所对应的逻辑地址。

然后将编译 `kernel` 目录下的源文件所得到的 `.o` 文件与 `.a` 进行链接, 并最终生成代理内核 `riscv-pke`。此时就可以使用 `spike` 命令来在内核中运行程序了。

2 系统调用、异常与中断

2.1 实验内容

2.1.1 lab1_1 系统调用

给定初始代码，代码中没有调用 `syscall`，需要通过调用 `syscall` 完成输出代码的实现。

2.1.2 lab1_2 异常处理

(在用户 U 模式下执行的) 应用企图执行 RISC-V 的特权指令 `csrw sscratch, 0`。该指令会修改 S 模式的栈指针，如果允许该指令的执行，执行的结果可能会导致系统崩溃。因此需要制止并输出“`Illegal instruction`”。

2.1.3 lab1_3 (外部)中断

实验中给出的 PKE 操作系统内核，在时钟中断部分并未完全做好，导致 (模拟) RISC-V 机器碰到第一个时钟中断后就会出现崩溃。要求完成 PKE 操作系统内核未完成的时钟中断处理过程，使得它能够完整地处理时钟中断。

2.1.4 lab1_challenge1 打印用户程序调用栈

通过修改 PKE 内核，来实现从给定应用到预期输出的转换。

对于 `print_backtrace()` 函数的实现要求：应用程序调用 `print_backtrace()` 时，能够通过控制输入的参数控制回溯的层数。

2.1.5 lab1_challenge2 打印异常代码行

通过修改 PKE 内核 (包括 `machine` 文件夹下的代码)，使得用户程序在发生异常时，内核能够输出触发异常的用户程序的源文件名和对应代码行。

2.2 代码实现与分析

用户程序在 U 模式下通过调用 `do_user_call`，来让内核在 S 模式下进行相应的功能，在这之前需要将相应数据存放在 `a0-a7` 寄存器中。

```
do_syscall(tf->regs.a0, tf->regs.a1, tf->regs.a2, tf->regs.a3,  
           tf->regs.a4, tf->regs.a5, tf->regs.a6, tf->regs.a7);
```

然后根据 `a0` 的属性，内核将执行相应的函数。

从代码中可以看到，`trap` 的入口处理函数首先将“进程” (即应用的运行现场) 进行保存；接下来将 `a0` 寄存器中的系统调用号保存到内核堆栈，再将 `p->trapframe->kernel_sp` 指向的为应用进程分配的内核栈设置到 `sp` 寄存器，完成

了栈的切换。最后为了还原保存起来的现场，栈需要切换回用户进程自带的用户栈。

在 `handle_syscall()` 函数的第 19 行，有一个 `tf->epc += 4;` 语句，因为系统调用中断之后需要返回 `PC+4` 执行下一条指令。`pke` 操作系统内核可以通过调用 `syscall` 时传入的地址得到应用程序中 “hello world!” 字符串的地址。

当遇到异常时，内核会保存现场并调用 `handle_mtrap()`，此时 `mcause` 中保存了异常的原因，此时检测到不合法指令，需要调用 `handle_illegal_instruction()`，这样就可以正常报错 `panic` 了。

当遇到时钟中断时，每遇到一次就将 `g_ticks` 加一处理。处理完中断后，`SIP` 寄存器中的 `SIP_SSP` 位仍然为 1（由 M 态的中断处理函数设置），如果该位持续为 1，会导致模拟的 RISC-V 机器始终处于中断状态。所以需要在 `handle_mtimer_trap()` 中对 `SIP` 的 `SIP_SSP` 位清零，以保证下次再发生时钟中断时，M 态的函数将该位置一会导致 S 模式的下一次中断。

如果采用死循环，并不会导致死机，因为在执行过程中没有额外资源的消耗。

```
Apple > ~/W/操/课/riscv-pke > git P lab1_1_syscall +1 ?8
spike ./obj/riscv-pke ./obj/app_helloworld
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
Application: ./obj/app_helloworld
Application program entry point (virtual address): 0x0000000081000000
Switch to user mode...
Hello world!
User exit with code:0.
System is shutting down with exit code 0.
```

图 1 lab1_1 结果

```
Apple > ~/W/操/课/riscv-pke > git P lab1_2_exception +3 ?8
spike ./obj/riscv-pke ./obj/app_illegal_instruction
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
Application: ./obj/app_illegal_instruction
Application program entry point (virtual address): 0x0000000081000000
Switch to user mode...
Going to hack the system by running privilege instructions.
Illegal instruction!
System is shutting down with exit code -1.
```

图 2 lab1_2 结果


```
Apple > ~/W/操/课/riscv-pke > gh P lab1_3_irq +5 ?8
spike ./obj/riscv-pke ./obj/app_long_loop
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
Application: ./obj/app_long_loop
Application program entry point (virtual address): 0x0000000081000000
Switch to user mode...
Hello world!
wait 0
wait 5000000
wait 10000000
Ticks 0
wait 15000000
wait 20000000
Ticks 1
wait 25000000
wait 30000000
wait 35000000
Ticks 2
wait 40000000
wait 45000000
Ticks 3
wait 50000000
wait 55000000
wait 60000000
Ticks 4
wait 65000000
wait 70000000
Ticks 5
```

图 3 lab1_3 结果

2.3 挑战实验

本实验的挑战实验需要查询 elf 中 section 的 header 和相应符号表，并正确地将其读取。使用关键的 `elf_fpread` 函数，将所需要的部分读取出来，并把地址的映射位置找到，就可以完成挑战实验。

```
elf_fpread(ctx, (void *)&sh, sizeof(sh),
           ctx->ehdr.shoff+ctx->ehdr.shstrndx*sizeof(elf_shdr))
```

这段代码表示将 `shstr` 所在的位置计算出来并把内容读到 `sh` 上，得到符号表与字符串表的信息以获取名字列表。接下来对于每个 section，查询它的 `name` 是否为 `.symtab` 或 `.strtab`，如果是，则将其取出并放入返回值。

取出后，则到用户栈中依次找到所有函数名，对于当前函数，如果它的返回值落在其中一个函数开始到它结束之间，那就说明是由这个函数调用的它。因此需要进行一部分判断与调整。每查询完一个函数就需要将栈指针回退两个位置，

找到下一个函数。

```
while(a1--)  
{  
    uint64 ra=((uint64*)fp-1);  
    if(!ra) break;  
    for(i=0;i<symnum;++i)  
        if(sym[i].st_value<ra&&sym[i].st_value>ra-sym[i].st_size)  
            sprintf("%s\n",&namelist[sym[i].st_name]);  
}
```

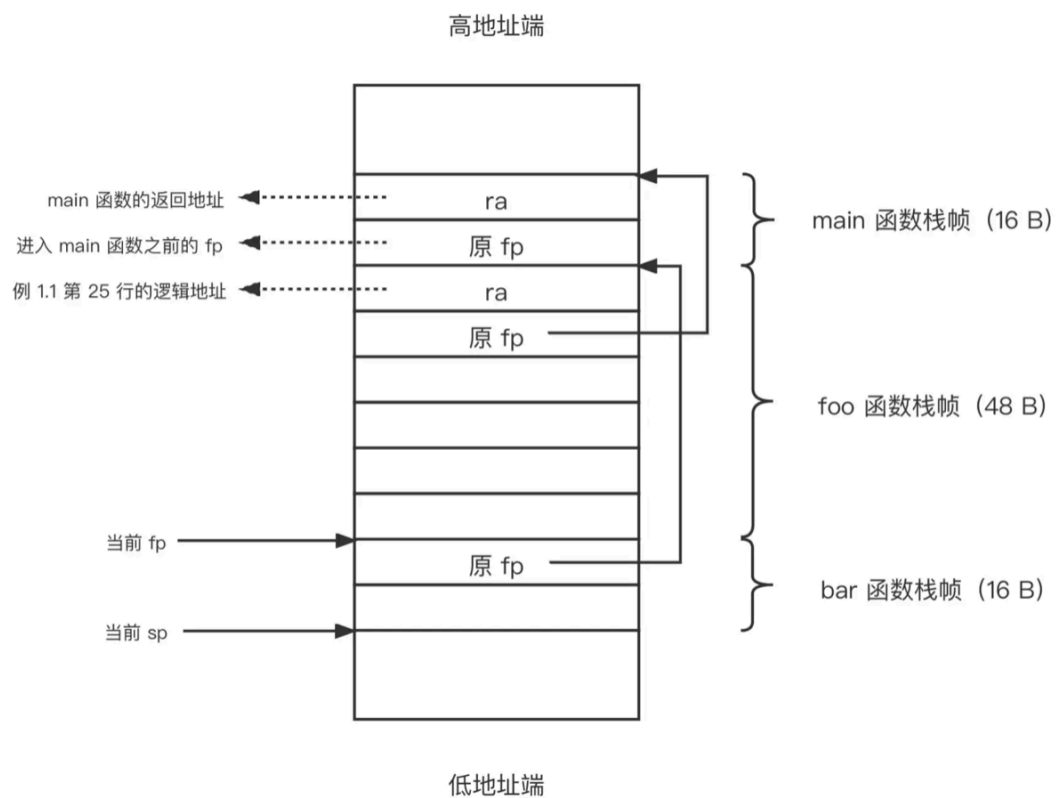


图 4 函数调用的栈帧

```

[ Apple > ~ / W / 操 / 课 / riscv-pke > P lab1_challenge1_backtrace +7 ?8
spike ./obj/riscv-pke ./obj/app_print_backtrace
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
Application: ./obj/app_print_backtrace
Application program entry point (virtual address): 0x00000000810000a2
Switch to user mode...
back trace the user app in the following:
f8
f7
f6
f5
f4
fff3
f2
EndUser exit with code:0.
System is shutting down with exit code 0.

```

图 5 lab1_challenge1 结果

打印异常代码时，也只需要和前面的过程一样，读出 debug_line 段，进行相应分析，并按照说明中提供的逻辑解析 process 结构体的 dir、file、line 三个指针。dir 是目录，file 是文件，line 是行号。当定位到出错的位置时，可以通过行号来对输出进行控制。

最终使用 spike_file_open 读入代码，用\n 来判断行的数量即可。

另外，使用静态数组来存储 debug_line 段数据，需要足够大的数组，经尝试，1000 的常量数组并不足够，因此考虑把 debug_line 直接放在程序所有需映射的段数据之后，每次记录所有数据的右边界，取最大值作为程序所有段数据的边界。

```

for (i = 0, off = ctx->ehdr.phoff; i < ctx->ehdr.phnum; i++, off
+= sizeof(ph_addr)) {
    if (elf_fpread(ctx, dest, ph_addr.memsz, ph_addr.off) !=
ph_addr.memsz)
        return EL_EIO;
    bound = bound > ph_addr.vaddr+ph_addr.memsz ? bound :
ph_addr.vaddr+ph_addr.memsz;
}

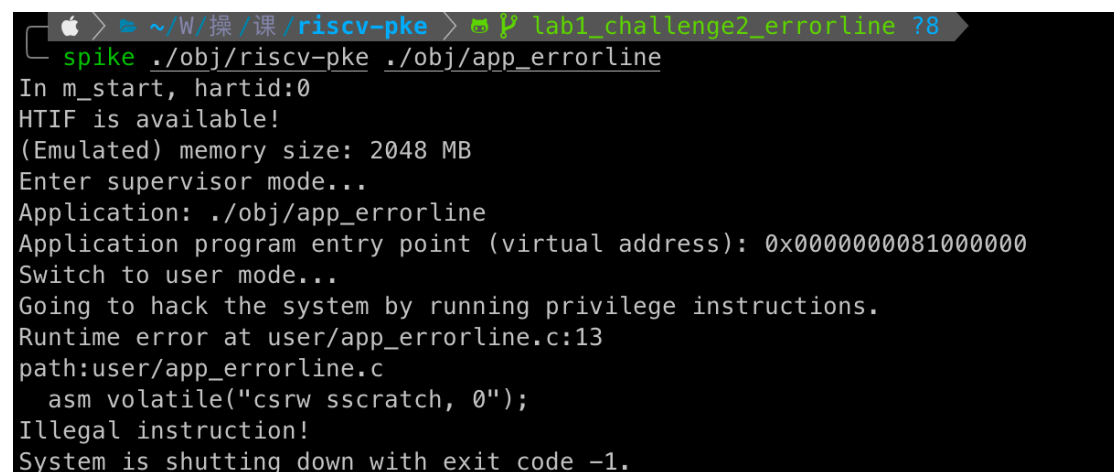
```

如上计算出 bound 之后，判断 current->line[i].addr==mepc，则代表当前行就是出错指令所在的位置。输出相应的文件名和行号即可。

.shstrtab 节中保存着以'\0'分割的所有节的名称的字符串。我们尝试测试打印

出所有节的名称。某一个节的名称的起始地址计算方式如下：.shstrtab 节头表.offset+该节的节头表.name。获取到起始地址之后，逐个字节往下获取，直到'\0'为止。

```
if(current->line[i].addr==mepc)
{
    uint64 file_index=current->line[i].file;
    uint64 dir=current->file[file_index].dir;
    sprintf("Runtime error at %s/%s:%ld\n", current->dir[dir],
current->file[file_index].file,current->line[i].line);
...
    for(cur=0;;++cur)
    {
        spike_file_pread(f, &code, 1, cur);
        if(line==current->line[i].line)
        {
            sprintf("%c",code);
            if(code=='\n')
                break;
        }
        if(code=='\n')
            ++line;
    }
}
```



```
Apple > ~/W/操/课/riscv-pke > lab1_challenge2_errorline ?8
spike ./obj/riscv-pke ./obj/app_errorline
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
Application: ./obj/app_errorline
Application program entry point (virtual address): 0x0000000081000000
Switch to user mode...
Going to hack the system by running privilege instructions.
Runtime error at user/app_errorline.c:13
path:user/app_errorline.c
    asm volatile("csrw sscratch, 0");
Illegal instruction!
System is shutting down with exit code -1.
```

图 6 lab1_challenge2 结果

2.4 实验总结

本实验的重点在于了解内核调用的原理，并知道程序段是如何在内核中存储的。此外 `fread` 的运用充分考察了对地址和内存的掌握，增强了对高级语言中段存储结构的理解。难点主要是 `elf` 的读取和地址转化。理解如何读取和存储符号表是一件比较困难的事情，同时还要理解地址和栈指针。

`ctx` 为 `elf` 文件的上下文指针，即 `context` 的缩写。

`mepc`: Machine Exception PC。指向发生异常的那条指令的地址。

另外，`p->trapframe->epc = elfloader.ehdr.entry`;不可以反复调用，不然会出现死循环。

3 内存管理

3.1 实验内容

3.1.1 lab2_1 虚实地址转换

应用的编译和链接并未指定程序中符号的逻辑地址。实现 `user_va_to_pa()` 函数，完成给定逻辑地址到物理地址的转换，并获得预期结果。

3.1.2 lab2_2 简单内存分配和回收

新定义了两个用户态函数 `naive_malloc()` 和 `naive_free()`，它们最终会转换成系统调用，完成内存的分配和回收操作。需要完成 `naive_free` 对应的功能，并获得预期的结果输出。

3.1.3 lab2_3 缺页异常

用户态栈空间仅有 1 个 4KB 的页面。在 PKE 操作系统内核中完善用户态栈空间的管理，使得它能够正确处理用户进程的“压栈”请求。

3.1.4 lab2_challenge1 复杂缺页异常

通过修改 PKE 内核（包括 `machine` 文件夹下的代码），使得对于不同情况的缺页异常进行不同的处理。

修改进程的数据结构以对虚拟地址空间进行监控。

修改 `kernel/strap.c` 中的异常处理函数。对于合理的缺页异常，扩大内核栈大小并为其映射物理块；对于非法地址缺页，报错并退出程序。

3.1.5 lab2_challenge2 堆空间管理

通过修改 PKE 内核（包括 `machine` 文件下的代码），实现优化后的 `malloc` 函数，使得应用程序两次申请块在同一页面，并且能够正常输出存入第二块中的字符串 "hello world"。

3.2 代码实现与分析

这一部分实验主要是区分了虚拟内存和物理内存，并在内核层面建立了虚拟内存与物理内存的映射。

Sv39 将 39 位虚拟地址“划分”为 4 个段：

[38,30]: 共 9 位，图中的 VPN[2]，用于在 512 (2^9) 个页目录 (page directory) 项中检索页目录项 (page directory entry, PDE)；

[29,21]: 共 9 位，图中的 VPN[1]，用于在 512 (2^9) 个页中间目录 (page medium

directory) 中检索 PDE;

[20,12]: 共 9 位, 图中的 VPN[0], 用于在 512 (2^9) 个页表 (page medium directory) 中检索 PTE;

[11,0]: 共 12 位, 图中的 offset, 充当 4KB 页的页内位移。

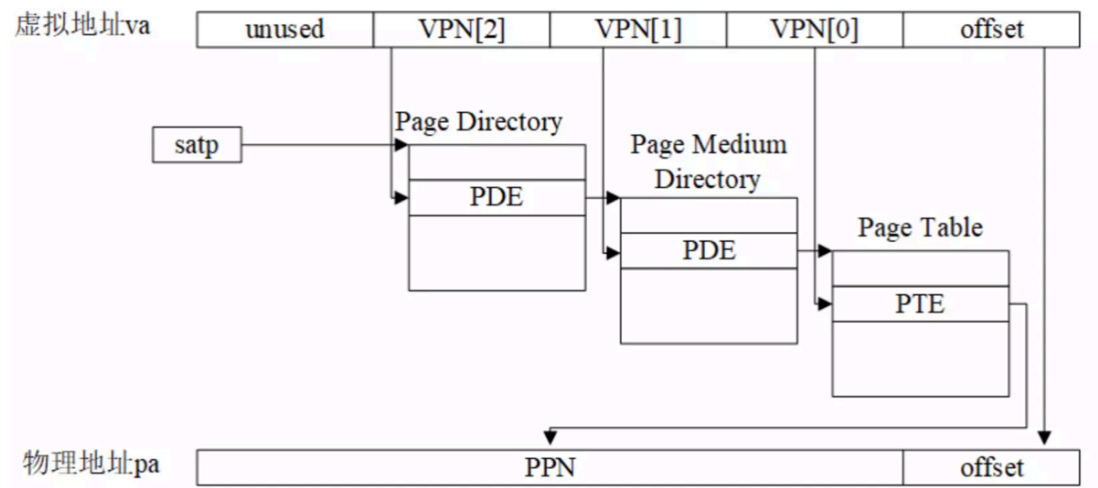


图 7 Sv39 中虚拟地址到物理地址的转换过程

所以在 `page_dir` 所指向的页表中查找逻辑地址 `va`, 需要通过调用页表操作相关函数找到包含 `va` 的页表项 (PTE), 通过该 PTE 的内容得知 `va` 所在的物理页面的首地址, 最后再通过计算 `va` 在页内的位移得到 `va` 最终对应的物理地址。

也就是说, 每连续 4KB 的内容存储在同一页, 但不同的页之间的逻辑关系不同, 因此需要操作系统提供一个链接和映射的渠道, 来完成这一工作。

`naive_malloc` 进行了内存申请和链接, `naive_free` 则需要进行解链接和释放。

在完成这个任务时, 发现在 `lab1_1` 中没有给 `syscall` 赋返回值, 如果不修改这个问题就会导致段错误, 丢失信息无法正常完成。

```
if(free)
```

```
    free_page(user_va_to_pa(page_dir, (void *)va));
```

```
(*page)^=1;
```

其中 `(*page)^=1` 表示将 Valid 位置 0。

在栈空间超出申请页的范围时, 需要用 `alloc_page()` 来申请一个新的页, 并使用 `map_page()` 来链接。

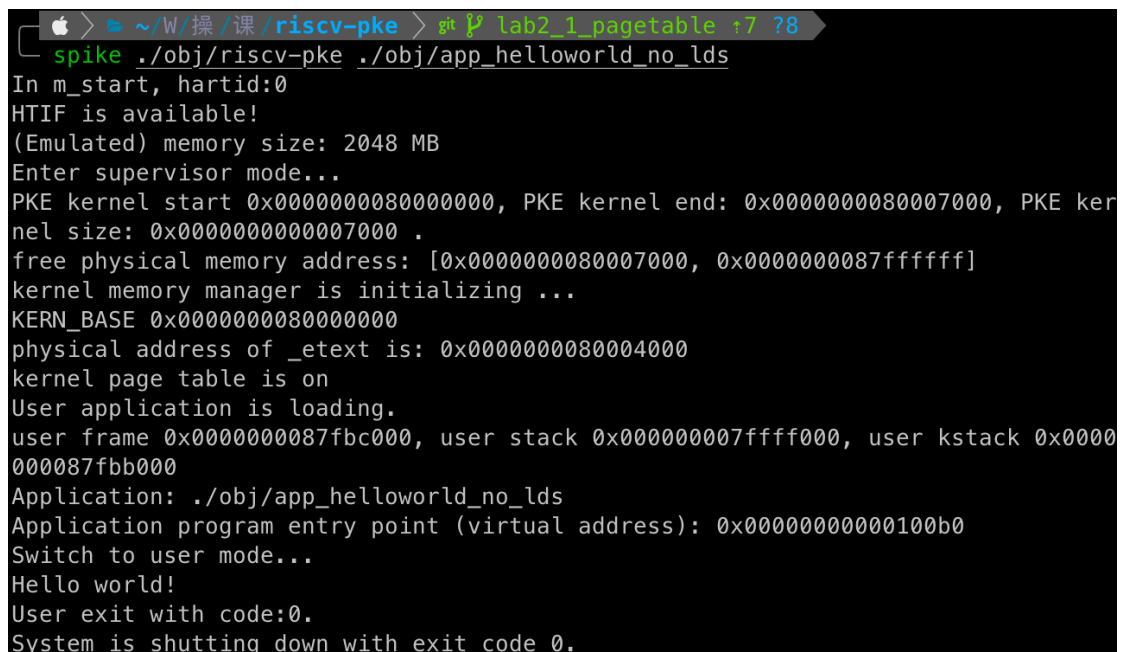
```
#define USER_FREE_ADDRESS_START 0x00000000 + PGSIZE * 1024
```

在 PKE 操作系统内核中，地址中预留了 1024 个页的空间，然后再开始动态分配内存。

找到一个给定 va 所对应的页表项 PTE；如果找到(过滤找不到的情形)，通过该 PTE 的内容得知 va 所对应物理页的首地址 pa；最后回收 pa 对应的物理页，并将 PTE 中的 Valid 位置为 0。

对于缺页处理，通过输入的参数 stval 判断缺页的逻辑地址在用户进程逻辑地址空间中的位置，判断是否比 USER_STACK_TOP 小，且比我们预设的可能的用户栈的最小栈底指针要大，也就是上述 START_ADDRESS，若满足，则为合法的逻辑地址。最后分配一个物理页，将所分配的物理页面映射到 stval 所对应的虚拟地址上。

```
if(stval<USER_STACK_TOP)
{
    uint64 page = (uint64)alloc_page();
    map_pages(current->pagetable, stval, 1, page,
prot_to_type(PROT_READ|PROT_WRITE,1));
}
```



```
Apple > ~/W/操/课/riscv-pke > # lab2_1_pagetable +7 ?8
spike ./obj/riscv-pke ./obj/app_helloworld_no_lds
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x0000000080007000, PKE kernel size: 0x0000000000007000 .
free physical memory address: [0x0000000080007000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080004000
kernel page table is on
User application is loading.
user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user kstack 0x0000000087fbb000
Application: ./obj/app_helloworld_no_lds
Application program entry point (virtual address): 0x00000000000100b0
Switch to user mode...
Hello world!
User exit with code:0.
System is shutting down with exit code 0.
```

图 8 lab2_1 结果


```

[ Apple > ~ / W / 操 / 课 / riscv-pke > P lab2_2_allocatepage +9 ?8
spike ./obj/riscv-pke ./obj/app_naive_malloc
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x0000000080007000, PKE kernel size: 0x0000000000007000 .
free physical memory address: [0x0000000080007000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080004000
kernel page table is on
User application is loading.
user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user kstack 0x0000000087fbb000
Application: ./obj/app_naive_malloc
Application program entry point (virtual address): 0x00000000000100b0
Switch to user mode...
s: 000000000400000, {a 1}
User exit with code:0.
System is shutting down with exit code 0.

```

图 9 lab2_2 结果

```

[ Apple > ~ / W / 操 / 课 / riscv-pke > P lab2_3_pagefault +12 ?8
spike ./obj/riscv-pke ./obj/app_sum_sequence
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x0000000080007000, PKE kernel size: 0x0000000000007000 .
free physical memory address: [0x0000000080007000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080004000
kernel page table is on
User application is loading.
user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user kstack 0x0000000087fbb000
Application: ./obj/app_sum_sequence
Application program entry point (virtual address): 0x00000000000100ce
Switch to user mode...
handle_page_fault: 000000007fffdff8
handle_page_fault: 000000007fffcff8
handle_page_fault: 000000007fffbff8
Summation of an arithmetic sequence from 0 to 1000 is: 500500
User exit with code:0.
System is shutting down with exit code 0.

```

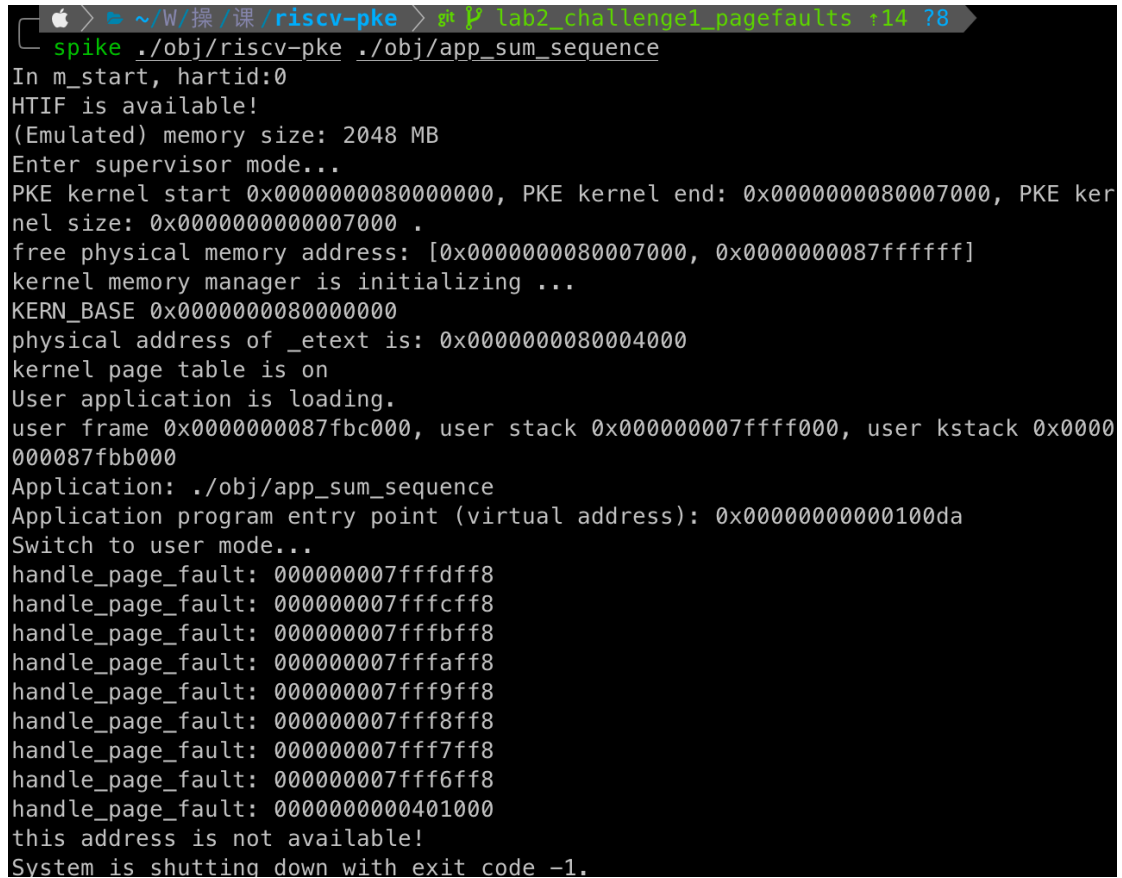
图 10 lab2_3 结果

3.3 挑战实验

复杂缺页异常的解决方式为：在 `handle_user_page_fault()` 中，判断 `PAGE_FAULT` 的原因是否在用户堆栈的有效范围之内，如果在则申请新页面，否则输出错误。

和上面 lab2_3 的区别仅为加强了判断范围。

```
stval<USER_STACK_TOP&&stval>=USER_STACK_TOP-20*(1<<12)
```



```
Apple > ~/W/操/课/riscv-pke > lab2_challenge1_pagefaults +14 ?8
spike ./obj/riscv-pke ./obj/app_sum_sequence
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x0000000080007000, PKE kernel size: 0x0000000000007000 .
free physical memory address: [0x0000000080007000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080004000
kernel page table is on
User application is loading.
user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user kstack 0x0000000087fbb000
Application: ./obj/app_sum_sequence
Application program entry point (virtual address): 0x00000000000100da
Switch to user mode...
handle_page_fault: 000000007ffffdf8
handle_page_fault: 000000007ffffcf8
handle_page_fault: 000000007ffffbf8
handle_page_fault: 000000007ffffaff8
handle_page_fault: 000000007ffff9ff8
handle_page_fault: 000000007ffff8ff8
handle_page_fault: 000000007ffff7ff8
handle_page_fault: 000000007ffff6ff8
handle_page_fault: 0000000000401000
this address is not available!
System is shutting down with exit code -1.
```

图 11 lab2_challenge1 结果

对于堆空间管理问题，解决方案为使用链表申请密集空间。设每次申请空间大小为 i ，依次对链表从前到后遍历，如果发现不小于 i 的空余位置，则把申请的空间左对齐插入到这里。

假设链表中 p 指向的是(第 x 页, 第 y_1 - y_2 个位置), $p \rightarrow next$ 指向的是(第 x 页, 第 z_1 - z_2 个位置), 那么需要保证的是 $z_1 - y_2 \geq i$ 。

链表定义如下（数组模拟）：

```
typedef struct used_t {
    uint64 page_num, page_begin, page_end;
    //page_num is va's 12~38bits
}used;
```

其中 `begin` 和 `end` 是左闭右开的。这样做时间复杂度较高，但是可以通过本挑战实验。

```

[~> ~/W/操/课/riscv-pke > # lab2_challenge2_singlepageheap +14 ?8
spike ./obj/riscv-pke ./obj/app_singlepageheap
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x000000008000b000, PKE kernel size: 0x00000000000b000 .
free physical memory address: [0x000000008000b000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080005000
kernel page table is on
User application is loading.
user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user kstack 0x0000000087fbb000
Application: ./obj/app_singlepageheap
Application program entry point (virtual address): 0x00000000000100c2
Switch to user mode...
p:000000000400064, m:000000000400000
hello, world!!!
User exit with code:0.
System is shutting down with exit code 0.

```

图 12 lab2_challenge2 结果

(注：结果中含有调试语句未删除，表示申请的页的位置)

3.4 实验总结

本实验的重点在于申请和释放空间，在这一过程中出现了虚拟地址和物理地址的映射与链接。在对空间的申请和释放中，有一些空间管理算法可以带来效率和性能的提升，如果在操作系统中进行微小的提升，那么在宏观应用程序中的改进也可能比较大。

`uint64 v=(uintptr_t)va;`可以做到强制类型转换。

4 进程管理

4.1 实验内容

4.1.1 lab3_1 进程创建 (fork)

完善操作系统内核 kernel/process.c 文件中的 do_fork()函数，并最终获得预期结果。

4.1.2 lab3_2 进程 yield

完善 yield 系统调用，实现进程执行过程中的主动释放 CPU 的动作。

4.1.3 lab3_3 循环轮转调度

实现 kernel/strap.c 文件中的 rrsched()函数，获得预期结果。

4.1.4 lab3_challenge1 进程等待和数据段复制

通过修改 PKE 内核和系统调用，为用户程序提供 wait 函数的功能，补充 do_fork 函数，lab3_1 实现了代码段的复制，继续实现数据段的复制并保证 fork 后父子进程的数据段相互独立。

4.1.5 lab3_challenge2 实现信号量

通过修改 PKE 内核和系统调用，为用户程序提供信号量功能。

添加系统调用，使得用户对信号量的操作可以在内核态处理。在内核中实现信号量的分配、释放和 PV 操作，当 P 操作处于等待状态时能够触发进程调度。

4.2 代码实现与分析

这一部分实验主要是完成操作系统对进程的调度，使用相关系统调用来完成进程处理的各种操作。通过结构体 process 所保存的信息来对进程进行维护。

do_fork()函数缺少的是父子进程之间的联系，需要把代码段信息继承到子进程中，因此建立如下映射。

```
map_pages(child->pagetable,      parent->mapped_info[i].va,
parent->mapped_info[i].npages*4096,
lookup_pa(parent->pagetable,      parent->mapped_info[i].va),
prot_to_type(PROT_READ|PROT_EXEC, 1));
```

do_fork()函数缺少的是父子进程之间的联系，需要把代码段信息继承到子进程中，因此建立如下映射。

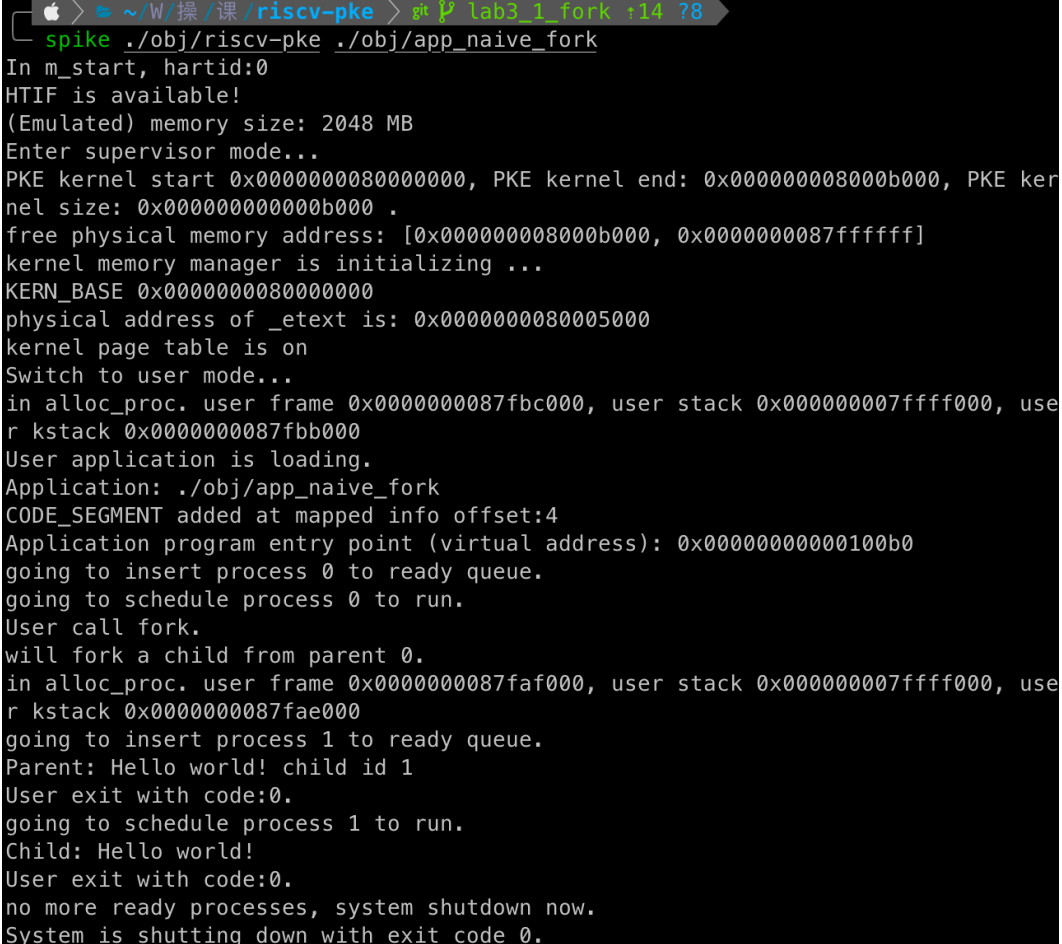
yield 系统调用，要实现进程执行过程中的主动释放 CPU 的动作，因此在 yield

实现过程中要进行 `schedule()`，调整当前进程状态，释放 CPU 给其他程序。

```
current->status=READY;
insert_to_ready_queue(current);
schedule();
```

时间片轮转时，要对每个进程控制其占有的时间片长度，一旦超出则将其转为 `READY` 并清空时间片。

```
if(current->tick_count+1>=TIME_SLICE_LEN)
{
    current->tick_count=0;
    insert_to_ready_queue(current);
    schedule();
}
else
    ++current->tick_count;
```



```
Apple > ~/W/操/课/riscv-pke > P lab3_1_fork +14 78
spike ./obj/riscv-pke ./obj/app_naive_fork
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x000000008000b000, PKE kernel size: 0x000000000000b000 .
free physical memory address: [0x000000008000b000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080005000
kernel page table is on
Switch to user mode...
in alloc_proc. user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user kstack 0x0000000087fbb000
User application is loading.
Application: ./obj/app_naive_fork
CODE_SEGMENT added at mapped info offset:4
Application program entry point (virtual address): 0x0000000000100b0
going to insert process 0 to ready queue.
going to schedule process 0 to run.
User call fork.
will fork a child from parent 0.
in alloc_proc. user frame 0x0000000087faf000, user stack 0x000000007ffff000, user kstack 0x0000000087fae000
going to insert process 1 to ready queue.
Parent: Hello world! child id 1
User exit with code:0.
going to schedule process 1 to run.
Child: Hello world!
User exit with code:0.
no more ready processes, system shutdown now.
System is shutting down with exit code 0.
```

图 13 lab3_1 结果

```

Parent running 50000
going to insert process 0 to ready queue.
going to schedule process 1 to run.
Child running 50000
going to insert process 1 to ready queue.
going to schedule process 0 to run.
Parent running 60000
going to insert process 0 to ready queue.
going to schedule process 1 to run.
Child running 60000
going to insert process 1 to ready queue.
going to schedule process 0 to run.
User exit with code:0.
going to schedule process 1 to run.
User exit with code:0.
no more ready processes, system shutdown now.
System is shutting down with exit code 0.

```

图 14 lab3_2 结果节选

```

Ticks 13
going to insert process 0 to ready queue.
going to schedule process 1 to run.
Child running 80000000
Ticks 14
Child running 90000000
Ticks 15
going to insert process 1 to ready queue.
going to schedule process 0 to run.
User exit with code:0.
going to schedule process 1 to run.
User exit with code:0.
no more ready processes, system shutdown now.
System is shutting down with exit code 0.

```

图 15 lab3_3 结果节选

4.3 挑战实验

进程等待和数据段复制和前面的 fork 子进程做法类似，进程等待可以参考前面 yield 的做法。此时需要在 procs 池里面寻找自己的子进程，并找到 wait()所需的信号量以及变量。

其中还要注意一个地方，在进行映射时，父进程的 page 不一定只有一面，使用 npages 控制父进程页面数，对于每个页面都需要进行相应的继承操作。

```
for(int j=0;j<parent->mapped_info[i].npages;++j)
```

```
...
```

```
user_vm_map(child->pagetable,
```

```
parent->mapped_info[i].va + j*4096, 4096,
```

```
(uint64)child_page, prot_to_type(PROT_READ|PROT_WRITE, 1));
```

同时，如果一个进程不是 Zombie 进程，在它的时间片结束之后就需要

yield, 如果是, 则马上通知父进程回收, 并把状态置为 FREE。

```
going to schedule process 1 to run.
User call fork.
will fork a child from parent 1.
in alloc_proc. user frame 0x0000000087fa1000, user stack 0x000000007ffff000, use
r kstack 0x0000000087fa0000
going to insert process 2 to ready queue.
wait going to insert process 1 to ready queue.
going to schedule process 0 to run.
Parent process end, flag = 0.
User exit with code:0.
going to schedule process 2 to run.
Grandchild process end, flag = 2.
User exit with code:0.
going to schedule process 1 to run.
Child process end, flag = 1.
User exit with code:0.
no more ready processes, system shutdown now.
System is shutting down with exit code 0.
```

图 16 lab3_challenge1 结果节选

对于信号量的实现, 需要先定义一组信号量如下, 当需要信号量时从中申请, 使用完毕之后需要释放。其中*head 指的是等待这个信号量的排队的进程的首指针。

```
typedef struct semaphore_t {
    int value;
    process *head;
}semaphore;
semaphore sem[512];
```

对于 P 操作, 如果信号量为负, 说明资源不够了, 需要等待, 所以把调用的进程加入 head; 对于 V 操作, 只要调用就说明有资源释放, 所以就检查 head 是否为空。

```
Parent print 8
going to insert process 1 to ready queue.
going to schedule process 1 to run.
Child0 print 8
going to insert process 2 to ready queue.
going to schedule process 2 to run.
Child1 print 8
going to insert process 0 to ready queue.
going to schedule process 0 to run.
Parent print 9
going to insert process 1 to ready queue.
User exit with code:0.
going to schedule process 1 to run.
Child0 print 9
going to insert process 2 to ready queue.
User exit with code:0.
going to schedule process 2 to run.
Child1 print 9
User exit with code:0.
no more ready processes, system shutdown now.
System is shutting down with exit code 0.
```

图 17 lab3_challenge2 结果节选

4.4 实验总结

本实验的重点在于对进程各个状态的管理和调度。主要是进程之间的继承和依赖关系，同时把这些内容结合前面的实验，完成内存地址的映射和管理。

同时这组实验也进一步深化了对进程和信号量的理解，它们其实是可以动态申请，也可以动态存为一个池，有需要则申请，没有需要就释放。

实现信号量时，V 操作不需要 `schedule()`，因为这会打乱现有的程序调度。

5 总结与收获

本次操作系统内核实验使我既收获丰富，又富有成就感。选择操作系统内核实验是因为我使用 Apple 芯片的 macOS 系统，可以作为很好的类 Unix 实验平台，同时也是因为我认为自己的操作系统理论课不够扎实，需要一些细节上的复习来掌握得更牢靠。

在这期间，我对 git 的掌握愈发熟练，同时也对命令行交互更加着迷。想起早起计算机的 DOS 系统没有图形界面，只有命令，让我产生了一种对计算机的亲切感。命令是最精准的，也是最直接的，在调试过程中，有许多问题我都是通过命令行反馈和输出到文件才发现的，而且安装时内核的许多状态也都可以实时反馈到窗口中。

在接触操作系统内核之前，我对操作系统的理解还比较朦胧，因为我理解的是把应用程序放在操作系统的“内部”运行，但是一直没有明白怎么放、放什么。完成这个实验之后，我认为应用程序能运行在操作系统上，有十分精巧且复杂的过程，在这些过程中又有无数个巧妙的方法将存在的问题恰到好处地化解。这又让我觉得，其实现在的操作系统或操作系统内核还存在着优化空间。

我认为这次实验的类型十分有意思，在闯关的模式下逐步化解难题，题目难度随着对系统的理解逐步递进。如果让我直接解决最后一个问题，我一定是无法想到答案的。但是操作系统的构建其实也是从小到大的，这次实验像一个放大镜，把操作系统内核呈现在我的眼前。

参考文献

- [1] aadimao 等.GIT（分布式版本控制系统）_百度百科[EB/OL]. 百度百科,
<https://baike.baidu.com/item/GIT/12647237?fr=aladdin>. 2023-05-15
- [2] 全 国 大 学 生 计 算 机 系 统 能 力 大 赛 [EB/OL].
<https://compiler.educg.net/?op=4#/>

附 录

lab1_challenge1 部分实现代码:

```
elf_status elf_load_symbol(elf_ctx *ctx, char namelist[], elf_sym sym[],
int* symnum) {
    elf_shdr sh;
    int i, off;
    // load symbol table and string table
    int symcnt=0, strcnt=0;
    char tmp[512];
    if (elf_fpread(ctx, (void *)&sh, sizeof(sh),
ctx->ehdr.shoff+ctx->ehdr.shstrndx*sizeof(elf_shdr)) != sizeof(sh))
return EL_EIO;
    elf_fpread(ctx, (void *)tmp, sizeof(tmp), sh.offset);
    int j;
    for (i = 0, off = ctx->ehdr.shoff; i < ctx->ehdr.shnum; ++i, off +=
sizeof(elf_shdr)) {
        if (elf_fpread(ctx, (void *)&sh, sizeof(sh), off) != sizeof(sh)) return
EL_EIO;
        if (strcmp(tmp+sh.name, ".symtab") == 0) { // symbol table
            if (elf_fpread(ctx, sym, sh.size, sh.offset) != sh.size)
                return EL_EIO;
            symcnt += sh.size;
        } else if (strcmp(tmp+sh.name, ".strtab") == 0) { // string table
            if (elf_fpread(ctx, namelist + strcnt, sh.size, sh.offset) !=
sh.size)
                return EL_EIO;
            strcnt += sh.size; //there may be several string tables
        }
        // sprintf("sh_size=%ld sh_offset=%ld\n, sh_type=%d\n",sh.size,
sh.offset, sh.type);
        // sprintf("symcnt=%d strcnt=%d\n",symcnt, strcnt);
        *symnum=symcnt;
        return EL_OK;
    }
}
```

lab1_challenge2 部分实现代码:

```
for(i=0; ; ++i)
{
    // sprintf("line number:%ld, addr number:%ld\n",current->line[i].line,
current->line[i].addr);
    if(current->line[i].addr==mepc)
    {
        uint64 file_index=current->line[i].file;
```

```

        uint64 dir=current->file[file_index].dir;
        sprintf("Runtime                                     error
at  %s/%s:%ld\n",current->dir[dir],current->file[file_index].file,current
t->line[i].line);

```

```

        struct stat mystat;
        char path[200], code;
        int len=strlen(current->dir[dir]);
        strcpy(path, current->dir[dir]);
        strcpy(path+len+1, current->file[file_index].file);
        path[len]='/';
        path[len+1+strlen(current->file[file_index].file)]='\0';
        sprintf("path:%s\n",path);
        spike_file_t *f = spike_file_open(path, O_RDONLY, 0);
        int cur,line=1;
        for(cur=0;;++cur)
        {
            spike_file_pread(f, &code, 1, cur);
            if(line==current->line[i].line)
            {
                sprintf("%c",code);
                if(code=='\n')
                    break;
            }
            if(code=='\n')
                ++line;
        }
        spike_file_close(f);
        break;
    }
}

```

lab2_challenge1 部分实现代码:

```

        if(stval<USER_STACK_TOP&&stval>=USER_STACK_TOP-20*(1<<12))
        {
            uint64 page = (uint64)alloc_page();
            map_pages(current->pagetable, stval, 1, page,
prot_to_type(PROT_READ|PROT_WRITE,1));
        }
        else
            panic("this address is not available!");

```

lab2_challenge2 部分实现代码:

```

typedef struct used_t {
    uint64 page_num, page_begin, page_end;
    //page_num is va's 12~38bits
}used;
used heap[512];
uint64 tot=0;
uint64 sys_user_allocate_page(uint64 width) {
    // heap[0].page_num=0;
    // heap[0].page_off=0; left_bound
    int i;
    for(i=0;i<tot;++i)
    {
        uint64 remain=0;
        if(i==0)
            remain=heap[0].page_begin;//special
        else if(heap[i-1].page_num!=heap[i].page_num)
            remain=4096-heap[i-1].page_end;
        else
            remain=heap[i].page_begin-heap[i-1].page_end;
        // sprint("remain:%ld\n",remain);
        if(remain>=width)
        {
            int j;
            for(j=tot+1;j>i;--j)
                heap[j]=heap[j-1];
            heap[i].page_num=heap[i-1].page_num;
            heap[i].page_begin=heap[i-1].page_end;
            heap[i].page_end=heap[i].page_begin+width;
            ++tot;
            return heap[i].page_num|heap[i].page_begin;
        }
    }
    //for the tail
    if(tot){
        uint64 remain=4096-heap[tot-1].page_end;
        if(remain>=width)
        {
            int j;
            for(j=tot+1;j>i;--j)
                heap[j]=heap[j-1];
            heap[i].page_num=heap[i-1].page_num;
            heap[i].page_begin=heap[i-1].page_end;
            heap[i].page_end=heap[i].page_begin+width;
            ++tot;
        }
    }
}

```

```

    return heap[i].page_num|heap[i].page_begin;
}}
//
void* pa = alloc_page();
uint64 va = g_ufree_page;
g_ufree_page += PGSIZE;
user_vm_map((pagetable_t)current->pagetable, va, PGSIZE, (uint64)pa,
            prot_to_type(PROT_WRITE | PROT_READ, 1));

heap[tot].page_num=va;
heap[tot].page_begin=0;
heap[tot].page_end=0;
++tot;
heap[tot].page_num=va;
heap[tot].page_begin=0;
heap[tot].page_end=width;
++tot;
return va;
}

```

lab3_challenge1 部分实现代码:

```
extern process procs[];
```

```

ssize_t sys_user_wait(ssize_t pid) {
    sprint("wait %l\n",pid);
    if(pid==-1)
    {
        while(1)
        {
            for(int i=0;i<NPROC;++i)
            {
                if(procs[i].parent==current && procs[i].status == ZOMBIE)
                {
                    procs[i].status = FREE;
                    return procs[i].pid;
                }
            }
            sys_user_yield();
        }
    }
    else if(pid>0&&pid<NPROC&&procs[pid].parent==current)
    {
        while(procs[pid].status!=ZOMBIE)
    }
}

```

```

    {
        sys_user_yield();
    }
    procs[pid].status = FREE;
    return pid;
}
else
    return -1;
}

```

lab3_challenge2 部分实现代码:

```

typedef struct semaphore_t {
    int value;
    process *head;
}semaphore;
semaphore sem[512];
int semcnt=0;
ssize_t sys_user_sem_new(int num) {
    sem[semcnt].value=num;
    sem[semcnt].head=NULL;
    return semcnt++;
}

ssize_t sys_user_sem_P(int semnum) {
    // sprintf("semp:%d\n",semnum);
    sem[semnum].value--;
    // sprintf("sempv:%d\n",sem[semnum].value);
    if(sem[semnum].value<0)
    {
        if(!sem[semnum].head)
            sem[semnum].head=current;
        else
        {
            process *p=sem[semnum].head;
            while(p->queue_next!=NULL)
                p=p->queue_next;
            p->queue_next=current;
        }
        current->queue_next=NULL;
        current->status=BLOCKED;
        schedule();
    }
    return 0;
}

```

```
ssize_t sys_user_sem_V(int semnum) {
    sem[semnum].value++;
    if(sem[semnum].head!=NULL)
    {
        // sem[semnum].head->status=READY;
        insert_to_ready_queue(sem[semnum].head);
        // schedule();
        sem[semnum].head=sem[semnum].head->queue_next;
        // --sem[semnum].value;
    }
    return 0;
}
```


教师评语评分

评语： _____

评分： _____

评阅人：

年 月 日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）