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 ： 宋伟

学 生 学 号 ： 2020302191970

学 生 姓 名 ： msm8976

二〇二二年七月

目 录

摘　要

本次操作系统内核实验采用代理内核思想，模拟出一个 RISC-V 平台上极简的操作系统内核，并具有概念和功能上的完整性。实验内容主要包括系统调用、内存管理和进程管理三个方面的九个基础实验与六个挑战实验。在完成基础实验的过程中，通过阅读、学习、简要补全给出的内核，建立操作系统课程中所学习到的概念对应的工程上的认识。在完成挑战实验的过程中通过完整的编写关键函数，加深对所学知识的理解，进一步对计算机的硬件和软件所构成的整体系统建立起较为完整的认识。

关键词：操作系统；RISC-V；内核

1 实验要求与目的

本实验要求同学们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，同学们首先要构建一个类Linux环境，通过直接安装ubuntu或麒麟操作系统或openEuler操作系统，或者在windows10环境下利用WSL+MobaXterm搭建一个类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。目前大家的电脑基本上都是x86的CPU，要设计能在模拟RISC-V硬件上运行的操作系统内核，就需要在linux中安装RISC-V交叉编译器，使得编译出来的应用程序可以在模拟RISC-V硬件上运行，以测试内核能否正常运行。

目前我们已经拥有了一个基于开源项目设计的操作系统内核，这是一个代理内核。代理内核的好处就在于只需要提供应用程序运行的最小操作系统内核服务支持，就可以让应用程序运行起来。完整的操作系统内核太过庞大，涉及的内容太多，难以阅读。为了降低实验难度，我们选择了代理内核方案，这个代理内核里有一部分内容被抽取出来了。为了支持应用程序的运行，比如helloworld程序的运行，就需要把代理内核里缺失的内容补充完整，这样才能在屏幕上看见helloworld 字样。所以同学们要做的工作就是读懂内核并填空，使得给定的应用程序可以正确运行。

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。同一个部分的基础实验要按顺序完成，他们之间也存在依赖关系。挑战实验要在基础实验完成的基础上才能做，但后一个部分的基础实验不依赖于前一个部分的挑战实验的结果。基础实验是必须完成的，挑战实验可以选择完成其中任意一个。

2 实验环境配置

2.1 实验硬件

CPU：AMD Ryzen 7 4800U(x64)

操作系统：Windows 10

2.2 环境配置

使用Windows 10的WSL搭建使用环境，使用WSL1版本并下载Ubuntu 18.05.4 LTS的Linux系统。使用命令安装完本地支撑软件包后，下载自包含交叉编译器+执行环境的riscv64-elf-gcc-20210923.tgz包在根目录解压，并设置环境变量后实验环境配置完成。

使用 git clone 可将实验所需代码下载至本地，尝试编译运行后可以看到lab1_1 的相关输出，证明实验环境配置成功。为了方便编辑代码及查看文件结构，在Linux中安装VSCode与Remote相关拓展，即可在Windows中的VSCode内编辑查看文件，通过Save as Root in Remote拓展可直接保存编辑后文件。

2.3 实验介绍

本次实验的代码已上传至GitHub(https://github.com/msm8976/OS-PKE)。由于在完成后续实验后发现lab2_2的实现存在问题，而后续实验已继承该分支的修改内容，若再次修改可能出现冲突或错误，因此仅在实验报告中修改为正确代码（已在对应位置标红），未在git内修改。

基础实验复杂在根据提示信息找到需要修改的位置，需要分析多个函数的调用与内核模式的变化，而提示中均已告知具体的分析过程，在修改位置的 TODO中也对需要添加的代码的功能进行介绍，因此基础实验的实验报告分为实验内容、分析过程，即分析需要修改的位置、实验结果，即针对目标位置所需执行的功能编写代码三项。

挑战实验继承自基础实验，修改位置可以通过前三个基础实验的分析过程得到，而更加注重如何通过实验内容得出一个可行的思路，再根据其进行具体的代码实现，因此挑战实验的实验报告分为实验内容、原理分析，即通过实验内容分析出的原理与思路、实现过程，即具体实现时对函数、结构的修改过程、实验结果，即本实验修改的所有内容与程序最终正确的输出四项。

3 系统调用、异常和中断处理

3.1 lab1_1 系统调用

3.1.1 实验内容

实验代码文件：user/app_helloworld.c

```
#include "user_lib.h"

int main(void) {
  printu("Hello world!\n");

  exit(0);
}
```

其功能为输出“Hello world!”，编译后直接运行报错：

call do_syscall to accomplish the syscall and lab1_1 here.

需要找到并完成对 do_syscall 的调用，并获得以下预期结果：

```
$ 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
Switching to user mode...
Hello world!
User exit with code:0.
System is shutting down with exit code 0.
```

3.1.2 分析过程

查看 printu()函数的定义，位于 user/user_lib.c 中，实现方式为

```
do_user_call(SYS_user_print, (uint64)buf, n, 0, 0, 0, 0, 0);
```

查看 do_user_call()函数，发现其先将函数的 8 个参数载入到 RISC-V 机器的 a0 到 a7 这 8 个寄存器中，再通过 ecall 指令完成系统调用。ecall 指令的执行将根据 a0 寄存器中的值获得系统调用号，并使 RISC-V 转到 S 模式的 trap 处理入口执行。

```
uint64 do_user_call(uint64 sysnum, uint64 a1, uint64 a2, uint64 a3,
uint64 a4, uint64 a5, uint64 a6, uint64 a7) {
  int ret;
```

```
  // before invoking the syscall, arguments of do_user_call are already
loaded into the argument
  // registers (a0-a7) of our (emulated) risc-v machine.
  asm volatile(
      "ecall\n"
      "sw a0, %0"  // returns a 32-bit value
      : "=m"(ret)
      :
      : "memory");

  return ret;
}
```

在 kernel/strap_vector.S 文件中可看到 trap 的汇编处理，首先将进程运行现场进行保存，再将 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，将用户栈切换至用户内核栈。后续的执行将使用应用进程所附带的内核栈来保存执行的上下文，最后将应用进程中的 p->trapframe->kernel_trap 写入 t0 寄存器，调用 p->trapframe->kernel_trap 所指向的 smode_trap_handler()函数。

smode_trap_handler()函数位于 kernel/strap.c 文件，该函数在确定当前为用户模式并保存发生系统调用的指令地址后，判断如果为系统调用就执行 handle_syscall()函数，而 handle_syscall()函数内即为 lab1_1 所需完成的位置。

```
static void handle_syscall(trapframe *tf) {
  tf->epc += 4;
  // TODO (lab1_1): remove the panic call below, and call do_syscall
(defined in
  // kernel/syscall.c) to conduct real operations of the kernel side for
a syscall.
  // IMPORTANT: return value should be returned to user app, or else,
you will encounter
  // problems in later experiments!
  panic( "call do_syscall to accomplish the syscall and lab1_1
here.\n" );
}
```

根据 lab1_1 的 TODO 注释，需要将 panic 语句替换为对 do_syscall()函数的调用，并将返回值返回至发出系统调用的用户应用。

3.1.3 实验结果

在 do_user_call()函数中，函数的 8 个参数已由编译器初始化至 RISC-V 机器的 a0 到 a7 寄存器，因此调用 do_syscall()函数的参数为 a0 到 a7 这 8 个寄存器的值。对于其返回值，在 do_user_call()内的汇编代码可以看出将其存入 a0，因此在这里将返回值赋给 a0 寄存器。

将 panic 语句替换为

```
tf->regs.a0=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;
```

后，再次编译运行可正常输出“Hello world!”。

3.2 lab1_2 异常处理

3.2.1 实验内容

实验代码文件：user/app_illegal_instruction.c

```
int main(void) {
  printu("Going to hack the system by running privilege
instructions.\n");
  // we are now in U(user)-mode, but the "csrw" instruction requires M-
mode privilege.
  // Attempting to execute such instruction will raise illegal
instruction exception.
  asm volatile("csrw sscratch, 0");
  exit(0);
}
```


其功能为应用在用户 U 模式尝试执行 RISC-V 的特权指令 csrw sscratch, 0，编译后直接运行报错：

call handle_illegal_instruction to accomplish illegal instruction interception of lab1_2.

需要找到对应位置调用 handle_illegal_instruction 函数完成异常指令处理，并获得以下预期结果：

```
$ 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
```

```
Switching to user mode...
Going to hack the system by running privilege instructions.
Illegal instruction!
System is shutting down with exit code -1.
```

3.2.2 分析过程

在 user/app_illegal_instruction.c 文件中，程序“企图”执行不能在用户模式（U 模式）运行的特权级指令：csrw sscratch, 0，对于机器而言该事件并不是它期望（它期望在用户模式下执行的都是用户模式的指令）发生的“异常事件”，需要介入和破坏该应用程序的执行。根据 RISC-V 体系结构，这类异常属于非法指令异常，即 CAUSE_ILLEGAL_INSTRUCTION。

由于 PKE 操作系统内核在启动时会将部分异常和中断“代理”给 S 模式处理，需要判断 CAUSE_ILLEGAL_INSTRUCTION 异常在哪个模式处理。可以在 kernel/machine/minit.c 文件中的 delegate_traps()函数查看代理给 S 模式处理的异常与中断，但并不存在 CAUSE_ILLEGAL_INSTRUCTION 异常，因此该异常交给 M 模式来处理。在 kernel/machine/mtrap_vector.S 文件中可以查看 M 模式的 trap 处理入口：

```
mtrapvec:
    csrrw a0, mscratch, a0

    addi t6, a0, 0
    store_all_registers
    csrr t0, mscratch
    sd t0, 72(a0)

    la sp, stack0
    li a3, 4096
    csrr a4, mhartid
    addi a4, a4, 1
    mul a3, a3, a4
    add sp, sp, a3

    csrw mscratch, a0

    call handle_mtrap

    csrr t6, mscratch
    restore_all_registers
```

```
    mret
```

在 mtrapvec 汇编函数保存当前应用的相关信息后会切换栈到 stack0，并调用 handle_mtrap()函数，可以在在 kernel/machine/mtrap.c 文件中找到对应异常的处理方式：

```
void handle_mtrap() {
  uint64 mcause = read_csr(mcause);
  switch (mcause) {
    case CAUSE_ILLEGAL_INSTRUCTION:
      // TODO (lab1_2): call handle_illegal_instruction to implement
illegal instruction
      // interception, and finish lab1_2.
      panic( "call handle_illegal_instruction to accomplish illegal
instruction interception for lab1_2.\n" );
      break;
  }
}
```

根据 lab1_2 的 TODO 注释，需要将 panic 语句替换为对 handle_illegal_instruction() 函数的调用，来处理 CAUSE_ILLEGAL_INSTRUCTION 异常。

3.2.3 实验结果

参照 handle_mtrap()函数内其他已完成的异常处理，在原因为 CAUSE_ILLEGAL_INSTRUCTION 的 case 内调用 handle_illegal_instruction()函数。

将 panic 语句替换为

```
handle_illegal_instruction();
```

后，再次编译运行可正常处理非法指令异常，输出“Illegal instruction!”并中止程序运行。

3.3 lab1_3 （外部）中断

3.3.1 实验内容

实验代码文件：user/app_long_loop.c

```
#include "user_lib.h"
#include "util/types.h"
```

```c
int main(void) {
  printu("Hello world!\n");
  int i;
  for (i = 0; i < 100000000; ++i) {
    if (i % 5000000 == 0) printu("wait %d\n", i);
  }

  exit(0);

  return 0;
}
```

其功能为执行一个长度为 100000000 次的循环，循环每次将整型变量 i 加一，当 i 的值是 5000000 的整数倍时，输出"wait i 的值\n"，编译后直接运行报错：lab1_3: increase g_ticks by one, and clear SIP field in sip register.

需要完成 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，并获得以下预期结果：

```
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): 0x000000008100007e
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
wait 75000000
wait 80000000
wait 85000000
Ticks 6
wait 90000000
wait 95000000
Ticks 7
User exit with code:0.
System is shutting down with exit code 0.
```

3.3.2 分析过程

分析直接编译运行输出的报错，程序在运行过程中受到了系统的外部时钟中断（timer irq）的干扰，在碰到第一个时钟中断后就会出现崩溃，因此需要寻找外部时钟中断具体的处理位置。

在 kernel/machine/minit.c 文件的 timerinit()函数中，可以看到机器时钟的初始化，设置下一次 timer 触发的时间为当前时间的 TIMER_INTERVAL 之后，并允许机器在 M 模式处理 timer 中断：

```
void timerinit(uintptr_t hartid) {
  // fire timer irq after TIMER_INTERVAL from now.
  *(uint64*)CLINT_MTIMECMP(hartid) = *(uint64*)CLINT_MTIME +
TIMER_INTERVAL;

  // enable machine-mode timer irq in MIE (Machine Interrupt Enable)
csr.
  write_csr(mie, read_csr(mie) | MIE_MTIE);
}
```

触发时钟中断后，会调用 kernel/machine/mtrap_vector.S 文件中的 mtrapvec 汇编函数，在保存当前应用的相关信息后调用 handle_mtrap()函数，对于 CAUSE_MTIMER 中断调用 handle_timer()函数进行处理：

```
void handle_mtrap() {
  uint64 mcause = read_csr(mcause);
  switch (mcause) {
    case CAUSE_MTIMER:
      handle_timer();
      break;
  }
}
```

```
static void handle_timer() {
  int cpuid = 0;
  *(uint64*)CLINT_MTIMECMP(cpuid) = *(uint64*)CLINT_MTIMECMP(cpuid) +
TIMER_INTERVAL;
  write_csr(sip, SIP_SSIP);
}
```

handle_timer()函数在设置下一次时钟中断的触发时间后，对 SIP（Supervisor Interrupt Pending，即 S 模式的中断等待寄存器）寄存器中的 SIP_SSIP 位进行设置，使内核在 S 模式收到一个来自 M 态的时钟中断请求（CAUSE_MTIMER_S_TRAP）。

对于 S 态的中断，通过 kernel/strap.c 文件中的 smode_trap_handler()函数处理，在确定当前为用户模式并保存发生系统调用的指令地址后，判断如果为 CAUSE_MTIMER_S_TRAP 时钟中断就执行 handle_mtimer_trap()函数，而 handle_mtimer_trap()函数内即为 lab1_3 所需完成的位置。

```
void handle_mtimer_trap() {
  sprint("Ticks %d\n", g_ticks);
  // TODO (lab1_3): increase g_ticks to record this "tick", and then
clear the "SIP"
  // field in sip register.
  // hint: use write_csr to disable the SIP_SSIP bit in sip.
    panic( "lab1_3: increase g_ticks by one, and clear SIP field in sip
register.\n" );
}
```

根据 lab1_3 的 TODO 注释，需要将 panic 语句替换为将 g_ticks+1，并清空 SIP 寄存器的 SIP_SSIP 位来处理时钟中断。

3.3.3 实验结果

将 g_ticks+1：g_ticks++

清空 SIP 寄存器的 SIP_SSIP 位：write_csr(sip, 0)

将 panic 语句替换为

```
g_ticks++;
write_csr(sip, 0);
```

后，再次编译运行可正常处理时钟中断，在程序运行的适当时刻插入 Ticks 输出。

4 内存管理

4.1 lab2_1 虚实地址转换

4.1.1 实验内容

实验代码文件：user/app_helloworld_no_lds.c

```
#include "user_lib.h"
#include "util/types.h"

int main(void) {
  printu("Hello world!\n");
  exit(0);
}
```

其功能为输出“Hello world!”，且编译和链接并未指定程序中符号的逻辑地址，编译后直接运行报错：

```
You have to implement user_va_to_pa (convert user va to pa) to print messages in lab2_1.
```

需要实现用以将逻辑地址转换为物理地址的 user_va_to_pa()函数，并获得以下预期结果：

```
$ 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:
0x000000008000e000, PKE kernel size: 0x000000000000e000 .
free physical memory address: [0x000000008000e000, 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): 0x00000000000100f6
Switching to user mode...
Hello world!
User exit with code:0.
System is shutting down with exit code 0.
```

4.1.2 分析过程

本实验用以输出的 printu()函数与 lab1_1 的调用路径完全相同，通过 do_user_call()函数内的 ecall 指令转到 S 模式通过 smode_trap_handler()函数执行系统调用。判断为 CAUSE_USER_ECALL 后，handle_syscall()函数会使用对应的寄存器参数调用 do_syscall()函数，并通过 SYS_user_print 的 switch 分支执行 sys_user_print()函数来输出内容。

```
ssize_t sys_user_print(const char* buf, size_t n) {
  // buf is now an address in user space of the given app's user stack,
  // so we have to transfer it into phisical address (kernel is running
in direct mapping).
  assert( current );
  char* pa = (char*)user_va_to_pa((pagetable_t)(current->pagetable),
(void*)buf);
  sprint(pa);
  return 0;
}
```

可以看到sys_user_print()函数中使用user_va_to_pa()函数来获取所需显示内容的地址，而 user_va_to_pa()函数内即为 lab2_1 所需完成的位置。

```
void *user_va_to_pa(pagetable_t page_dir, void *va) {
  // TODO (lab2_1): implement user_va_to_pa to convert a given user
virtual address "va"
  // to its corresponding physical address, i.e., "pa". To do it, we
need to walk
  // through the page table, starting from its directory "page_dir", to
locate the PTE
  // that maps "va". If found, returns the "pa" by using:
  // pa = PYHS_ADDR(PTE) + (va & (1<<PGSHIFT -1))
  // Here, PYHS_ADDR() means retrieving the starting address (4KB
aligned), and
  // (va & (1<<PGSHIFT -1)) means computing the offset of "va" inside
its page.
  // Also, it is possible that "va" is not mapped at all. in such case,
we can find
  // invalid PTE, and should return NULL.
  panic( "You have to implement user_va_to_pa (convert user va to pa) to
print messages in lab2_1.\n" );
}
```

根据 lab2_1 的 TODO 注释，需要将 panic 语句替换为 user_va_to_pa ()函数的具体实现，该函数输入页表与逻辑地址，将其转换为物理地址并返回，从而让

sprint()函数可以获取到存放输出信息的物理地址。

4.1.3 实验结果

对于函数输入的页表与逻辑地址，先要通过 page_walk()函数找到逻辑地址所对应的页表项：

```
pte_t *PTE=page_walk(page_dir,(uint64)va,0);
```

获取到页表项后，若非空则可通过页表项来获取其对应页面的物理首地址，再与逻辑地址的页内偏移相加即可获取到对应的物理地址：

```
pa=((*PTE>>10)<<12) + ((uint64)va & ((1<<PGSHIFT) -1));
```

将 panic 语句替换为

```
pte_t *PTE=page_walk(page_dir,(uint64)va,0);
uint64 pa;
if(PTE==NULL)
    return NULL;
pa=((*PTE>>10)<<12) + ((uint64)va & ((1<<PGSHIFT) -1));
return (void*)pa;
```

后，再次编译运行可正常执行逻辑地址物理地址转换，输出“Hello world!”。

4.2 lab2_2 简单内存分配和回收

4.2.1 实验内容

实验代码文件：user/app_naive_malloc.c

```
struct my_structure {
  char c;
  int n;
};

int main(void) {
  struct my_structure* s = (struct my_structure*)naive_malloc();
  s->c = 'a';
  s->n = 1;

  printu("s: %lx, {%c %d}\n", s, s->c, s->n);

  naive_free(s);
  exit(0);
}
```

其功能为使用 naive_malloc()、naive_free()两个函数进行内存的分配与回

收，编译后直接运行报错：

You have to implement user_vm_unmap to free pages using naive_free in lab2_2.

需要完成 naive_free()函数对应的功能，并获得以下预期结果：

```
$ 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:
0x000000008000e000, PKE kernel size: 0x000000000000e000 .
free physical memory address: [0x000000008000e000, 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): 0x0000000000010078
Switching to user mode...
s: 0000000000400000, {a 1}
User exit with code:0.
System is shutting down with exit code 0.
```

4.2.2 分析过程

题目中通过 naive_malloc()函数进行内存的分配，naive_free()函数进行内存的回收，根据直接编译运行的输出结果，可以发现内存分配的相关功能已经实现，考虑到内存的分配与回收是互补事件，会以相反的顺序来调用相同的位置，因此先分析已实现完成的 naive_malloc()函数。

naive_malloc()函数与前述函数调用路径相同，都是通过 do_user_call()函数 ->ecall 指令 ->smode_trap_handler() 函数 ->handle_syscall() 函数 ->do_syscall()函数->sys_user_allocate_page()函数来执行系统调用。

```
uint64 sys_user_allocate_page() {
  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));

  return va;
}
```

可以发现 sys_user_allocate_page()函数先申请一个物理页，再通过 g_ufree_page 变量得到分配的逻辑地址，将 g_ufree_page 增加一个页大小后建立物理地址与逻辑地址间的映射，而 g_ufree_page 定义在 kernel/process.c 文件中，其初值 USER_FREE_ADDRESS_START 位于 kernel/memlayout.h 文件中：

```
uint64 g_ufree_page = USER_FREE_ADDRESS_START;

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

因此可以知道内存分配的初始逻辑地址为 USER_FREE_ADDRESS_START 即 4MB 开始的逻辑地址空间。

参考 naive_malloc()函数的调用过程，可以找到 naive_free()函数最终通过 sys_user_free_page()函数执行系统调用，而其调用的 user_vm_unmap()函数内即为 lab2_2 所需完成的位置。

```
void user_vm_unmap(pagetable_t page_dir, uint64 va, uint64 size, int
free) {
  // TODO (lab2_2): implement user_vm_unmap to disable the mapping of
the virtual pages
  // in [va, va+size], and free the corresponding physical pages used by
the virtual
  // addresses when if 'free' (the last parameter) is not zero.
  // basic idea here is to first locate the PTEs of the virtual pages,
and then reclaim
  // (use free_page() defined in pmm.c) the physical pages. lastly,
invalidate the PTEs.
  // as naive_free reclaims only one page at a time, you only need to
consider one page
  // to make user/app_naive_malloc to behave correctly.
  panic( "You have to implement user_vm_unmap to free pages using
naive_free in lab2_2.\n" );
}
```

根据 lab2_2 的 TODO 注释，需要将 panic 语句替换为 user_vm_unmap ()函数的具体实现，该函数输入页表、回收内存逻辑首地址、大小及是否回收，若需回收则将其对应的物理地址所对应的数据清零并将页表项有效位置零。

4.2.3 实验结果

分析完内存分配的过程后，反过来执行其对应操作即可实现内存回收：

找到一个给定 va 所对应的页表项 PTE

```
pte_t *PTE=page_walk(page_dir,va,0);
```

通过 PTE 计算 va 所对应物理页的首地址 pa

```
uint64 pa=(*PTE>>10)<<12);
```

回收 pa 对应的物理页（将内存对应地址对应的数据清零）

```
memset((void *)pa, 0, size);
```

将 PTE 中的 Valid 位（最低位）置为 0

```
if(*PTE<<63==1)
    *PTE-=1;
```

此实验标红部分与上传至 GitHub 上的不同，本实验报告内为正确代码：pa 应为物理页首地址无需加上页偏移、内存清零大小为函数参数的 size 而不是写死的 PGSIZE。但由于后续实验已继承该分支的修改内容，若再次修改可能出现冲突或错误，因此未在 git 内修复这两个错误。

将 panic 语句替换为

```
if(free){
  pte_t *PTE=page_walk(page_dir,va,0);
  uint64 pa=(*PTE>>10)<<12);
  memset((void *)pa, 0, size);
  if(*PTE<<63==1)
    *PTE-=1;
}
```

后，再次编译运行可回收内存并正常运行结束后退出。

4.3 lab2_3 缺页异常

4.3.1 实验内容

实验代码文件：user/app_sum_sequence.c

```
uint64 sum_sequence(uint64 n) {
  if (n == 0)
    return 0;
  else
    return sum_sequence( n-1 ) + n;
}

int main(void) {
  // we need a large enough "n" to trigger pagefaults in the user stack
```

```
  uint64 n = 1000;

  printu("Summation of an arithmetic sequence from 0 to %ld is: %ld \n",
n, sum_sequence(1000) );
  exit(0);
}
```

其功能为运用递归解法计算自然数数列 0 至 n 的和，通过递归函数 sum_sequence(n)将将求和问题转换为 sum_sequence(n-1)+n 的问题，并在 n=0 时返回 0，编译后直接运行报错：

You need to implement the operations that actually handle the page fault in lab2_3.

需要完善用户态栈空间的管理，在缺页中断时给程序分配新的页面，使得它能够正确处理用户进程的“压栈”请求，并获得以下预期结果：


```
$ 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:
0x000000008000e000, PKE kernel size: 0x000000000000e000 .
free physical memory address: [0x000000008000e000, 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): 0x0000000000010096
Switching 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.
```

4.3.2 分析过程

缺页中断与 lab1_2 的非法指令异常的分析过程类似，在

kernel/machine/minit.c 文件中的 delegate_traps() 函数中可以发现 CAUSE_STORE_PAGE_FAULT 即缺页中断代理给 S 模式进行处理。

```
static void delegate_traps() {
  if (!supports_extension('S')) {
    sprint("S mode is not supported.\n");
    return;
  }

  uintptr_t interrupts = MIP_SSIP | MIP_STIP | MIP_SEIP;
  uintptr_t exceptions = (1U << CAUSE_MISALIGNED_FETCH) | (1U <<
CAUSE_FETCH_PAGE_FAULT) |
                         (1U << CAUSE_BREAKPOINT) | (1U <<
CAUSE_LOAD_PAGE_FAULT) |
                         (1U << CAUSE_STORE_PAGE_FAULT) | (1U <<
CAUSE_USER_ECALL);

  write_csr(mideleg, interrupts);
  write_csr(medeleg, exceptions);
  assert(read_csr(mideleg) == interrupts);
  assert(read_csr(medeleg) == exceptions);
}
```

因此需要查看 S 模式中如何处理缺页中断，S 态的中断通过 kernel/strap.c 文件中的 smode_trap_handler() 函数处理，判断为缺页中断后就执行 handle_user_page_fault() 函数，而 handle_user_page_fault() 函数内即为 lab2_3 所需完成的位置。

```
void handle_user_page_fault(uint64 mcause, uint64 sepc, uint64 stval) {
  sprint("handle_page_fault: %lx\n", stval);
  switch (mcause) {
    case CAUSE_STORE_PAGE_FAULT:
      // TODO (lab2_3): implement the operations that solve the page
fault to
      // dynamically increase application stack.
      // hint: first allocate a new physical page, and then, maps the
new page to the
      // virtual address that causes the page fault.
      panic( "You need to implement the operations that actually handle the
page fault in lab2_3.\n" );

      break;
    default:
      sprint("unknown page fault.\n");
```

```
      break;
  }
}
```

根据 lab2_3 的 TODO 注释，需要处理因用户栈的增长导致的缺页中断，处理过程为先分配一个新的物理页，再将这个物理页映射到导致缺页中断的逻辑地址所对应的页上，而在本实验中默认输入的逻辑地址合法。

4.3.3 实验结果

分配一个新的物理页即调用 alloc_page()函数

页的映射则调用 user_vm_map()函数实现，其中需要将逻辑地址的页内偏移置零来获取其虚拟页的首地址：(stval>>12)<<12

将 panic 语句替换为

```
user_vm_map((pagetable_t)current->pagetable, (stval>>12)<<12, PGSIZE,
(uint64)alloc_page(), prot_to_type(PROT_WRITE | PROT_READ, 1));
break;
```

后，再次编译运行可正常处理缺页中断，在需要时为用户栈分配新的页面，处理完 3 次缺页中断后输出计算结果 500500。

5 进程管理

5.1 lab3_1 进程创建（fork）

5.1.1 实验内容

实验代码文件：user/app_naive_fork.c

```
int main(void) {
  uint64 pid = fork();
  if (pid == 0) {
    printu("Child: Hello world!\n");
  } else {
    printu("Parent: Hello world! child id %ld\n", pid);
  }

  exit(0);
}
```

其功能为主进程调用 fork()函数，通过系统调用基于主进程这个模板创建它的子进程，编译后直接运行报错：

```
You need to implement the code segment mapping of child in lab3_1.
```

需要完善 kernel/process.c 文件中的 do_fork()函数，将父进程的代码段映射到子进程的逻辑地址，使子进程正常创建并运行，并获得以下预期结果：

```
$ 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:
0x0000000080010000, PKE kernel size: 0x0000000000010000 .
free physical memory address: [0x0000000080010000, 0x000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080005000
kernel page table is on
Switching to user mode...
in alloc_proc. user frame 0x000000087fbc000, user stack
0x000000007ffff000, user kstack 0x000000087fbb000
User application is loading.
Application: ./obj/app_naive_fork
CODE_SEGMENT added at mapped info offset:3
Application program entry point (virtual address): 0x0000000000010078
```

```
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
do_fork map code segment at pa:0000000087fb2000 of parent to child at
va:0000000000010000.
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.
```

5.1.2 分析过程

fork()函数与前述函数调用路径相同，都是通过 do_user_call()函数->ecall 指令->smode_trap_handler()函数->handle_syscall()函数->do_syscall()函数->sys_user_fork()函数来执行系统调用，而其调用的do_fork()函数内即为 lab3_1 所需完成的位置。

```
int do_fork( process* parent)
{
  sprint( "will fork a child from parent %d.\n", parent->pid );
  process* child = alloc_process();

  for( int i=0; i<parent->total_mapped_region; i++ ){
    // browse parent's vm space, and copy its trapframe and data
segments,
    // map its code segment.
    switch( parent->mapped_info[i].seg_type ){
      case CONTEXT_SEGMENT:
        *child->trapframe = *parent->trapframe;
        break;
      case STACK_SEGMENT:
        memcpy( (void*)lookup_pa(child->pagetable,
child->mapped_info[0].va),
          (void*)lookup_pa(parent->pagetable,
parent->mapped_info[i].va), PGSIZE );
        break;
      case CODE_SEGMENT:
```

```
        // TODO (lab3_1): implment the mapping of child code segment to
parent's
        // code segment.
        // hint: the virtual address mapping of code segment is tracked
in mapped_info
        // page of parent's process structure. use the information in
mapped_info to
        // retrieve the virtual to physical mapping of code segment.
        // after having the mapping information, just map the
corresponding virtual
        // address region of child to the physical pages that actually
store the code
        // segment of parent process.
        // DO NOT COPY THE PHYSICAL PAGES, JUST MAP THEM.
        panic( "You need to implement the code segment mapping of child
in lab3_1.\n" );
        // after mapping, register the vm region (do not delete codes
below!)
        child->mapped_info[child->total_mapped_region].va =
parent->mapped_info[i].va;
        child->mapped_info[child->total_mapped_region].npages =
          parent->mapped_info[i].npages;
        child->mapped_info[child->total_mapped_region].seg_type =
CODE_SEGMENT;
        child->total_mapped_region++;
        break;
    }
  }

  child->status = READY;
  child->trapframe->regs.a0 = 0;
  child->parent = parent;
  insert_to_ready_queue( child );

  return child->pid;
}
```

根据 lab3_1 的 TODO 注释，需要将 panic 语句替换为子进程代码段的映射：父进程代码段的地址映射位于其进程结构的 mapped_info[]中，需要将子进程中对应的逻辑地址空间映射到其父进程中装载代码段的物理页面，并将权限设置为可读可执行，从而使子进程在无需复制代码段占用系统开销的条件下继承父进程代码段。

5.1.3 实验结果

页的映射需要使用 map_pages()函数写入子进程的页表 child->pagetable，被映射的逻辑地址通过将父进程的 mapped_info[]内的逻辑地址去除页内偏移后得到，物理地址则通过 lookup_pa()函数在父进程的页表中查询其逻辑地址对应的物理地址。

将 panic 语句替换为

```
map_pages(child->pagetable,((parent->mapped_info[i].va)<<12)>>12,
PGSIZE,lookup_pa(parent->pagetable, parent->mapped_info[i].va),
prot_to_type(PROT_READ | PROT_EXEC, 1));
```

后，再次编译运行可正常为子进程建立代码段的映射，使子进程正常创建并运行。

5.2 lab3_2 进程 yield

5.2.1 实验内容

实验代码文件：user/app_yield.c

```
#include "user/user_lib.h"
#include "util/types.h"

int main(void) {
  uint64 pid = fork();
  uint64 rounds = 0xffff;
  if (pid == 0) {
    printu("Child: Hello world! \n");
    for (uint64 i = 0; i < rounds; ++i) {
      if (i % 10000 == 0) {
        printu("Child running %ld \n", i);
        yield();
      }
    }
  } else {
    printu("Parent: Hello world! \n");
    for (uint64 i = 0; i < rounds; ++i) {
      if (i % 10000 == 0) {
        printu("Parent running %ld \n", i);
        yield();
      }
    }
```

```
  }

  exit(0);
  return 0;
}
```

其功能为程序通过 fork()函数创建了一个子进程，父进程和子进程都执行一个很长的 for 循环，且两进程每执行 10000 次循环后都会调用 yield()释放 CPU 的执行权，从而使父子进程可以轮流使用 CPU，编译后直接运行报错：

You need to implement the yield syscall in lab3_2.

需要完善 yield 系统调用，实现进程执行过程中的主动释放 CPU 的动作，并获得以下预期结果：


```
$ spike ./obj/riscv-pke ./obj/app_yield
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end:
0x0000000080010000, PKE kernel size: 0x0000000000010000 .
free physical memory address: [0x0000000080010000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080005000
kernel page table is on
Switching to user mode...
in alloc_proc. user frame 0x0000000087fbc000, user stack
0x000000007ffff000, user kstack 0x0000000087fbb000
User application is loading.
Application: ./obj/app_yield
CODE_SEGMENT added at mapped info offset:3
Application program entry point (virtual address): 0x000000000001017c
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
do_fork map code segment at pa:0000000087fb2000 of parent to child at
va:0000000000010000.
going to insert process 1 to ready queue.
Parent: Hello world!
Parent running 0
```

```
going to insert process 0 to ready queue.
going to schedule process 1 to run.
Child: Hello world!
Child running 0
going to insert process 1 to ready queue.
going to schedule process 0 to run.
Parent running 10000
going to insert process 0 to ready queue.
going to schedule process 1 to run.
Child running 10000
going to insert process 1 to ready queue.
going to schedule process 0 to run.
Parent running 20000
going to insert process 0 to ready queue.
going to schedule process 1 to run.
Child running 20000
going to insert process 1 to ready queue.
going to schedule process 0 to run.
Parent running 30000
going to insert process 0 to ready queue.
going to schedule process 1 to run.
Child running 30000
going to insert process 1 to ready queue.
going to schedule process 0 to run.
Parent running 40000
going to insert process 0 to ready queue.
going to schedule process 1 to run.
Child running 40000
going to insert process 1 to ready queue.
going to schedule process 0 to run.
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.
```

5.2.2 分析过程

yield()函数与前述函数调用路径相同，都是通过 do_user_call()函数->ecall 指令->smode_trap_handler()函数->handle_syscall()函数->do_syscall()函数->sys_user_yield()函数来执行系统调用，而sys_user_yield()函数内即为 lab3_2 所需完成的位置。

```
ssize_t sys_user_yield() {
  // TODO (lab3_2): implment the syscall of yield.
  // hint: the functionality of yield is to give up the processor.
therefore,
  // we should set the status of currently running process to READY,
insert it in
  // the rear of ready queue, and finally, schedule a READY process to
run.
  panic( "You need to implement the yield syscall in lab3_2.\n" );
  return 0;
}
```

根据 lab3_2 的 TODO 注释，需要将 panic 语句替换为进程的 yield，即释放 CPU 执行权，其操作为将当前进程置为就绪状态（READY）、将当前进程加入到就绪队列的队尾、转进程调度。

5.2.3 实验结果

将当前进程置为就绪状态：current->status=READY

将当前进程加入到就绪队列的队尾：insert_to_ready_queue(current)

转进程调度：schedule()

将 panic 语句替换为

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

后，再次编译运行可进行进程的 yield，使父子进程可以交替使用 CPU 并正确输出相关信息。

5.3 lab3_3 循环轮转调度

5.3.1 实验内容

实验代码文件：user/app_two_long_loops.c

```
#include "user/user_lib.h"
#include "util/types.h"

int main(void) {
  uint64 pid = fork();
  uint64 rounds = 100000000;
  uint64 interval = 10000000;
  uint64 a = 0;
  if (pid == 0) {
    printu("Child: Hello world! \n");
    for (uint64 i = 0; i < rounds; ++i) {
      if (i % interval == 0) printu("Child running %ld \n", i);
    }
  } else {
    printu("Parent: Hello world! \n");
    for (uint64 i = 0; i < rounds; ++i) {
      if (i % interval == 0) printu("Parent running %ld \n", i);
    }
  }

  exit(0);
  return 0;
}
```

其功能与 lab3_2 类似程序通过 fork()函数创建了一个子进程，父进程和子进程都执行一个很长的 for 循环，且两进程每执行 10000000 次循环后都会显示当前循环次数，但不会主动调用 yield()释放 CPU 的执行权，因此会导致某个进程长期占据 CPU，而另一个进程无法得到执行，编译后直接运行报错：

You need to further implement the timer handling in lab3_3.

需要利用时钟中断来实现进程的循环轮转调度实现 kernel/strap.c 文件中的 rrsched()函数，并获得以下预期结果：

```
$ spike ./obj/riscv-pke ./obj/app_two_long_loops
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
```

```
PKE kernel start 0x0000000080000000, PKE kernel end:
0x0000000080010000, PKE kernel size: 0x0000000000010000 .
free physical memory address: [0x0000000080010000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080005000
kernel page table is on
Switching to user mode...
in alloc_proc. user frame 0x0000000087fbc000, user stack
0x000000007ffff000, user kstack 0x0000000087fbb000
User application is loading.
Application: ./obj/app_two_long_loops
CODE_SEGMENT added at mapped info offset:3
Application program entry point (virtual address): 0x000000000001017c
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
do_fork map code segment at pa:0000000087fb2000 of parent to child at
va:0000000000010000.
going to insert process 1 to ready queue.
Parent: Hello world!
Parent running 0
Parent running 10000000
Ticks 0
Parent running 20000000
Ticks 1
going to insert process 0 to ready queue.
going to schedule process 1 to run.
Child: Hello world!
Child running 0
Child running 10000000
Ticks 2
Child running 20000000
Ticks 3
going to insert process 1 to ready queue.
going to schedule process 0 to run.
Parent running 30000000
Ticks 4
Parent running 40000000
Ticks 5
going to insert process 0 to ready queue.
```

```
going to schedule process 1 to run.
Child running 30000000
Ticks 6
Child running 40000000
Ticks 7
going to insert process 1 to ready queue.
going to schedule process 0 to run.
Parent running 50000000
Parent running 60000000
Ticks 8
Parent running 70000000
Ticks 9
going to insert process 0 to ready queue.
going to schedule process 1 to run.
Child running 50000000
Child running 60000000
Ticks 10
Child running 70000000
Ticks 11
going to insert process 1 to ready queue.
going to schedule process 0 to run.
Parent running 80000000
Ticks 12
Parent running 90000000
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.
```

5.3.2 分析过程

为了实现进程的轮转，避免单个进程长期霸占 CPU 的情况，可以在时钟中断时进行进程调度。本题中在 kernel/sched.h 文件中定义了时间片的长度（TIME_SLICE_LEN）为 2，即每个进程在一个时间片（2 次时钟中断）后若仍未

停止，则触发触发一次进程重新调度动作，以实现循环轮转调度。

```
//length of a time slice, in number of ticks
#define TIME_SLICE_LEN  2
```

为配合调度的实现，在进程结构中定义了整型成员 tick_count，用以记录当前进程在本次运行时间内所经历的时钟中断次数，并在 ernel/strap.c 文件中的 rrsched()函数里进行具体操作。

```
void rrsched() {
  // TODO (lab3_3): implements round-robin scheduling.
  // hint: increase the tick_count member of current process by one, if it is bigger than
  // TIME_SLICE_LEN (means it has consumed its time slice), change its status into READY,
  // place it in the rear of ready queue, and finally schedule next process to run.
  panic( "You need to further implement the timer handling in lab3_3.\n" );
}
```

根据 lab3_3 的 TODO 注释，需要将 panic 语句替换为判断是否轮转时间片，其操作为判断当前进程的 tick_count 加 1 后是否大于等于 TIME_SLICE_LEN，若是则将当前进程的 tick_count 清零，并将当前进程加入就绪队列，转进程调度，否则将当前进程的 tick_count 加 1，并返回。

5.3.3 实验结果

判断当前进程的 tick_count 加 1 后是否大于等于 TIME_SLICE_LEN：

```
if (++current->tick_count>=TIME_SLICE_LEN)
```

若是则将当前进程的 tick_count 清零，并将当前进程加入就绪队列，转进程调度：

```
current->tick_count=0;
current->status=READY;
insert_to_ready_queue(current);
schedule();
```

否则将当前进程的 tick_count 加 1，并返回：判断时已将 tick_count 加 1，直接返回即可。

将 panic 语句替换为

```
if (++current->tick_count>=TIME_SLICE_LEN)
{
  current->tick_count=0;
```

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

后，再次编译运行可进行时间片长度为 2 个时钟中断的循环轮转调度，使父子进程可以交替使用 CPU 并正确输出相关信息。

6 挑战实验

6.1 选题介绍

lab1 的挑战实验需要熟悉内核模式切换的具体过程与 ELF 文件结构，由于在操作系统理论课程中仅对模式的概念进行了解而并未探究其过程，且在本次操作系统课程设计前也并未接触过 ELF 文件，因此未选择 lab1 中的挑战实验。lab2、lab3 的挑战实验知识要点在内存管理中的地址转换、分页与进程状态变迁的相关内容，在理论课程中学习过具体的实现过程，对实验中所需功能有大致的了解，因此选择完成 lab2_challenge1、lab3_challenge1 两个挑战实验。

6.2 lab2_challenge1 复杂缺页异常

6.2.1 实验内容

实验代码文件：user/app_sum_sequence.c

```
#include "user_lib.h"
#include "util/types.h"

uint64 sum_sequence(uint64 n, int *p) {
  if (n == 0)
    return 0;
  else
    return *p=sum_sequence( n-1, p+1 ) + n;
}

int main(void) {
  uint64 n = 1024;
  int *ans = (int *)naive_malloc();

  printu("Summation of an arithmetic sequence from 0 to %ld is: %ld \n",
n+1, sum_sequence(n+1, ans) );

  exit(0);
}
```

程序运用递归解法计算自然数数列 0 至 n 的和，函要求将每一步递归的结果保存在数组 ans 中。创建数组时，我们使用了当前的 malloc 函数申请了一个页面（4KB）的大小，对应可以存储的个数上限为 1024。在函数调用时，我们试图计算 1025 求和，首先由于 n 足够大，所以在函数递归执行时会触发用户栈的缺

页，你需要对其进行正确处理，确保程序正确运行；其次，1025 在最后一次计算时会访问数组越界地址，由于该处虚拟地址尚未有对应的物理地址映射，因此属于非法地址的访问，这是不被允许的，对于这种缺页异常，应该提示用户并退出程序执行。

本实验的具体要求为：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。

你对内核代码的修改可能包含以下内容：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。

6.2.2 原理分析

本实验需要判断并处理两类缺页异常，对于函数递归压栈导致的用户栈空间缺页，为其分配新页并建立映射，此类缺页已在 lab2_3 中进行处理，若为数组越界访问非法虚拟地址（也表现为缺页），需要提示异常并停止运行，因此需要在 lab2_3 的基础上完善缺页异常的判断。

在 lab2_3 中默认逻辑地址合法，因此仅判断逻辑地址小于用户栈顶地址且缺页时即为其分配新页建立映射。考虑两种缺页所访问的逻辑地址：用户栈空间总是由 USER_STACK_TOP 栈顶向下扩展，即 0x80000000 下方空间，而由于用户代码段使用的为低位的逻辑地址，为 0x10000 上方空间。需要针对这一区别进行判断。

6.2.3 实现过程

在 lab2_3 中，给出了一个思路：比我们预设的可能的用户栈的最小栈底指针要大（这里，我们可以给用户栈空间一个上限，例如 20 个 4KB 的页面），若满足，则为合法的逻辑地址。即判断所访问地址是否大于用户栈底指针，而对于不同的应用，所需的用户栈空间不同，也存在正常运行却超过了预设的固定页面限制的可能性，因此需要动态判断栈底指针。

对于用户栈压栈，单次入栈内容一定低于页面大小（4KB），因此可以通过已分配的页面数来计算当前正常访问所需满足的栈底指针地址要求：为进程结构新增 stack_map_page_num 项表示当前用户栈由于缺页已分配的页数，并在每次分配用户栈的页面时将此变量+1。由于在进程初始化时也会为用户栈分配一个页面，

因此用户栈实际已分配的页数为 stack_map_page_num+1，在考虑新的用户栈缺页时，还允许由于缺页为其分配一个新的页面，因此认为用户栈允许的最大大小为(stack_map_page_num+2) *PGSIZE。

根据上述分析，可以计算得到用户栈底指针为 USER_STACK_TOP-(current->stack_map_page_num+2)*PGSIZE，判断所访问的地址是否大于该动态计算的栈底指针

```
stval>=USER_STACK_TOP-(current->stack_map_page_num+2)*PGSIZE
```

即可判断是否为合法地址。

6.2.4 实验结果

本实验完整的实现内容如下（未写出头文件新增的函数声明）：

kernel/process.h 文件：

```
typedef struct process_t {
  int stack_map_page_num;
}process;
```

kernel/strap.c 文件：

```
if(stval<USER_STACK_TOP){
    if(stval>=USER_STACK_TOP-(current->stack_map_page_num+2)*PGSIZE){
        user_vm_map((pagetable_t)current->pagetable, (stval>>12)<<12,
PGSIZE, (uint64)alloc_page(), prot_to_type(PROT_WRITE | PROT_READ, 1));
        current->stack_map_page_num++;
    }
    else
        panic("this address is not available!");
}
```

完成后，重新编译运行可以获得正确的输出：

```
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): 0x00000000000100a2
Switch to user mode...
handle_page_fault: 000000007fffdff8
handle_page_fault: 000000007fffcff8
handle_page_fault: 000000007fffbff8
handle_page_fault: 000000007fffaff8
handle_page_fault: 000000007fff9ff8
handle_page_fault: 000000007fff8ff8
handle_page_fault: 000000007fff7ff8
handle_page_fault: 000000007fff6ff8
handle_page_fault: 0000000000401000
this address is not available!
System is shutting down with exit code -1.
```

6.3 lab3_challenge1 进程等待和数据段复制

6.3.1 实验内容

实验代码文件：user/app_wait.c

```c
#include "user/user_lib.h"
#include "util/types.h"

int flag;
int main(void) {
    flag = 0;
    int pid = fork();
    if (pid == 0) {
        flag = 1;
        pid = fork();
        if (pid == 0) {
            flag = 2;
            printu("Grandchild process end, flag = %d.\n", flag);
        } else {
            wait(pid);
            printu("Child process end, flag = %d.\n", flag);
        }
    } else {
        wait(-1);
        printu("Parent process end, flag = %d.\n", flag);
    }
```

```
    exit(0);
    return 0;
}
```

在以上程序中，父进程把 flag 变量赋值为 0，然后 fork 生成一个子进程，接着通过 wait 函数等待子进程的退出。子进程把自己的变量 flag 赋值为 1，然后 fork 生成孙子进程，接着通过 wait 函数等待孙子进程的退出。孙子进程给自己的变量 flag 赋值为 2 并在退出时输出信息，然后子进程退出时输出信息，最后父进程退出时输出信息。由于 fork 之后父子进程的数据段相互独立（同一虚拟地址对应不同的物理地址），子进程对全局变量的赋值不影响父进程全局变量的值。

本实验的具体要求为：

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，wait 函数接受一个参数 pid：

当 pid 为-1 时，父进程等待任意一个子进程退出即返回子进程的 pid；

当 pid 大于 0 时，父进程等待进程号为 pid 的子进程退出即返回子进程的 pid；

如果 pid 不合法或 pid 大于 0 且 pid 对应的进程不是当前进程的子进程，返回-1。

补充 do_fork 函数，实验 3_1 实现了代码段的复制，你需要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。

注意：最终测试程序可能和给出的用户程序不同，但都只涉及 wait 函数、fork 函数和全局变量读写的相关操作。

你对内核代码的修改可能包含添加系统调用、在内核中实现 wait 函数的功能以及对 do_fork 函数的完善。

6.3.2 原理分析

本实验的 wait()函数用于让父进程等待指定的子进程运行完毕，获取所需的子进程相关数据后继续运行。考虑到理论课程中所学的进程状态与变迁过程，可以想到其过程为父进程在执行 wait()函数后进入阻塞状态，而所需的子进程运行结束后将该父进程唤醒为就绪状态。因此除了实现 wait()函数对父进程的阻塞，还要修改进程退出时的相关函数执行唤醒操作。

补充的 do_fork()函数需要实现数据段的复制并保证父子进程数据独立性，因此在参考代码段的复制实现上，考虑数据段与代码段在程序运行中的差别进行修改。

6.3.3 实现过程

首先完善 do_fork()函数实现数据段的复制，lab3_1 中完善的代码段复制相关代码如下：

```
case CODE_SEGMENT:
    map_pages(child->pagetable,((parent->mapped_info[i].va)<<12)>>12,PGS
IZE,lookup_pa(parent->pagetable,
parent->mapped_info[i].va),prot_to_type(PROT_READ | PROT_EXEC, 1));
    child->mapped_info[child->total_mapped_region].va =
parent->mapped_info[i].va;
    child->mapped_info[child->total_mapped_region].npages =
parent->mapped_info[i].npages;
    child->mapped_info[child->total_mapped_region].seg_type =
CODE_SEGMENT;
    child->total_mapped_region++;
    break;
```

考虑到数据段与代码段在程序运行中的差别，需要修改上述代码中标红的内容：

1、新建一个 DATA_SEGMENT 的 case，用以判断复制数据段并执行相应操作。

2、由于父子进程数据相互独立且可能分别进行不同的修改，子进程的数据段逻辑地址不能像代码段一样直接映射父进程的物理地址来节省空间，而是需要将父进程的数据段复制到一个新的页中，并建立逻辑地址物理地址的映射。

3、代码段的权限为读与执行，而数据段的权限应为读与写。

4、数据段的类型为 DATA_SEGMENT。

然后考虑 wait()函数，与 fork()函数类似，也要通过 do_user_call()函数执行系统调用。在 kernel/syscall.h 中加入 SYS_user_wait，并在 user/user_lib.c 中参照 fork()函数进行实现，同时也在 kernel/syscall.c 中加入 SYS_user_wait 的判断，返回 do_wait()函数的返回值，其参数为 pid。

```
int wait(int pid){
  return do_user_call(SYS_user_wait, pid, 0, 0, 0, 0, 0, 0);
}
```

wait()函数需要阻塞父进程并（在父进程被唤醒后）返回子进程相应数据。

因此，（子）进程需要得知当前是否有（父）进程在等待自己的执行完毕，父进程也需要在子进程结束后获取并返回其 pid。为了保存这两个数据，需要修改进程的结构为其添加两项：

waitpid：表示父进程正在等待的子进程的 pid，根据题目要求，wait()函数的 pid 参数为-1（任意子进程）或 0-31（对应 pid 子进程）。但 process 结构体中 waitpid 默认初始化为 0，且进程具有无需等待子进程的状态，因此约定 waitpid 的值为-1 时为等待任意子进程，为 0 时为无需等待子进程，为 1-32 时分别代表等待 pid 为 0-31 的子进程（即 waitpid=pid+1）。

returnpid：wait()函数需要返回等待成功的子进程 pid，而此时该子进程已经结束，因此子进程在唤醒父进程时将自己的 pid 写入到父进程的 returnpid 项中供父进程的 wait()函数返回，returnpid 与 pid 数值为一一对应的映射（即 returnpid=pid）。

新增了进程结构后，可以开始具体实现 do_wait()函数。根据题目要求，在 pid 为-1 时父进程等待任意一个子进程退出即返回子进程的 pid，因此遍历所有进程，若存在子进程则将其 waitpid 设为 1，设为阻塞状态并转进程调度，（在任意子进程唤醒父进程后）返回 returnpid：

```
if (pid == -1) {
  for (int i = 0; i < NPROC; i++) {
    if (procs[i].parent == current) {
      current->waitpid=-1;
      current->status=BLOCKED;
      schedule();
      return current->returnpid;
    }
  }
}
```

若 pid 不合法，即非-1 或有效进程号或对应进程号非自身子进程则返回-1：

```
if (pid < -1 || pid > NPROC || procs[pid].parent != current) return -1;
```

其余情况均为合法的子进程 pid，按照上述 waitpid 规则将其设为 pid+1，设为阻塞状态并转进程调度，在其唤醒父进程后）返回自身 pid：

```
current->waitpid=pid+1;
current->status=BLOCKED;
schedule();
return pid;
```

父进程的唤醒设在进程结束处的 sys_user_exit()函数实现，若进程结束时判断存在状态为阻塞的父进程，且父进程的 waitpid 为-1 或自身 pid+1（等待任意子进程或自身），则将父进程的 returnpid 设为自身 pid，waitpid 清零，并将其状态设为就绪插入到就绪队列中以激活父进程：

```
if(current->parent!=NULL && current->parent->status==BLOCKED){
  if(current->parent->waitpid==-1 ||
current->parent->waitpid==current->pid+1){
    current->parent->returnpid=current->pid;
    current->parent->waitpid=0;
    current->parent->status=READY;
    insert_to_ready_queue(current->parent);
  }
}
```

6.3.4 实验结果

本实验完整的实现内容如下（未写出头文件新增的函数声明）：

kernel/process.c 文件：

```
case DATA_SEGMENT:
    map_pages(child->pagetable,((parent->mapped_info[i].va)<<12)>>12,PGS
IZE,(uint64)alloc_page(),prot_to_type(PROT_READ | PROT_WRITE, 1));
    child->mapped_info[child->total_mapped_region].va =
parent->mapped_info[i].va;
    child->mapped_info[child->total_mapped_region].npages =
parent->mapped_info[i].npages;
    child->mapped_info[child->total_mapped_region].seg_type =
DATA_SEGMENT;
    child->total_mapped_region++;
    break;

int do_wait(int pid) {
  if (pid == -1) {
    for (int i = 0; i < NPROC; i++) {
      if (procs[i].parent == current) {
        current->waitpid=-1;
        current->status=BLOCKED;
        schedule();
        return current->returnpid;
      }
    }
  }
```

```
  if (pid < -1 || pid > NPROC || procs[pid].parent != current) return -
1;
  current->waitpid=pid+1;
  current->status=BLOCKED;
  schedule();
  return pid;
}
```

kernel/process.h 文件：

```
typedef struct process_t {
  int waitpid;
  int returnpid;
}process;
```

kernel/syscall.c 文件：

```
case SYS_user_wait:
  return sys_user_wait(a1);

ssize_t sys_user_wait(int pid) {
  return do_wait(pid);
}

if(current->parent!=NULL && current->parent->status==BLOCKED){
  if(current->parent->waitpid==-1 ||
current->parent->waitpid==current->pid+1){
    current->parent->returnpid=current->pid;
    current->parent->waitpid=0;
    current->parent->status=READY;
    insert_to_ready_queue(current->parent);
  }
}
```

kernel/syscall.h 文件：

```
#define SYS_user_wait (SYS_user_base + 6)
```

user/user_lib.c 文件：

```
int wait(int pid){
  return do_user_call(SYS_user_wait, pid, 0, 0, 0, 0, 0, 0);
}
```

完成后，重新编译运行可以获得正确的输出：

```
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end:
0x0000000080009000, PKE kernel size: 0x0000000000009000 .
```

```
free physical memory address: [0x0000000080009000, 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_wait
CODE_SEGMENT added at mapped info offset:3
DATA_SEGMENT added at mapped info offset:4
Application program entry point (virtual address): 0x00000000000100b0
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 0x0000000087fae000, user stack
0x000000007ffff000, user kstack 0x0000000087fad000
going to insert process 1 to ready queue.
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, user kstack 0x0000000087fa0000
going to insert process 2 to ready queue.
going to schedule process 2 to run.
Grandchild process end, flag = 2.
User exit with code:0.
going to insert process 1 to ready queue.
going to schedule process 1 to run.
Child process end, flag = 1.
User exit with code:0.
going to insert process 0 to ready queue.
going to schedule process 0 to run.
Parent process end, flag = 0.
User exit with code:0.
no more ready processes, system shutdown now.
System is shutting down with exit code 0.
```

参考文献

[1]操作系统内核实验（2022 年春） https://shimo.im/docs/vVqRVplzMbIWQLqy

[2]郑鹏，曾平，金晶编著. 计算机操作系统 第 2 版. 武汉：武汉大学出版社，2014.07.

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。）