

华中科技大学

操作系统原理课程设计报告

姓 名:	徐锦慧
学 院:	计算机科学与技术
专 业:	计算机科学与技术
班 级:	CS2011
学 号:	U202011675
指导教师:	阳富民

分数	
教师签名	

2023 年 03 月 07 日

目 录

1 LAB1_CHALLENGE1	1
1.1 实验目的	1
1.2 实验内容	1
1.3 实验调试及心得	4
2 LAB1_CHALLENGE2	6
2.1 实验目的	6
2.2 实验内容	6
2.3 实验调试及心得	8
3 LAB2_CHALLENGE2	10
3.1 实验目的	10
3.2 实验内容	10
3.3 实验调试及心得	13
4 LAB3_CHALLENGE2	15
4.1 实验目的	15
4.2 实验内容	15
4.3 实验调试及心得	18

1 lab1_challenge1

1.1 实验目的

1. 实验要求

本实验涉及用户态栈、ELF 文件等内容，给定引用 `app_print_backtrace.c` 文件，需要实现 `print_backtrace()` 函数，通过输入参数控制回溯的层数，并依次打印用户程序调用的函数。具体的实验要求为：

- 1) 完善系统调用路径。
- 2) 进入操作系统内核后，找到用户进程的用户态栈来开始回溯。
- 3) 通过用户栈找到函数的返回地址，将虚拟地址转换为源程序中的符号后打印输出。

2. 实验目的

- 1) 了解函数调用时栈帧的结构。
- 2) 了解 ELF 文件的结构，熟悉符号节、字符串节的存储内容及格式。

1.2 实验内容

1. 分析给定应用

首先观察给定应用 `app_print_backtrace.c` 文件。关键代码如下：

```
void f8() { print_backtrace(7); }
void f7() { f8(); }
void f6() { f7(); }
void f5() { f6(); }
void f4() { f5(); }
void f3() { f4(); }
void f2() { f3(); }
void f1() { f2(); }
```

用户程序在执行时，函数调用关系如下：

`main -> f1 -> f2 -> f3 -> f4 -> f5 -> f6 -> f7 -> f8`

`print_backtrace(7)` 的作用是将以上用户程序的函数调用关系，从最后的 `f8` 向上打印 7 层，预期的输出为：`f8 f7 f6 f5 f4 f3 f2`。

2. 完善系统调用路径

首先在 `user_lib.h` 文件中添加 `print_backtrace()` 函数的原型，参数 `length` 表示回溯的层数。代码如下：

```
int print_backtrace(int length);
```

接着在 user_lib.c 文件中添加 print_backtrace()函数的实现，代码如下：

```
int print_backtrace(int length){  
    return do_user_call(SYS_user_backtrace, length, 0, 0, 0, 0, 0, 0);  
}
```

在 syscall.h 文件中添加 SYS_user_backtrace 的注册，代码如下：

```
#define SYS_user_backtrace (SYS_user_base + 2)
```

相应地，在 syscall.c 文件中添加 SYS_user_backtrace 系统调用的实现，代码如下：

```
case SYS_user_backtrace:  
    return sys_user_backtrace(a1);
```

以上均为添加进程回溯的系统调用部分，完成这些全局变量的定义后，在 sys_user_backtrace() 函数中实现真正的回溯功能。

3. 符号解析

由于获取函数名涉及到 ELF 文件中存储的符号节，我们在 elf.h 文件中增加了符号表的数据结构，用于表示符号节中每个符号的具体信息，代码如下：

```
typedef struct elf_symbol_t{  
    uint32 name;          /* Symbol name */  
    unsigned char info;   /* Symbol type and binding */  
    unsigned char other;  /* Symbol visibility */  
    uint16 shndx;         /* Section index */  
    uint64 value;         /* Symbol value */  
    uint64 size;          /* Symbol size */  
} elf_symbol;
```

其中，info 存储每个符号的类型和绑定属性；value 给出每个符号的值，在可重定位目标文件中指符号所在位置的字节偏移量；name 为符号名在字符串表中的索引值；size 表示符号名对应的虚地址空间的大小。接着我们在 elf_ctx 结构中添加 elf_symbol 类型的数组，用于存储一个 ELF 文件中的所有符号，并定义了一个 elf_ctx 类型的公共变量 elfloader，用于后续查找函数符号名。

为了实现对程序调用函数的回溯，我们还需要将.symtab 和.strtab 两个节的内容加载到主存中，目的地址放在 elfloader 内，通过 elfloader 就可以在回溯时轻松获取 ELF 文件中每个符号的值，进而打印出函数名。关键代码如下：

```
for (int i = 0, j = ctx->ehdr.shoff; i < ctx->ehdr.shnum; i+=1, j +=  
sizeof(elf_sect_header)) {  
    // load the content of elf file into memory
```

```

    if(elf_fpread(ctx, (void*)&sect_header, sizeof(sect_header), j) !=
sizeof(sect_header)) return EL_EIO;
    if (sect_header.type == ELF_STRTAB) { // string table
        if (elf_fpread(ctx, &ctx->strtab + count * sect_header.size,
sect_header.size, sect_header.offset) != sect_header.size) return EL_EIO;
        count++;
    }
    else if (sect_header.type == ELF_SYMTAB) { // symbol table
        if(elf_fpread(ctx, ctx->symbol, sect_header.size,
sect_header.offset) != sect_header.size) return EL_EIO;
        else ctx->sym_num = sect_header.size / sizeof(elf_symbol);
    }
}
}

```

4. 获取函数返回地址

我们在函数 `sys_user_backtrace()` 中实现用户态栈和函数返回地址的获取。

由于该函数在系统调用时处于内核态，用户态栈必须通过 `trapframe` 中存储的进程上下文获取，即通过 `current->trapframe->regs.sp` 来获取当前用户态栈的栈底。我们将 `app_print_backtrace.c` 文件反汇编观察函数的栈帧结构，由图 1.1 可知，返回地址 `ra` 距离 `sp` 的长度为 8+16 个字节，因此回溯过程的第一个返回地址为 `current->trapframe->regs.sp + 24`。

```

xjh@ubuntu:~/Desktop/riscv-pke$ riscv64-unknown-elf-objdump -d obj/app_print_backtrace
obj/app_print_backtrace:      file format elf64-littleriscv

Disassembly of section .text:

0000000081000000 <f8>:
81000000: 1141          addi    sp,sp,-16
81000002: e406          sd      ra,8(sp)
81000004: e022          sd      s0,0(sp)
81000006: 0800          addi    s0,sp,16
81000008: 451d          li      a0,7
8100000a: 160000ef     jal     ra,8100016a <print_backtrace>
8100000e: 60a2          ld      ra,8(sp)
81000010: 6402          ld      s0,0(sp)
81000012: 0141          addi    sp,sp,16
81000014: 8082          ret

```

图 1.1 反汇编结果

接着通过第一个返回地址和输入的层数回溯，依次获得各个函数调用的返回地址，并通过对比函数的逻辑地址范围和符号表来反向解析 `ra` 对应的函数名称，输出所需结果。具体的回溯流程图如图 1.2 所示：

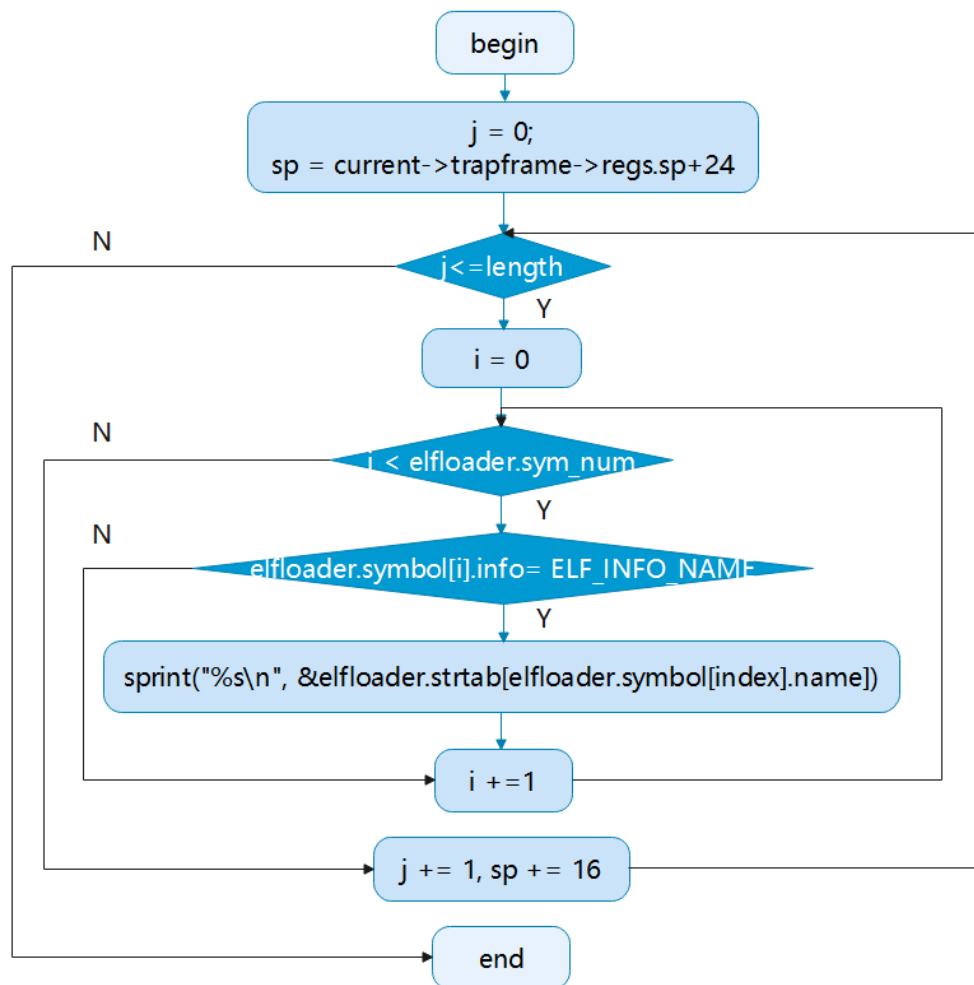


图 1.2 回溯流程图

1.3 实验调试及心得

1. 实验调试

最初按照以上分析实现对应函数，得到如图 1.3 所示的结果：

```

xjh@ubuntu:~/Desktop/riscv-pke$ 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
Misaligned Load!
System is shutting down with exit code -1.
  
```

图 1.3 访问非法地址错误

经过检查发现，是由于非法访问地址导致输出了“Misaligned Load!”的报错。继续分析该报错的来源，最终修改完得到正确结果，如图 1.4 所示：

```
xjh@ubuntu:~/Desktop/riscv-pke$ 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:
vsnprintf
f8
f7
f6
f5
f4
f3
f2
User exit with code:0.
System is shutting down with exit code 0.
```

图 1.4 正确输出

2. 心得体会

本次操作系统实验的主要目的是加深对操作系统中函数调用时栈帧结构和 ELF 文件结构的理解，并掌握符号节和字符串节的存储内容和格式。

通过实验，我对函数调用时栈帧结构有了更深入的了解，包括栈帧的组成结构、栈帧中不同部分的作用和存储方式。同时，我也掌握了 ELF 文件的结构和常见的符号节、字符串节的存储内容和格式。这些知识对于理解操作系统的底层机制、编译链接过程以及代码的运行和调试都非常重要。

在实验中，我通过分析给定的 ELF 文件，学会了如何进行符号解析和函数栈帧结构的分析，并通过注释、参数等推断函数的作用，进一步完善系统调用路径。同时，我也遇到了一些问题，例如非法访问地址等错误，但是通过查找资料 and 与同学交流讨论，我成功解决了这些问题。这些实践和解决问题的经验也有助于我在今后的操作系统开发和调试过程中更加熟练地运用这些知识和技能。

总之，本次实验让我深入理解了操作系统中的函数调用栈帧结构和 ELF 文件的结构，加强了我对操作系统的理论认识和分析能力，有助于我更好地理解操作系统的内部机制，提高编程和调试能力。

在建议方面，希望后续可以简化头歌平台上的提交测评。由于平台中需要修改的代码文件较多，很容易在切换文件时漏改、错改；并且每个关卡独立，不能继承前面的代码，每次都需要重新复制粘贴。当然，目前的测评流程已经做的很到位，如果能够继承其他关卡的代码就更好了。

2 lab1_challenge2

2.1 实验目的

1. 实验要求

本实验为挑战实验，给定应用 `user/app_errorline.c`，需要修改内核的代码，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。具体的实验要求为：

- 1) 修改读取 `elf` 文件的代码，找到包含调试信息的段，将其内容保存起来。
- 2) 在适当的位置调用 `debug_line` 段解析函数，对调试信息进行解析，构造指令地址-源代码行号-源代码文件名的对应表。
- 3) 在异常中断处理函数中，通过相应寄存器找到触发异常的指令地址，然后在上述表中查找地址对应的代码行号和文件名。

2. 实验目的

- 1) 熟悉 ELF 文件的结构，能够在 ELF 文件中找到所需数据。
- 2) 了解调试信息格式 DWARF。
- 3) 了解 PKE 系统的异常处理过程。

2.2 实验内容

1. 分析给定应用

首先观察给定应用 `app_errorline.c` 文件，主要代码如下：

```
int main(void) {
    printf("Going to hack the system by running privilege instructions.\n");
    asm volatile("csrw sscratch, 0");
    exit(0);
}
```

在以上程序中，`asm volatile("csrw sscratch, 0")`语句试图在用户态读取在内核态才能读取的寄存器 `sscratch`，因此此处会触发 `illegal instruction` 异常，需要修改内核代码，输出该文件名和代码行。

2. 寻找调试信息段

我们对 `elf_load` 函数进行修改。首先用 `max_va` 变量存储程序所有需映射的段数据的最大虚拟地址，其计算方法如下：

```
max_va = max_va > (ph_addr.vaddr+ph_addr.memsz) ? max_va :
(ph_addr.vaddr+ph_addr.memsz);
```


接着需要找到 `debug_line` 段，把它保存起来并调用所给的解析函数。我们先读取一个专门存储各段名称的段的段头 `str_tab`。接着枚举所有段，读取段头 `sh_addr`，利用 `sh_addr` 中包含的段名索引和前面得到的 `str_tab`，从地址 `str_tab.offset + sh_addr.name` 处读取该段名称。如果段名为 “.debug_line”，则把这个段的实际数据读下来放到 `max_va` 处，并对该数据调用 `make_addr_line()` 函数进行解析。关键代码如下：

```
elf_sct_header str_tab, sh_addr;
char name[15];
((elf_info *)ctx->info)->p->debugline = NULL;
if(elf_fpread(ctx, (void *)&str_tab, sizeof(str_tab), ctx->ehdr.shoff+
ctx->ehdr.shstrndx*sizeof(str_tab)) != sizeof(str_tab)) return EL_EIO;

for(int i = 0, off = ctx->ehdr.shoff; i < ctx->ehdr.shnum; i+=1,
off+=sizeof(sh_addr)){
    if(elf_fpread(ctx, (void *)&sh_addr, sizeof(sh_addr), off) !=
sizeof(sh_addr)) return EL_EIO;
    elf_fpread(ctx, (void *)name, 15, str_tab.offset + sh_addr.name);
    if(! strcmp(name, ".debug_line")){
        if(elf_fpread(ctx, (void *)max_va, sh_addr.size, sh_addr.offset) !=
sh_addr.size) return EL_EIO;
        make_addr_line(ctx, (char *)max_va, sh_addr.size);
        break;
    }
}
```

3. 中断处理

处理的中断是在 M 态下进行的,我们对 Illegal instructions 的中断进行一个更改,添加一个对于 `print_error()` 函数的调用，并在该函数中实现异常程序文件和代码的查找输出。

首先使用 `read_csr()` 函数找到断点,断点位置为 `mepc`。接着用 `while` 循环找到断点处指令对应的 `addr_line` 结构体，根据 `addr_line` 找到对应的 `code_file` 结构体和 `dir` 的位置，从而得到异常文件的路径 `path`。具体代码如下：

```
uint64 mepc = read_csr(mepc);
// get the addr_line structure of the instruction
int i = 0;
while(i < current->line_ind && mepc >= current->line[i].addr) i++;
addr_line *line = current->line+i-1;
// get the file name
```

```
int len = strlen((char *)current->dir[current->file[line->file].dir]);
strcpy(path, current->dir[current->file[line->file].dir]);
path[len] = '/';
strcpy(path + dir_len + 1, current->file[line->file].file);
```

得到文件路径后，我们就可以使用 `spike_file` 相关操作来读入源代码文件，并把对应行号的代码输出。具体实现如下：

```
// read the code line and print
spike_file_t *fp = spike_file_open(path, O_RDONLY, 0);
spike_file_stat(fp, &st);
spike_file_read(fp, co_line, st.st_size);
spike_file_close(fp);

int j = 0, off = 0;
while (off < st.st_size) {
    int k = off; while (k < st.st_size && co_line[k] != '\n') k++;
    if (j == line->line - 1) {
        co_line[k] = '\0';
        sprintf("Runtime error at %s:%d\n%s\n", path, line->line, co_line + off);
        break;
    }
    j++, off = k + 1;
}
```

最后，在 `handle_mtrap()` 中所有需要打印异常代码的 case 下调用 `print_error()` 函数，就可以输出相应的文件名和代码行，具体代码略。

2.3 实验调试及心得

1. 实验调试

根据以上分析修改 PKE 内核，得到正确输出结果，如图 2.1 所示。

```
xjh@ubuntu:~/Desktop/riscv-pke$ 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
    asm volatile("csrw sscratch, 0");
Illegal instruction!
System is shutting down with exit code -1.
```

图 2.1 正确输出结果

2. 心得体会

通过本次挑战实验，我进一步深入了解了操作系统的内部实现和调试信息格式 DWARF。在实验中，我学习了如何读取 ELF 文件的内容，找到包含调试信息的段，并将其保存下来。我还学会了调试信息的解析，构造指令地址-源代码行号-源代码文件名的对应表，并在异常中断处理函数中根据相应寄存器的值，在表中查找地址对应的代码行号和文件名。

此外，我还进一步熟悉了 ELF 文件的结构，掌握了如何在 ELF 文件中找到所需数据，熟悉了调试信息格式 DWARF，并了解了 PKE 系统的异常处理过程。在实验过程中，我也遇到了一些问题，例如如何正确解析调试信息、如何正确使用寄存器等，但通过查阅资料、与同学交流，我逐渐掌握了解决问题的方法。实验的最终结果也证明，我的解决方案是可行的。

总之，本次挑战实验让我更加深入地了解操作系统的内部实现和调试信息格式 DWARF，并提高了我的问题解决能力和编程技能。

在建议方面，由于这个挑战实验相对较难，希望实验指导书能够更加清晰地给出实验要求和步骤，尤其是在涉及到读取 ELF 文件和解析调试信息的部分，可以提供更加详细和具体的指导和代码示例，以帮助学生更好地完成实验。同时，也可以在指导书中列出需要学习的内容，例如 ELF 文件格式、DWARF 调试信息格式、PKE 系统的异常处理机制等，从而帮助学生更好地理解实验要求和目的。

3 lab2_challenge2

3.1 实验目的

1. 实验要求

本实验为挑战实验，给定应用 `app_singlepageheap.c`，通过修改 PKE 内核，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"。具体的实验要求为：

- 1) 增加内存控制块数据结构对分配的内存块进行管理。
- 2) 修改 `process` 的数据结构以扩展对虚拟地址空间的管理。
- 3) 设计函数对进程的虚拟地址空间进行管理，借助以上内容具体实现 `heap` 扩展。
- 4) 设计 `malloc` 函数和 `free` 函数对内存块进行管理。

2. 实验目的

- 1) 进一步理解操作系统的内存管理。
- 2) 进一步理解虚拟地址、物理地址，及其映射关系。

3.2 实验内容

1. 分析给定应用

首先观察给定应用 `app_singlepageheap.c` 文件，主要代码如下：

```
char *m = (char *)better_malloc(100);
char *p = (char *)better_malloc(50);
... ..
better_free((void *)m);
... ..
char *n = (char *)better_malloc(50);
if(m != n) {
    printf("your malloc is not complete.\n");
    exit(-1);
}
```

以上程序先利用 `better_malloc()` 分别申请 100 和 50 字节的一个物理页的内存，然后使用 `better_free()` 释放掉 100 字节，向 50 字节中复制一串字符串，进行输出。从预期结果可以看出，两次申请的空间在同一页面，并且释放第一块时，不会释放整个页面，所以需要设计合适的数据结构对各块进行管理。

2. 增加内存控制块

首先修改 `vmm.h` 文件，增加内存控制块数据结构对分配的内存块进行管理：

```
typedef struct mem_block{
    int size;
    // whether the memory block is occupied
    int flag;
    uint64 offset;
    struct mem_block *next;
}mem_block;
```

其中 `size` 表示内存块的数据大小，`flag` 表示内存块是否已被占用，`offset` 表示内存块的偏移，`next` 指针指向下一个内存块。我们通过 `mem_block` 管理已经申请的内存，即每调用一次 `malloc()` 申请 `n` 个字节时，实际物理空间中在 `n` 个字节之前会加上一个 `mem_block`。

3. 扩展 heap

在 `process.c` 中，添加 `user_vm_extend()` 函数，作用是对新增的虚拟地址添加对应的物理地址映射，便于实现对进程 `heap` 大小的管理，关键代码如下：

```
uint64 old_size = current->heap_size;
uint64 new_size = old_size + space;
old_size = ((old_size)+PGSIZE-1) & ~(PGSIZE-1);
for(uint64 i = old_size; i < new_size; i += PGSIZE){
    char *pa = (char *)alloc_page();
    if(pa != 0){
        memset(pa,0,sizeof(uint8) * PGSIZE);
        map_pages(current->pagetable, old_size, PGSIZE, (uint64)pa,
prot_to_type(PROT_READ | PROT_WRITE,1) );
    }
}
// update the heap size
current->heap_size = new_size;
```

`old_size` 是扩展前的虚拟地址空间，`new_size` 是扩展后的虚拟地址空间。值得注意的是，在映射前要先将 `old_size` 向上对齐一页，即 `old_size = ((old_size)+PGSIZE-1) & ~(PGSIZE-1)`。接着遍历新增加的虚拟地址空间，依次为其添加对应的物理地址的映射，具体操作为用 `alloc_page()` 函数申请一页物理页，用 `memset()` 初始化该页，再调用 `map_pages()` 函数将该页与虚拟地址 `old_size` 之后的一页地址映射，最后更新 `heap` 的大小。

4. 初始化内存池

在初始化进程时，需要对其堆内存池进行初始化。我们在 `vmm.c` 文件中用全局变量 `is_init` 来表示内存池是否被初始化，并增加 `init_memory()` 函数实现初始化。

若 `is_init` 的值为 1，则表示内存池已被初始化，可以直接进行后续的 `malloc` 等操作；若 `is_init` 的值为 0，则需要进行初始化。首先设置 `current->heap_size` 的大小，调用 `user_vm_extend()` 函数扩展 `heap` 的大小，新增地址空间的大小为 `sizeof(mem_block)`。接着通过 `page_walk()` 实现对虚拟地址空间的映射，最后更新内存控制块的数据。具体代码略。

5. 实现 `malloc()`

修改 `vmm.c` 文件，创建 `malloc()` 函数。根据题意，当用户程序申请 `n` 个字节的空间时，内核首先对该用户的内存池进行遍历，查找是否有空闲且大小大于 `n` 的可用内存块，如果有，则将该内存块的内存返回。如果没有，则向内核申请请在 `heap` 中增加 `sizeof(mem_block) + n` 个字节的大小，生成一个内存块并加入内存池中。函数的流程图如图 3.1 所示：

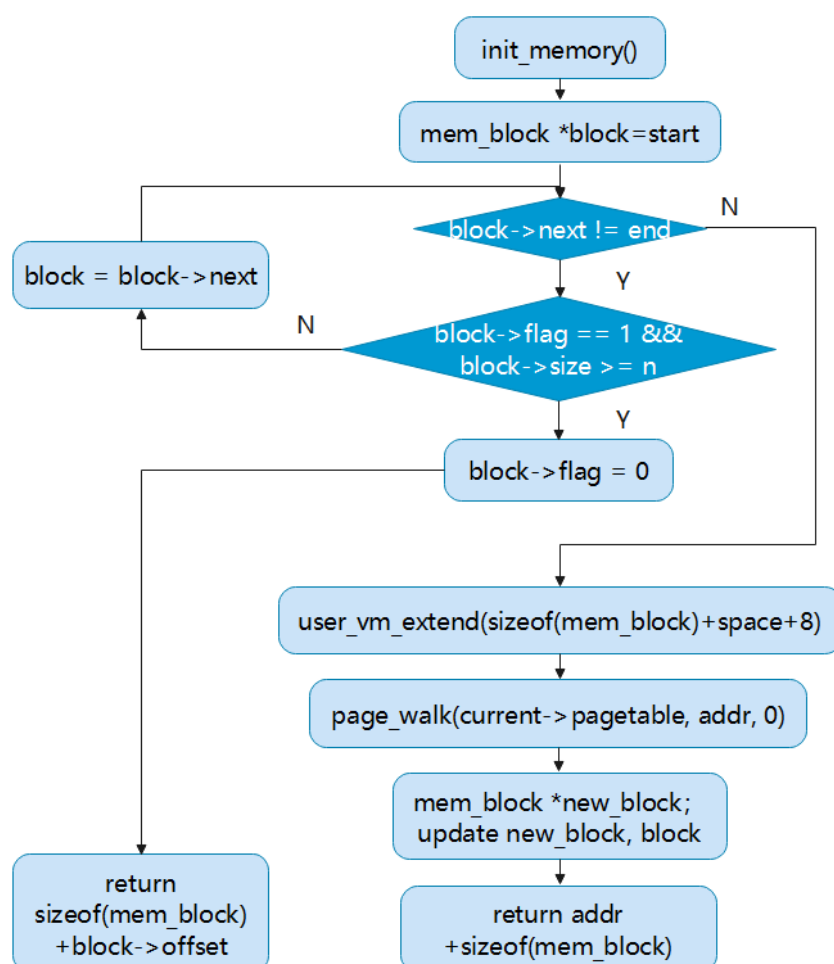


图 3.1 `malloc()` 函数流程图

6. 实现 `free()`

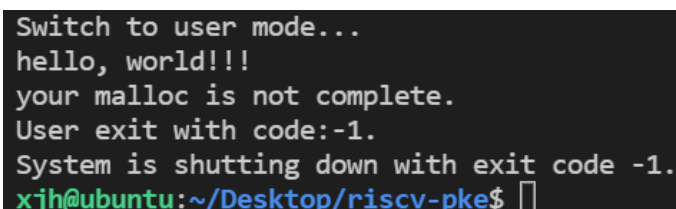
修改 vmm.c 文件，创建 free()函数，完成内存块的释放。先根据参数的虚拟地址和进程的 pagetable，找到对应的物理地址，接着找到其内存控制块 mem_block，将内存块的状态修改为空闲即可。代码如下：

```
void free(void *addr){
    // find the physical address according to the virtual address
    addr = (void *)((uint64)addr - sizeof(mem_block));
    pte_t *pte = page_walk(current->pagetable, (uint64)(addr), 0);
    // find the memory control block
    mem_block *block = (mem_block *)(((uint64)addr & 0xfff) + PTE2PA(*pte));
    block = (mem_block *)((8 - ((uint64)block % 8))%8 + (uint64)block);
    block->flag = 1;
}
```

3.3 实验调试及心得

1. 实验调试

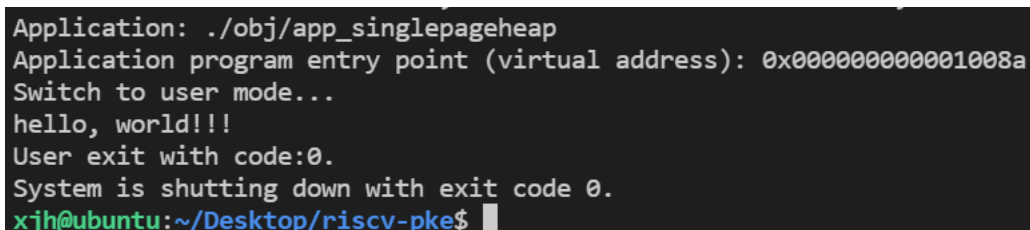
根据以上分析修改代码，设计分配和回收策略，执行结果如图 3.2 所示。程序能够正确输出“hello, word!!!”语句，但是出现“your malloc is not complete.”报错，且退出时 code 值为-1。



```
Switch to user mode...
hello, world!!!
your malloc is not complete.
User exit with code:-1.
System is shutting down with exit code -1.
xjh@ubuntu:~/Desktop/riscv-pke$
```

图 3.2 malloc() 错误

经过检查发现，是由于 malloc()函数出错，导致 app_singlepageheap.c 文件中两次申请块不在同一页面。进一步分析 malloc()函数，发现是 for 循环中结束条件有误，应为 block->next != end 而不是 block->= end。修改后得到正确结果，如图 3.3 所示：



```
Application: ./obj/app_singlepageheap
Application program entry point (virtual address): 0x000000000001008a
Switch to user mode...
hello, world!!!
User exit with code:0.
System is shutting down with exit code 0.
xjh@ubuntu:~/Desktop/riscv-pke$
```

图 3.3 正确输出结果

2. 心得体会

通过实验，我对内存管理和虚拟地址空间的扩展有了更深入的理解。在实验过程中，我需要根据实验要求对 PKE 内核进行修改，实现优化后的 malloc 函数，使应用程序能够在同一页面上申请内存块并正常输出字符串。

首先，我增加了内存控制块数据结构，对分配的内存块进行管理，并深刻认识到了内存控制块是管理内存块的关键数据结构，它包含了内存块的大小、状态和位置等重要信息，能够帮助我们高效地管理内存。

其次，我修改了进程的数据结构并扩展了对虚拟地址空间的管理，了解到进程的虚拟地址空间管理是操作系统内存管理的重要组成部分，可以通过合理的虚拟地址空间布局和管理策略来优化内存使用效率和系统性能。

接着，我设计了函数对进程的虚拟地址空间进行管理，借助以上内容实现了 heap 扩展。在实验中，我更深入地了解了堆内存管理的原理和实现方法，掌握了如何进行内存分配、回收和保护等操作。

最后，我还设计了 malloc() 函数和 free() 函数对内存块进行管理，掌握了如何设计一个高效的内存分配和回收算法，如何利用内存控制块和虚拟地址空间管理来优化内存分配的性能和效率。

总的来说，通过此次实验，我不仅学会了如何对 PKE 内核进行修改和优化，还深入了解了内存管理和虚拟地址空间的扩展原理和方法。这对我今后的学习和工作都有着重要的指导意义。

4 lab3_challenge2

4.1 实验目的

1. 实验要求

本实验为挑战实验，给定应用 `app_semaphore.c`，通过修改 PKE 内核和系统调用，为用户程序提供信号量功能。具体的实验要求为：

- 1) 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。
- 2) 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。

2. 实验目的

- 1) 进一步理解 PKE 的进程调度，熟悉系统调用路径。
- 2) 进一步理解信号量，通过实践了解其在内核中的实现。

4.2 实验内容

1. 分析给定应用

首先观察给定应用 `app_semaphore.c` 文件，主要代码如下：

```
int pid = fork();
if (pid == 0) {
    pid = fork();
    for (int i = 0; i < 10; i++) {
        sem_P(child_sem[pid == 0]);
        printu("Child%d print %d\n", pid == 0, i);
        if (pid != 0) sem_V(child_sem[1]); else sem_V(main_sem);
    }
} else {
    for (int i = 0; i < 10; i++) {
        sem_P(main_sem);
        printu("Parent print %d\n", i);
        sem_V(child_sem[0]);
    }
}
```

以上程序通过信号量的增减，控制主进程和两个子进程，按主进程、第一个子进程、第二个子进程、主进程、第一个子进程、第二个子进程……这样的顺序轮流输出。需要实现的函数为 `sem_new()`，`sem_P()`和 `sem_V()`。

2. 完善系统调用路径

首先在 `user_lib.h` 文件中添加 `sem_new()` 函数的原型, 参数 `value` 表示信号量的值。代码如下:

```
int sem_new(int value);
```

接着在 `user_lib.c` 文件中添加 `sem_new()` 函数的实现, 代码如下:

```
int sem_new(int value) {  
    return do_user_call(SYS_sem_new, value, 0, 0, 0, 0, 0, 0);  
}
```

在 `syscall.h` 文件中添加 `SYS_sem_new` 的注册, 代码如下:

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

相应地, 在 `syscall.c` 文件中添加 `SYS_sem_new` 系统调用的实现, 代码如下:

```
case SYS_sem_new:  
    return sys_sem_new(a1);
```

函数 `sys_sem_new()` 的实现为:

```
ssize_t sys_sem_new(int value){  
    return do_sem_new(value);  
}
```

以上均为添加信号量的系统调用部分, `sem_P()`、`sem_V()` 的系统调用也类似, 此处省略。完成这些全局变量的定义后, 在 `do_sem_new()`、`do_sem_P()`、`do_sem_V()` 中分别实现真正的信号量申请、P 操作、V 操作功能。

3. 完善数据结构

为了更好的管理信号量, 我们在 `process.h` 文件中添加了 `semaphore` 结构, 代码如下:

```
typedef struct semaphore_t{  
    int value;        // value of the semaphore  
    bool flag;        // whether the semaphore is used  
    process *head;    // the head of the process wait queue  
    process *tail;    // the tail of the process wait queue  
}semaphore;  
  
semaphore sem_array[NPROC];
```

`value` 是当前信号量的值; `flag` 表示当前信号量是否已被使用, 在后续申请信号量时需要选取 `flag` 为 `FALSE` 的信号量; `head` 表示当前正在等待该信号量的进程队列头指针; `tail` 表示该信号量的等待队列尾指针。我们用一个 `semaphore` 类型的数组 `sem_array` 来管理系统中的所有信号量。

4. 实现 `do_sem_new(int value)`

该函数用于申请信号量。当系统需要新的信号量时，我们首先遍历 `sem_array` 数组，若有 `flag` 为 `FALSE` 的信号量，则更新其 `value` 值、`flag` 值、等待队列，并返回在数组中的索引；若没有则返回-1。具体代码如下：

```
int do_sem_new(int value){
    for(int i = 0; i < NPROC; i++){ // traverse the semaphore array
        if(sem_array[i].flag == FALSE){ // semaphore is not used
            sem_array[i].value = value;
            sem_array[i].flag = TRUE;
            sem_array[i].head = 0;
            sem_array[i].tail = 0;
            return i;
        }
    }
    return -1;
}
```

4. 实现 `do_sem_P(int sem)`

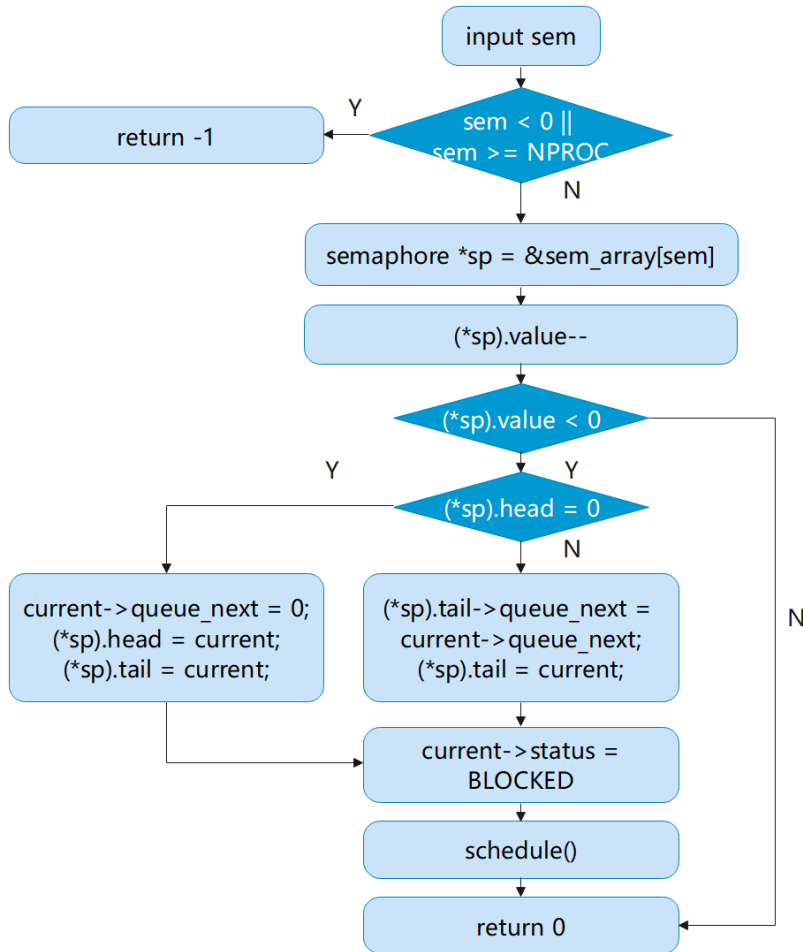


图 4.1 `do_sem_P()` 流程图

do_sem_P()函数用于实现信号量的 P 操作。当信号量减小时，首先更新信号量的值，如果信号量的值小于 0，说明当前进程需阻塞并等待信号量增加，所以将其状态修改为 **BLOCKED**，并加入到该信号量的等待进程队列中，然后触发 CPU 转进程调度。由于此时该进程已不再调度队列中，所以 CPU 会执行其他用户程序。函数的流程图如图 4.1 所示。

5. 实现 do_sem_V(int sem)

do_sem_V()函数用于实现信号量的 V 操作。当信号量增加时，首先更新信号量的值，然后检查等待该信号量的进程队列，如果队列不为空，则取出队首进程，并加入到调度队列中。函数的流程图如图 4.2 所示。

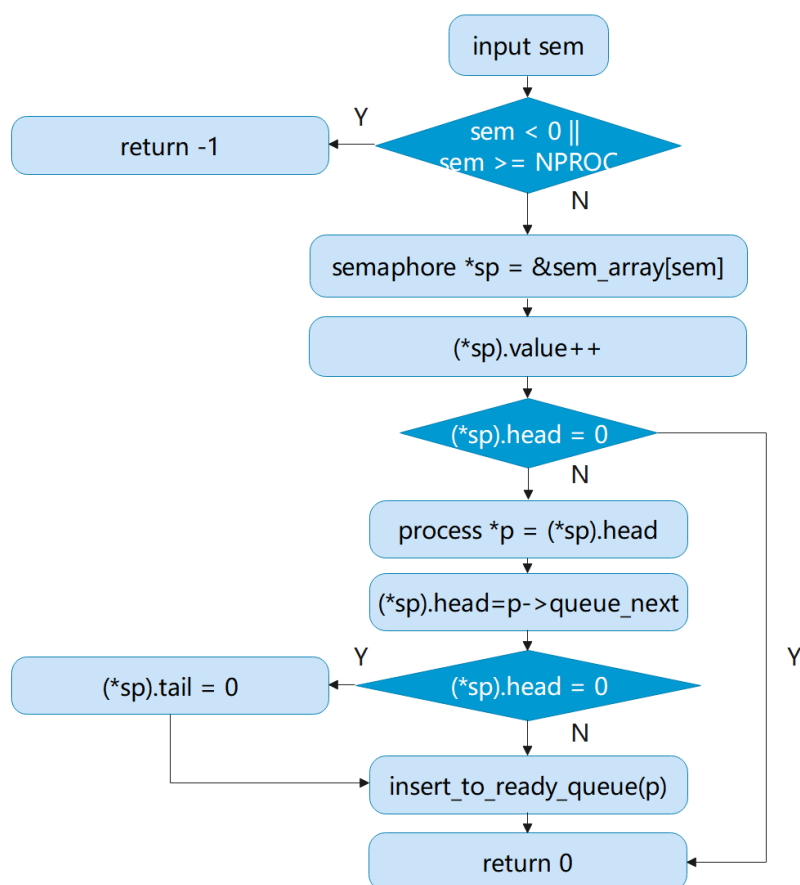


图 4.2 do_sem_V() 流程图

4.3 实验调试及心得

1. 实验调试

根据以上分析修改代码，实现信号量的分配释放的 P/V 操作，执行结果如图 4.3 所示。程序能够打印出主进程和子进程，但是出现顺序错乱并出现报错 “Not handled: we should let system wait for unfinished processes.”。

```
Application program entry point (virtual address): 0x00000000010078
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 print 0
going to schedule process 1 to run.
User call fork.
will fork a child from parent 1.
in alloc_proc. user frame 0x0000000087fa3000, user stack 0x000000007ffff000, user kstack 0x0000000087fa2000
going to insert process 2 to ready queue.
Child0 print 0
going to schedule process 2 to run.
Child1 print 0
ready queue empty, but process 0 is not in free/zombie state:3
ready queue empty, but process 1 is not in free/zombie state:3
ready queue empty, but process 2 is not in free/zombie state:3
Not handled: we should let system wait for unfinished processes.

System is shutting down with exit code -1.
```

图 4.3 malloc() 错误

全局搜索后发现，报错在 schedule()函数中出现，是由于进程池中还有未结束的进程导致的。进一步分析 schedule()的调用路径，发现 do_sem_P()中更新等待进程队列的指针出错，修改后得到正确结果。部分正确结果如图 4.4 所示。

```
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.
xjh@ubuntu:~/Desktop/riscv-pke$
```

图 4.4 正确输出结果

2. 心得体会

在本次实验中，我通过实践更深入地了解了 PKE 内核中信号量的实现方式和系统调用的处理，并深刻认识到了信号量在进程同步和互斥方面的作用和重要性，以及在内核中的实现方式和挑战。

首先，通过实验，我学会了添加系统调用，使得用户程序可以在内核态处理对信号量的操作。在实验中，我通过在内核中添加调用路径，并在用户程序中调用相关函数，使得用户可以在内核态进行信号量的操作。同时，我还深入了解了系统调用的实现方式，包括系统调用号的定义、系统调用的参数传递、系统调用的执行和返回等方面。通过这些实践，我对系统调用有了更深入的了解，也提高了自己的内核开发能力。

其次，我进一步掌握了在内核中实现信号量的分配、释放和 **PV** 操作的方式。在实验中，我通过在内核中添加相关数据结构和函数，并在用户程序中进行相关操作，了解了信号量在内核中的实现方式和调用路径。同时，我还深入了解了信号量在进程同步和互斥方面的作用和实现方式，包括 **PV** 操作的实现、等待队列的管理等方面。

实验过程中也遇到了一些难题。例如使用 `git` 提交代码失败、更新等待进程队列时出错等，通过查阅相关资料、请教老师同学，最终都顺利解决了。总的来说，本次实验趣味性很强，所给的指导资料也很丰富，老师们也在课程群里很积极的答疑解惑，在此非常感谢课程组老师们的辛苦付出！