



华中科技大学

操作系统原理课程设计报告

姓 名： 孙溪

学 院： 计算机科学与技术学院

专 业： 计算机科学与技术

班 级： CS2007

学 号： U202015512

指导教师： 谢美意

分数	
教师签名	

2023 年 3 月 30 日

目 录

实验一 打印用户程序调用栈.....	1
1.1 实验目的.....	1
1.2 实验内容.....	1
1.3 实验调试及心得.....	2
实验二 打印异常代码行.....	3
1.1 实验目的.....	3
1.2 实验内容.....	3
1.3 实验调试及心得.....	5
实验三 堆空间管理.....	7
1.1 实验目的.....	7
1.2 实验内容.....	7
1.3 实验调试及心得.....	9
实验四 实现信号量.....	11
1.1 实验目的.....	11
1.2 实验内容.....	11
1.3 实验调试及心得.....	13

实验一 打印用户程序调用栈

1.1 实验目的

实现系统调用以支撑 `print_backtrace` 函数，应用通过对该函数的调用，能够打印出之前的函数调用栈。

1.2 实验内容

该实验的主要思路为添加系统调用，并实现一个用于回溯函数调用栈的函数 `print_backtrace()`。

通过在 `user_lib` 和 `syscall` 里添加对应调用逻辑完成系统调用的注册：

```
int print_backtrace(int layers) {
    layers ++;
    return do_user_call(SYS_user_print_backtrace, layers, 0, 0, 0, 0, 0, 0);
}
```

图 1 在 `user_lib.c` 中

```
long do_syscall(long a0, long a1, long a2, long a3, long a4, long a5, long a6, long a7) {
    switch (a0) {
        case SYS_user_print:
            return sys_user_print((const char*)a1, a2);
        case SYS_user_exit:
            return sys_user_exit(a1);
        case SYS_user_print_backtrace:
            return sys_user_print_backtrace(a1);
        default:
            panic("Unknown syscall %ld \n", a0);
    }
}
```

图 2 在 `syscall.c` 中注册系统调用

在实现该函数时，首先需要获取用户进程的栈底指针。通过实验 1 的基础实验，我们知道在 PKE 环境中，应用在发出系统调用后，系统会切换栈到“用户内核”栈。获取用户态栈必须通过 `trapframe` 中存储的进程上下文中获取。通过 `current->trapframe->regs.sp` 获得用户态栈的栈底。

进一步需要通过用户态栈的结构，获取存储在栈中的应用函数调用的返回地址。对于 `f8` 函数，由于是最后一个函数调用，所以其栈帧里并未存放函数的返回地址。而其他函数的栈帧里，都存放了返回这些函数的地址 `ra`。因此，回溯函数调用栈的基本思路为通过对比返回地址 `ra` 的逻辑地址范围，来反向解析 `ra` 对应的函数名称。回溯过程的第一个 `ra` 地址，距离 `sp` 的长度为 `16+8` 个字节。

```

ssize_t sys_user_print_backtrace(int64 layers) {
    int64 cur_layer=0;
    for (uint64 p = current->trapframe->regs.sp + 24;
    cur_layer<layers; ++cur_layer, p += 16) {
        if (*(uint64*)p == 0) break;
        uint64 near_func = 0;
        int idx = -1;
        for (int i = 0; i < g_elfloader.syms_count; ++i) {
            if (g_elfloader.syms[i].info == STT_FUNC && g_elfloader.syms
[i].value < *(uint64*)p && g_elfloader.syms[i].value > near_func) {
                near_func = g_elfloader.syms[i].value;
                idx = i;
            }
        }
        if (idx == -1)
            continue;
        sprintf("%s\n", &g_elfloader.strtb[g_elfloader.syms[idx].name]);
    }
    return 0;
}

```

图 3 print_backtrace 函数实现

在回溯过程中，需要使用程序的符号表、字符串表以及逻辑地址到符号的解析，来获取函数的实际文本和虚拟地址。符号表和字符串表会在 ELF 文件中加入，在字符串表中对应位置上的字符串就是该符号名字的实际文本。符号的虚拟地址可以通过符号结构中的 value 成员获取，而符号名在虚地址空间的大小可以通过 size 成员获取。

1.3 实验调试及心得

该实验收获如下：

1. 理解 PKE 环境中应用和内核的栈的区别。
2. 了解如何在 PKE 环境中添加系统调用。
3. 掌握如何在 trapframe 中获取用户进程的上下文，包括栈底指针。
4. 理解用户态栈的结构，包括存储在栈中的应用函数调用的返回地址。
5. 掌握回溯函数调用栈的基本思路，包括对比返回地址逻辑地址范围来解析函数名称。
6. 掌握使用程序的符号表、字符串表和解析逻辑地址到符号的方法来获取函数的实际文本和虚拟地址。
7. 更深入地了解 ELF 文件格式的相关内容。

实验二 打印异常代码行

1.1 实验目的

修改 PKE 内核，包括 machine 文件夹下的代码，以便在用户程序发生异常时，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。需要使内核能够捕获用户程序的异常，并将异常相关的信息与用户程序源代码的位置进行关联，最终将该信息输出。

1.2 实验内容

对 elf_load 函数进行修改。需要添加 maxva 变量来存储程序各段虚拟地址的最大值，同时也需要找到 debug_line 段，并将其保存起来，最后调用构造表的函数。这个修改可以在函数内部添加一个循环来遍历所有段，计算出每个段的最大虚拟地址，然后取最大值作为 maxva。在找到 debug_line 段后，将其保存起来，可以将其指针作为参数传递给构造表函数进行处理。

```
char na[20]; ((elf_info *)ctx->info)->p->debugline = NULL;
elf_ssect_header name_seg, tmp_seg;
if (elf_fpread(ctx, (void *)&name_seg, sizeof(name_seg),
               ctx->ehdr.shoff + ctx->ehdr.shstrndx * sizeof(name_seg)) != sizeof(name_seg)) return EL_EIO;
for (i = 0, off = ctx->ehdr.shoff; i < ctx->ehdr.shnum; i++, off += sizeof(tmp_seg)) {
    if (elf_fpread(ctx, (void *)&tmp_seg, sizeof(tmp_seg), off) != sizeof(tmp_seg)) return EL_EIO;
    elf_fpread(ctx, (void *)na, 20, name_seg.offset + tmp_seg.name);
    if (strcmp(na, ".debug_line") == 0) {
        if (elf_fpread(ctx, (void *)maxva, tmp_seg.size, tmp_seg.offset) != tmp_seg.size) return EL_EIO;
        make_addr_line(ctx, (char *)maxva, tmp_seg.size); break;
    }
}
```

图 4 找到 debug_line 段

在 print_error()函数中，需要根据文件索引找到文件名和文件夹索引，以及根据文件夹索引找到文件夹名。然后将文件夹名和文件名拼接成触发异常的源代码文件路径。可以利用 htif 功能读入源代码文件，然后找到对应行号的代码输出即可。

```

void print_error() {
    uint64 mepc = read_csr(mepc);
    for (int i = 0; i < current->line_ind; i++) {
        if (mepc < current->line[i].addr) {
            addr_line *line_ar = current->line + i - 1;
            strcpy(path, current->dir[current->file[line_ar->file].dir]);
            path[strlen(current->dir[current->file[line_ar->file].dir])] = '/';
            strcpy(path + strlen(current->dir[current->file[line_ar->file].dir) + 1,
                current->file[line_ar->file].file);
            spike_file_t *f = spike_file_open(path, O_RDONLY, 0);
            spike_file_stat(f, &m_stat); spike_file_read(f, code, m_stat.st_size);
            spike_file_close(f); int off = 0, cnt = 0;
            while (off < m_stat.st_size) {
                int x = off;
                while (x < m_stat.st_size && code[x] != '\n')
                    x++;
                if (cnt == line_ar->line - 1) {
                    code[x] = '\0';
                    sprintf("Runtime error at %s:%d\n%s\n",
                        path, line_ar->line, code + off);
                    break;
                } else cnt++, off = x + 1;
            }
            break;
        }
    }
}
}

```

图 5 print_error()函数实现

在 handle_mtrap 函数中，需要对所有需要打印异常代码行的 case 下调用 print_error()函数。也就是说，在处理每种异常的过程中，如果需要打印出对应的代码行，就将相应的参数传递给 print_error()函数来实现。

```

void handle_mtrap() {
    uint64 mcause = read_csr(mcause);
    switch (mcause) {
        case CAUSE_MTIMER:
            handle_timer();
            break;
        case CAUSE_FETCH_ACCESS:
            print_error();
            handle_instruction_access_fault();
            break;
        case CAUSE_LOAD_ACCESS:
            print_error();
            handle_load_access_fault();
        case CAUSE_STORE_ACCESS:
            print_error();
            handle_store_access_fault();
            break;
        case CAUSE_ILLEGAL_INSTRUCTION:
            print_error();
            handle_illegal_instruction();
            break;
        case CAUSE_MISALIGNED_LOAD:
            print_error();
            handle_misaligned_load();
            break;
        case CAUSE_MISALIGNED_STORE:

```

图 6 handle_mtrap 中部分异常打印处理

总体来说，这个实验的目的是在 PKE 内核中添加一个新的异常处理功能，使得在异常处理时能够打印出对应的异常代码行。具体实现需要对几个函数进行修改，并且需要涉及到读取源代码文件等操作。

1.3 实验调试及心得

这个实验的思路涉及到对 PKE 内核中的一些函数进行修改，以及在异常处理时打印出对应的异常代码行。我从这个实验中获得了以下几个方面的收获：

1. 理解了操作系统异常处理的原理。在这个实验中，我需要在异常处理时打印出对应的异常代码行。这要求我理解操作系统异常处理的原理，了解如何在异常处理时获取异常的上下文信息。这让我对操作系统的异常处理有了更深入的了解。
2. 掌握了操作系统的内核开发技能。在这个实验中，我需要对 PKE 内核中的一些函数进行修改，实现新的功能。这让我学会了如何在操作系统内核中添加新的功能模块，并且熟练掌握了内核开发技能。
3. 学会了操作系统编程的一些技巧。在这个实验中，我需要读取源代码文件，

并且输出对应行号的代码。这让我学会了如何在操作系统中使用文件系统，以及如何进行文件读写操作。同时，我也掌握了一些调试技巧，如如何输出调试信息等。

4. 在这个实验中，我需要对 PKE 内核进行修改，同时需要解决一些代码实现的问题。这要求我具备一定的问题解决能力，能够独立思考、分析问题，并提出解决方案。这是一个非常重要的能力，在操作系统开发和其他领域的工作中都非常有用。

实验三 堆空间管理

1.1 实验目的

修改 PKE 内核，包括 machine 文件夹下的代码，以实现优化后的 malloc 函数。需要使得应用程序在两次申请块时能够在同一页面中申请内存，并且能够正常输出存入第二块中的字符串"hello world"。

1.2 实验内容

在 vmm.h 中增加内存控制块，使用 control_block 来管理已经申请的内存。当调用 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 control_block。

```
typedef struct control_block{
    int available;
    int size;
    uint64 off;
    struct control_block *next;
} control_block;
```

图 7 control_block 定义

在 kernel/vmm.c 中声明 user_vm_malloc 函数，目的是将进程 new_size 与 old_size 之间的虚拟地址映射到实际的物理地址上。将 old_size 向上对其一页，含义是对其后的地址之前的虚拟地址已经与实际的物理地址有了映射，所以此时的 new_size 是可能小于 old_size 的。当 new_size < old_size 时，无需对虚拟地址与物理地址之间的映射进行操作，直接返回即可。当 new_size > old_size 时，通过该函数实现对 new_size 与 old_size 之间的虚拟地址添加对应的物理地址的映射，具体操作为申请一页物理页，初始化该页，将该页与虚拟地址 old_size 之后的一页地址映射。该函数的实现可以方便后面直接通过系统调用来实现对进程 heap 大小的管理。

```

uint64 user_vm_malloc(pagetable_t pagetable, uint64 old_size, uint64 new_size) {
    if(old_size > new_size)
        return old_size;
    old_size = PGROUNDUP(old_size);
    char *mem;
    for(uint64 old = old_size; old < new_size; old += PGSIZE) {
        mem = (char *)alloc_page();
        if(mem == 0)
            panic("failed to user_vm_malloc .\n");
        memset(mem, 0, sizeof(uint8) * PGSIZE);
        map_pages(pagetable, old_size, PGSIZE, (uint64)mem, prot_to_type(PROT_READ | PROT_WRITE, 1));
    }
    return new_size;
}

```

图 7 user_vm_malloc 函数实现

利用 user_vm_malloc 构造 extendProcess 函数。对进程的 heap_sz 进行实际的管理。抽象 extendProcess 函数成系统调用 sys_user_sbrk。

管理内存池。修改 kernel/vmm.c 代码，创建函数 malloc(int n)，实现优先从内存池中获取申请的空间。

```

uint64 malloc(int n)
{
    if(!init_flag)
    {
        current->heap_sz = USER_FREE_ADDRESS_START;
        uint64 addr = current->heap_sz;
        extendProcess(sizeof(control_block));
        pte_t *pte = page_walk(current->pagetable, addr, 0);
        control_block *fir_control_block = (control_block *) PTE2PA(*pte);
        current->heap_memory_start = (uint64) fir_control_block;
        fir_control_block->next = fir_control_block;
        fir_control_block->size = 0;
        current->heap_memory_last = (uint64) fir_control_block;
        init_flag = 1;
    }
    control_block *head = (control_block *)current->heap_memory_start;
    control_block *last = (control_block *)current->heap_memory_last;
    while (1) {
        if(head->size >= n && head->available == 1) {
            head->available = 0;
            return head->off + sizeof(control_block);
        }
        if(head->next == last) break;
        head = head->next;
    }
    uint64 allo_addr = current->heap_sz;
    extendProcess((uint64) (sizeof(control_block) + n + 8));
    pte_t *pte = page_walk(current->pagetable, allo_addr, 0);
    control_block *cur = (control_block *) (PTE2PA(*pte) + (allo_addr & 0xfff));
    uint64 amo = (8 - ((uint64)cur % 8))%8;
    cur = (control_block *) ((uint64)cur + amo);
    cur->available = 0;
    cur->off = allo_addr;
    cur->size = n;
    cur->next = head->next;
    head->next = cur;
    head = (control_block *)current->heap_memory_start;
    return allo_addr + sizeof(control_block);
}

```

图 8 malloc 函数实现

1.3 实验调试及心得

本实验主要实现了增强版的内存管理方式,通过引入内存控制块来管理已经申请的内存,从而更好地控制内存的使用。

在进程调用 malloc()申请内存时,会分配一个 control_block 与所申请的内存相关联。同时,在 kernel/vmm.c 中实现 user_vm_malloc 函数将进程的虚拟地址与实际物理地址进行映射,并实现 extendProcess 函数通过系统调用管理进程的

heap 大小。这样，当进程需要使用内存时，可以更好地控制内存的分配和使用。

为了优化内存使用，实验中引入了内存池的概念，通过管理内存池的方式，实现内存的复用，减少申请内存的次数。当用户请求 n 个字节的内存时，首先遍历该用户的内存池，查找是否有可用的内存块，如果有，直接返回内存块，并将其状态修改为已分配。如果没有可用的内存块，则向内核申请物理内存，生成一个内存块并将其加入该用户的内存池中。

为方便内存管理，实验中还实现了 `free` 函数。它会根据提供的虚拟地址查找对应的物理地址和控制块，并将其放回进程的内存池中，以便后续的内存使用。

通过这些改进，可以更好地管理内存，提高内存利用率并减少内存碎片，从而提高系统的性能和稳定性。

我学会了如何在操作系统中管理内存，并实现了内存管理的各种功能。具体来说，我了解了如何创建内存控制块，如何将虚拟地址与物理地址进行映射，以及如何通过内存池的方式来优化内存使用。同时，我也掌握了如何实现系统调用以及如何在用户空间中使用这些系统调用，以方便应用程序的开发。

实验四 实现信号量

1.1 实验目的

修改 PKE 内核和系统调用，以提供信号量功能，从而实现对进程的控制。实现一个信号量结构体，并修改内核和系统调用，以使用户程序可以使用信号量。测试实现，确保它能够正确地控制进程的输出顺序。

1.2 实验内容

要在 PKE 内核中实现信号量机制，以实现进程间的同步和互斥。具体的实现步骤如下：

在 process.h 头文件中添加信号量结构体，该结构体包含了信号量的值，以及表示该信号量是否已被占用的 occupied 标志位，以及等待该信号量的进程队列的队列头指针和队列尾指针。添加信号量的申请、释放和增减操作函数。在分配信号量时，需要在信号量数组中选取 occupied 属性为 0 的信号量。

```
typedef struct {
    int value;
    int occupied;
    process *head;
    process *tail;
} semaphore;

//信号量的分配和回收功能
int increase_se(int n);
int decrease_se(int n);
int allocate_se(int value);
```

图 9 process.h 中的信号量结构体和功能函数定义

在 process.c 文件中实现函数。其中，P 操作表示对信号量进行减小操作，V 操作表示对信号量进行增加操作。当进行 V 操作时，需要先更新信号量的值，然后检查等待该信号量的进程队列，如果队列不为空，则取出队头进程，加入到

调度队列中。当进行 P 操作时，首先更新信号量的值，如果信号量的值小于 0，则说明当前进程需要阻塞并等待信号量增加，所以将其状态修改为 **BLOCKED**，加入到该信号量的等待进程队列中，然后触发 CPU 的调度，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。

```
semaphore semas[NPROC];

int increase_se(int n) {
    if(n < 0 || n >= NPROC)
        return -1;
    semas[n].value++;
    if(semas[n].head) {
        process *p = semas[n].head;
        semas[n].head = p->queue_next;
        if(!p->queue_next) semas[n].tail = 0;
        insert_to_ready_queue(p);
    }
    return 0;
}

int decrease_se(int n) {
    if(n < 0 || n >= NPROC)
        return -1;
    semas[n].value--;
    if(semas[n].value < 0) {
        if(semas[n].head) {
            semas[n].tail->queue_next = current->queue_next; semas[n].tail = current;
        }
        else {
            semas[n].head = semas[n].tail = current; current->queue_next = 0;
        }
        current->status = BLOCKED;
        schedule();
    }
    return 0;
}

int allocate_se(int value) {
    for(int i=0; i<NPROC; i++) {
        if(semas[i].occupied==0) {
            semas[i].occupied = 1;
            semas[i].value = value;
            semas[i].head = semas[i].tail = 0;
            return i;
        }
    }
    return -1;
}
```

图 10 process.c 文件中实现函数

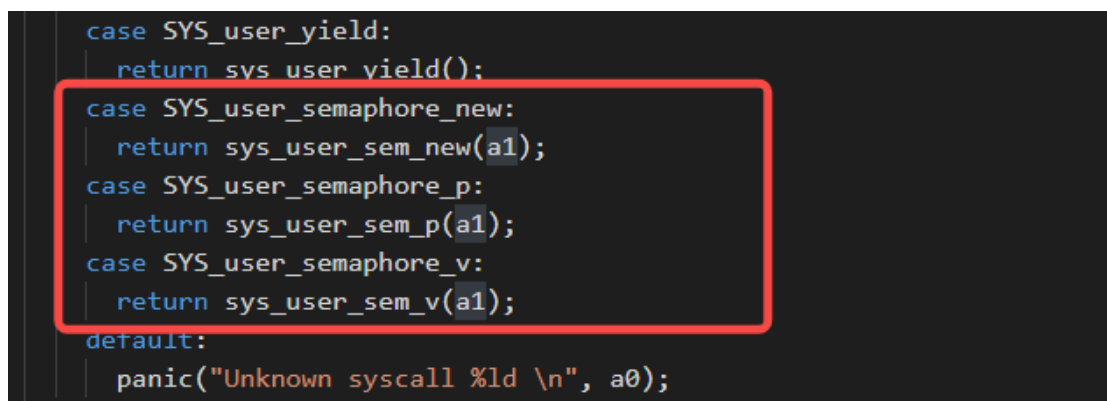
具体来说，需要在系统调用中添加三个函数，分别是 `sem_new`、`sem_P`、和

sem_V 函数。

sem_new 函数会调用系统内部的分配函数，根据输入的信号量值和信号量数组的数量，为该信号量分配唯一的标识符，并将该信号量标识符、信号量数组和信号量的值返回给用户程序。

sem_P 函数会接收一个信号量标识符作为参数，根据该标识符找到对应的信号量，并进行 P 操作。如果信号量的值小于 0，则将当前进程加入到等待该信号量的进程队列，并调用 CPU 的调度函数，将 CPU 控制权交给其他可以运行的程序。

sem_V 函数会接收一个信号量标识符作为参数，根据该标识符找到对应的信号量，并进行 V 操作。如果信号量对应的进程队列不为空，则将队头进程加入到调度队列中，等待 CPU 的执行。否则，直接增加信号量的值。



```
case SYS_user_yield:
    return sys_user_yield();
case SYS_user_semaphore_new:
    return sys_user_sem_new(a1);
case SYS_user_semaphore_p:
    return sys_user_sem_p(a1);
case SYS_user_semaphore_v:
    return sys_user_sem_v(a1);
default:
    panic("Unknown syscall %ld \n", a0);
```

图 11 系统调用注册

通过内部函数和系统调用的配合，我们就可以实现对信号量的申请、释放、和修改操作，从而进行进程间通信和管理，实现多任务操作系统的基本功能。

总的来说，这个实验实现了信号量机制，可以让用户程序通过系统调用函数来进行信号量的申请、释放和增减操作，从而实现进程间的同步和互斥。

1.3 实验调试及心得

通过这个实验，我深入了解了信号量机制的原理和实现方法。信号量是一种用于进程间同步和互斥的机制，是操作系统设计和实现中非常重要的概念。熟悉操作系统内核的实现：通过这个实验，我学习了如何在操作系统内核中实现信号量机制，熟悉了操作系统内核的实现方式和开发技术。

因为需要编写一些底层的代码来实现信号量机制，对我的编程能力有一定的提高作用。加深对操作系统的理解，更深入地了解了操作系统的原理和机制，对我理解操作系统的整体结构和功能有很大帮助。