



华中科技大学

操作系统原理课程设计报告

姓 名： 张钧玮
学 院： 计算机学院
专 业： 数据科学与大数据专业
班 级： 大数据 2102 班
学 号： U202115520
指导教师： 廖小飞

分数	
教师签名	

2024 年 4 月 1 日

目 录

1	打印异常代码行	1
1.1	实验目的	1
1.2	实验内容	2
1.3	实验调试及心得	5
2	堆空间管理	6
2.1	实验目的	6
2.2	实验内容	6
2.3	实验调试及心得	11

1 打印异常代码行

1.1 实验目的

修改 pke 内核，使得程序在用户态试图读取内核态寄存器时，抛出错误并且输出触发异常的用户程序源文件名和对应代码行。正确的程序输出如下。

```
1 riscv-pke on ʘ lab1_challenge2_errorline
2 > spike ./obj/riscv-pke obj/app_errorline
3 In m_start, hartid:0
4 HTIF is available!
5 (Emulated) memory size: 2048 MB
6 Enter supervisor mode...
7 Application: obj/app_errorline
8 Application program entry point (virtual address): 0x0000000081000000
9 Switch to user mode...
10 Going to hack the system by running privilege instructions.
11 Runtime error at user/app_errorline.c:13
12     asm volatile("csrw sscratch, 0");
13 Illegal instruction!
14 System is shutting down with exit code -1.
```

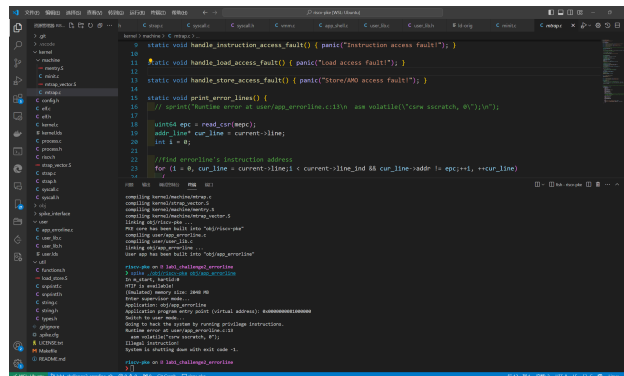


图 1: 正确的输出 lab1_challenge2

1.2 实验内容

为了完成实验，需要执行并且完成以下的两部分内容。

①首先，需要增加对 elf 可执行文件的调试探查。具体来说，就是在 elf.h 头文件中定义，在 elf.c 文件中实现函数 `elf_status load_debugline(elf_ctx *ctx);`，并且在 elf 的二进制程序加载执行函数 `void load_bincode_from_host_elf(process *p)` 中运行以加载 debug 段。

以下是 `load_debugline` 的具体代码以及思路。

```

1 // added @lab1_challenge2 | load debugline
2 elf_status load_debugline( elf_ctx *ctx ) {
3     // sprint( "hello load_debugline\n");
4     // sprint("%d\n",ctx);
5     // claim string table header
6     elf_sect_header strtab;
7     // get string table header's info
8     int offset;
9
10    offset = ctx->ehdr.shoff;
11    // sprint("%d\n",ctx->ehdr.version);
12    // sprint( "%d\n", offset );
13    // sprint( "%d\n", ctx->info );
14    offset += sizeof( strtab ) * ( ctx->ehdr.shstrndx );
15    // sprint( "%d\n", offset );
16
17    // save string table header
18    if ( elf_fpread( ctx, (void *) &strtab, sizeof( strtab ), offset ) !=
19        sizeof( strtab ) ) {
20        panic( "string table header get failed!\n" );
21    }
22
23    // save string table
24    static char strtab_info[ STRTAB_MAX ];
25    if ( elf_fpread( ctx, (void *) strtab_info, strtab.size,
26        strtab.offset ) !=
27        strtab.size ) {
28        panic( "string table get failed!\n" );
29    }
30
31    // get .debug_line segment
32    elf_sect_header debugseg;
33    int index;
34    for ( index = 0, offset = ctx->ehdr.shoff; index < ctx->ehdr.shnum;
35        index++, offset += sizeof( debugseg ) ) {
36        if ( elf_fpread( ctx, (void *) &debugseg,
37            sizeof( elf_sect_header ),
38            offset ) != sizeof( elf_sect_header ) )

```

```
37         panic( "debug header get failed!\n" );
38         // sprintf("i=%d,%s\n",i, strtab_info+debugseg.name);
39         if ( strcmp( (char *) ( strtab_info + debugseg.name ),
40                     ".debug_line" ) == 0 )
41             break;
42     }
43
44     if ( index == ctx->ehdr.shnum ) {
45         panic( "can't find debugline!\n" );
46         return EL_ERR;
47     }
48     int i;
49
50     // get debugline's information
51     static char debugline[ DEBUGLINE_MAX ];
52     if ( elf_fpread( ctx, (void *) debugline, debugseg.size,
53                     debugseg.offset ) != debugseg.size )
54         panic( "debugline get failed!\n" );
55     make_addr_line( ctx, debugline, debugseg.size );
56     return EL_OK;
57 }
```

函数主要思路如下：

- 1.首先，从ELF上下文中获取字符串表头信息。这个信息包含了字符串表的偏移和大小等。
- 2.然后，通过上一步得到的字符串表头信息，从ELF文件中读取字符串表的内容，并保存到数组 *strtab_info* 中。
- 3.接下来，遍历ELF文件中的节头表，寻找名称为.debug_line的段。
- 4.一旦找到了.debug_line段，就从文件中读取该段的内容，保存到另一个数组中。这些内容包含调试信息，比如源代码文件名、行号等。
- 5.最后，调用 *make_addr_line* 函数来处理这些调试线信息将缓冲区指针传入。
- 6.函数返回一个表示加载调试线信息是否成功的状态，EL_OK表示成功，或者EL_ERR表示失败。

②完成对 elf 二进制程序的 debug_line 段的读取以后，接下来要做的就是抛出异常时打印出源程序名和对应代码行。

异常属于中断的一种，因此通过 *mstrap.c* 中定义实现并且在触发 *handle_illegal_instruction* 函数时候执行 *print_error_lines* 函数，就可以正确的完成需求。以下是 *print_error_lines* 函数的具体代码以及实现。

```

1 static void print_error_lines() {
2     // sprint("Runtime error at user/app_errorline.c:13\n    asm
3     volatile("\csrwr sscratch, 0\");\n");
4     uint64 epc = read_csr(mepc);
5     addr_line* cur_line = current->line;
6     int i = 0;
7
8     //find errorline's instruction address
9     for (i = 0, cur_line = current->line; i < current->line_ind &&
10    cur_line->addr != epc; ++i, ++cur_line)
11     {
12         //sprint("i=%d,addr=%x\n",i,cur_line->addr);
13         if (cur_line->addr == epc) break;
14     }
15     if (i == current->line_ind) panic("can't find errorline!\n");
16     //sprint("find errorline,i=%d\n",i);
17
18     //find file's path and name
19     char filename[FILENAME_MAX];
20     //find filename
21     code_file* cur_file = current->file + cur_line->file;
22     char* single_name = cur_file->file;
23     //find file's path
24     char* file_path = (current->dir)[cur_file->dir];
25     //combine path and name
26     int start = strlen(file_path);
27     strcpy(filename, file_path);
28     filename[start] = '/';
29     start++;
30     strcpy(filename + start, single_name);
31     sprint("Runtime error at %s:%d\n", filename, cur_line->line);
32
33     //find error line
34     //error instruction's line
35     int error_line = cur_line->line;
36     //open file
37     spike_file_t* file = spike_file_open(filename, 0_RDONLY, 0);
38     if (IS_ERR_VALUE(file)) panic("open file failed!\n");
39     //get file's content
40     char file_detail[FILE_MAX];
41     spike_file_pread(file, (void*)file_detail, sizeof(file_detail), 0);

```

```

42
43 //fine error line's start
44 int line_start = 0;
45 for (i = 1; i < error_line; i++)
46 {
47     //sprintf("line=%d,%s\n", i, file_detail + line_start);
48     while (file_detail[line_start] != '\n') line_start++;
49     line_start++;
50 }
51 char* errorline = file_detail + line_start;
52 while (*errorline != '\n') errorline++;
53 *errorline = '\0';
54 //print error line
55 sprintf("%s\n", file_detail + line_start);
56 spike_file_close(file);
57 }

```

函数的主要思路如下：

- 1.寄存器 `epc` 是用于保存当前执行代码的地址的处理器寄存器，因此为了获得抛出错误的代码位置需要首先获得代码的地址。`read_csr(mepc)`获取发生错误的代码地址。
- 2.遍历当前进程的代码行数组，在其中查找发生异常的指令地址对应的代码行。
- 3.根据进程提供信息，拼接获取代码文件的路径和文件名。
- 4.打开代码文件，读取文件内容。定位到错误行在文件内容中的起始位置。
- 5.提取错误行的内容并打印出来，同时关闭文件。

1.3 实验调试及心得

关于 `elf` 文件与进程函数之间的接口问题：读取 `elf` 文件的代码，找到包含调试信息的段以后，想要将其内容保存起来，有两个可以的手段。要么是用静态数组来存储 `debug_line` 段数据，那么这个数组必须足够大；也可以把 `debug_line` 直接放在程序所有需映射的段数据之后，这样可以保证有足够大的动态空间。本次挑战因为提供了方便的 `util` 函数 `make_addr_line` 来解析 `debug_line` 段并保存到静态数组，因此选择后者。

2 堆空间管理

2.1 实验目的

修改内核，使用 *better_malloc* 替代原来 pke 非常简陋的 *malloc*，使用 *better_free* 替代原来的 *free*。目标正确的程序输出如下。

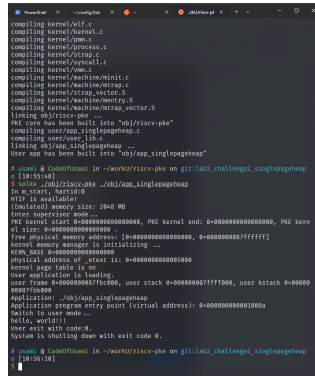


图 2: 正确的输出 lab2_challenge2

```
#      usami      @      CodeOfUsami      in      ~/workU/riscv-pke      on
1
git:lab2_challenge2_singlepageheap o [11:22:30]
2 $ spike ./obj/riscv-pke ./obj/app_singlepageheap
3 In m_start, hartid:0
4 HTIF is available!
5 (Emulated) memory size: 2048 MB
6 Enter supervisor mode...
PKE   kernel   start   0x0000000080000000,   PKE   kernel   end:
7 0x0000000080008000, PKE kernel size: 0x0000000000008000 .
free   physical   memory   address:   [0x0000000080008000,
8 0x0000000087ffffff]
9 kernel memory manager is initializing ...
10 KERN_BASE 0x0000000080000000
11 physical address of _etext is: 0x0000000080005000
12 kernel page table is on
13 User application is loading.
user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user
14 kstack 0x0000000087fbb000
15 Application: ./obj/app_singlepageheap
16 Application program entry point (virtual address): 0x00000000001008a
17 Switch to user mode...
18 hello, world!!!
19 User exit with code:0.
20 System is shutting down with exit code 0.
```

2.2 实验内容

为了完成实验，需要首先学习理解原先简单的 *malloc* 和 *free* 函数的实现，然后再实现 *better_malloc* 和 *better_free* 函数。

以下是 *better_malloc* 和 *better_free* 的具体代码以及思路。

```

1 //
2 // maybe, the simplest implementation of malloc in the world ... added
3 @lab2_2
4 //
5 uint64 sys_user_allocate_page() {
6     void *pa = alloc_page();
7     uint64 va = g_ufree_page;
8     g_ufree_page += PGSIZE;
9     user_vm_map( (pagetable_t) current->pagetable, va, PGSIZE, (uint64)
10     pa,
11                 prot_to_type( PROT_WRITE | PROT_READ, 1 ) );
12     return va;
13 }

```

原始的 *malloc* 函数实现

```

1 // added @lab2_challenge2
2 // allocate a block to user,size is n.  return virtual address
3 uint64 sys_user_better_allocate_page( long n ) {
4     uint64 va = user_better_allocate( n );
5     return va;
6 }
7 //
8 //
9 //
10
11 // added @lab2_challenge2
12 uint64 user_better_allocate( long n ) {
13
14     // 以 8 为倍数向上取整
15     n = ROUNDUP( n, 8 );
16     if ( n > PGSIZE ) {
17         panic( "fail to allocate PGSIZE due to too big size.\n" );
18     }
19
20     // 遍历块链表取到一个合适的块
21     BLOCK *cur = current->free_start, *pre = current->free_start;
22     while ( cur ) {
23         if ( cur->size >= n ) break;
24         pre = cur;
25         cur = cur->next;
26     }
27
28     // 页表耗尽则触发分页
29     if ( cur == NULL ) {

```

```

30
31     void *pa = alloc_page();
32     uint64 va = g_ufree_page;
33     g_ufree_page += PGSIZE;
34     user_vm_map( (pagetable_t) current->pagetable, va, PGSIZE,
35 (uint64) pa,
36                 prot_to_type( PROT_WRITE | PROT_READ, 1 ) );
37     // turn this new page to a block
38     BLOCK *temp = (BLOCK *) pa;
39     temp->start = (uint64) pa + sizeof( BLOCK );
40     temp->size = PGSIZE - sizeof( BLOCK );
41     temp->va = va + sizeof( BLOCK );
42     temp->next = NULL;
43     // 插入新块
44     if ( pre == NULL ) {
45         current->free_start = temp;
46         cur = temp;
47     } else {
48         if ( pre->va + pre->size >= ( ( pre->va ) / PGSIZE +
49 PGSIZE ) ) {
50             pre->size += PGSIZE;
51             cur = pre;
52         } else {
53             pre->next = temp;
54             cur = temp;
55         }
56     }
57     //分配块
58     if ( n + sizeof( BLOCK ) < cur->size ) {
59         BLOCK *to_use = (BLOCK *) ( cur->start + n );
60         to_use->start = cur->start + n + sizeof( BLOCK );
61         to_use->size = cur->size - n - sizeof( BLOCK );
62         to_use->va = cur->va + n + sizeof( BLOCK );
63         to_use->next = cur->next;
64         if ( cur == current->free_start )
65             current->free_start = to_use;
66         else
67             pre->next = to_use;
68     } else {
69         if ( cur == current->free_start )
70             current->free_start = cur->next;
71         else
72             pre->next = cur->next;
73     }
74     // 块置入已用链表
75     cur->size = n;
76     cur->next = current->used_start;
77     current->used_start = cur;
78     return cur->va;
79

```

80 }

better_malloc 函数实现

原始的*free*函数实现每次分配内存都会分配一个新的页，这样会导致内存资源的浪费。

而*better_malloc*函数接受一个参数*n*作为要分配的内存大小，接下来进行以下步骤高效地分配内存。

1. 首先对要分配的内存大小向上取整为8的倍数。
2. 遍历块链表，找到第一个大小符合要求的空闲块。
3. 如何找不到合适大小的空闲块，再分配新的物理页，并将其映射到用户虚拟地址空间，然后将其转换成一个空闲块，并插入到空闲块链表中。
4. 如果找到合适大小的空闲块，将其分配给用户，并将其从空闲块链表中移除，插入到已用块链表中。

第二步是完成*better_free*函数的实现，以下是*better_free*的具体代码以及思路。

```

1 //
2 // reclaim a page, indicated by "va". added @lab2_2
3 //
4 uint64 sys_user_free_page( uint64 va ) {
5     user_vm_unmap( (pagetable_t) current->pagetable, va, PGSIZE, 1 );
6     return 0;
7 }
```

简单的*free*函数实现

```

1 // added @lab2_challenge2
2 void user_better_free( uint64 va ) {
3     // 寻找需要释放的块
4     BLOCK *cur = current->used_start, *pre = current->free_start;
5     while ( cur ) {
6         if ( cur->va <= va && cur->va + cur->size > va ) break;
7         pre = cur;
8         cur = cur->next;
9     }
10    if ( cur == NULL ) {
11        panic( "fail to find specific block.\n" );
12    }
13    if ( cur == current->used_start ) {
14        current->used_start = cur->next;
15    } else {
16        pre->next = cur->next;
17    }
18    // 将块插入到空闲链表, 可以使用头节点置空操作简化逻辑
```

```
19     BLOCK *free_cur = current->free_start, *free_pre = current-
20     >used_start;
21     while ( free_cur ) {
22         if ( free_cur->va > cur->va ) {
23             break;
24         }
25         free_pre = free_cur;
26         free_cur = free_cur->next;
27     }
28     if ( free_cur == NULL ) {
29         if ( free_pre == NULL ) {
30             cur->next = current->free_start;
31             current->free_start = cur;
32         } else {
33             if ( free_pre->va + free_pre->size >= cur->va ) {
34                 free_pre->size += cur->size;
35             } else
36                 free_pre->next = cur, cur->next = NULL;
37         }
38     } else {
39         if ( free_cur == current->free_start ) {
40             if ( cur->va + cur->size >=
41                 free_cur->va )
42             {
43                 cur->size += free_cur->size;
44                 cur->next = free_cur->next;
45                 current->free_start = cur;
46             } else {
47                 cur->next = current->free_start;
48                 current->free_start = cur;
49             }
50         } else {
51             if ( free_pre->va + free_pre->size >= cur->va &&
52                 cur->va + cur->size >= free_cur->va )
53             {
54                 free_pre->size += cur->size + free_cur->size;
55                 free_pre->next = free_cur->next;
56             } else if ( cur->va + cur->size >=
57                 free_cur->va )
58             {
59                 cur->size += free_cur->size;
60                 cur->next = free_cur->next;
61                 free_pre->next = cur;
62             } else if ( free_pre->va + free_pre->size >=
63                 cur->va )
64             {
65                 free_pre->size += cur->size;
66             } else
67             {
68                 free_pre->next = cur;
69                 cur->next = free_cur;
70             }
71         }
72     }
```

```
71     }  
72  
73 }
```

高效的 *better_free* 函数实现

对于简单的 *free* 函数实现。因为分配空间时无脑分页，因此释放资源只需要调用 *user_vm_unmap* 函数，将用户虚拟地址空间中的物理页映射取消即可。

而对于 *challenge* 所实现的 *better_free* 函数，情况复杂得多。首先需要找到要释放的块，然后将其从已用块链表中移除，再将其插入到空闲块链表中。插入时需要考虑多种情况，比如要释放的块与空闲块链表中的块相邻，需要合并；或者要释放的块与空闲块链表中的块不相邻，需要插入到合适的位置。这样就实现了正确的 *better_free* 函数。

2.3 实验调试及心得

堆内存管理是最简单的实验。全程玩链表和哈希，不用管奇奇怪怪的指针，写好逻辑就可以。接下来叙述我在选择 *malloc* 策略的体会和心得：课上页分配内存给出了三种可行的策略：首次适应算法（选择第一个合适的连续内存块分配）；最佳适应算法（选择最小的合适内存块分配）；最差适应算法（选择最大的合适内存块分配）。观察题目要求，因为没有性能上的要求，同时要求分配的内存不出现缺页异常（也就是分配在同一页上），首次适应算法就非常合适。虽然会产生内存碎片，但是可以避免分页，既然实验要求就是不出现缺页异常，那么首次适应算法就是最好的选择。