目　　录

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。

```
riscv-pke on  lab1_challenge2_errorline
❯ 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.
```

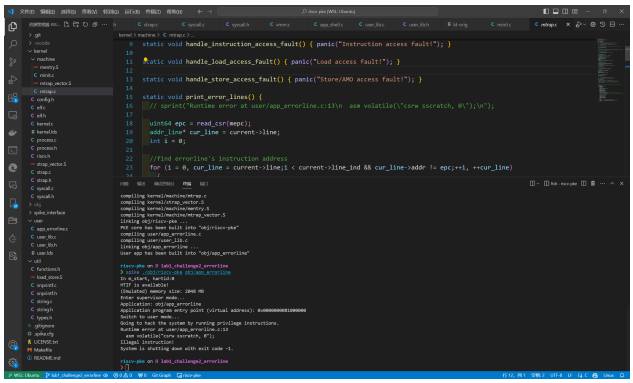

图 1: 正确的输出 lab1_challenge2

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。

```
// added @lab1_challenge2 | load debugline
elf_status load_debugline( elf_ctx *ctx ) {
    // sprint( "hello load_debugline\n");
    // sprint("%d\n",ctx);
    // claim string table header
    elf_sect_header strtab;
    // get string table header's info
    int offset;

    offset = ctx->ehdr.shoff;
    // sprint("%d\n",ctx->ehdr.version);
    // sprint( "%d\n", offset );
    // sprint( "%d\n", ctx->info );
    offset += sizeof( strtab ) * ( ctx->ehdr.shstrndx );
    // sprint( "%d\n", offset );

    // save string table header
   if ( elf_fpread( ctx, (void *) &strtab, sizeof( strtab ), offset ) !
=
         sizeof( strtab ) ) {
        panic( "string table header get failed!\n" );
    }

    // save string table
    static char strtab_info[ STRTAB_MAX ];
       if  (  elf_fpread(  ctx,  (void  *)  strtab_info,  strtab.size,
strtab.offset ) !=
         strtab.size ) {
        panic( "string table get failed!\n" );
    }

    // get .debug_line segment
    elf_sect_header debugseg;
    int index;
   for ( index = 0, offset = ctx->ehdr.shoff; index < ctx->ehdr.shnum;
          index++, offset += sizeof( debugseg ) ) {
                  if   (   elf_fpread(   ctx,   (void   *)   &debugseg,
sizeof( elf_sect_header ),
                         offset ) != sizeof( elf_sect_header ) )
```

```
37              panic( "debug header get failed!\n" );
38          // sprint("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。

```
static void print_error_lines() {
    //  sprint("Runtime  error  at  user/app_errorline.c:13\n    asm volatile(\"csrw sscratch, 0\");\n");

  uint64 epc = read_csr(mepc);
  addr_line* cur_line = current->line;
  int i = 0;

  //find errorline's instruction address
   for (i = 0, cur_line = current->line;i < current->line_ind && cur_line->addr != epc;++i, ++cur_line)
    {
    //sprint("i=%d,addr=%x\n",i,cur_line->addr);
    if (cur_line->addr == epc) break;
    }
  if (i == current->line_ind) panic("can't find errorline!\n");
  //sprint("find errorline,i=%d\n",i);


//find file's path and name
  char filename[FILENAME_MAX];
  //find filename
  code_file* cur_file = current->file + cur_line->file;
  char* single_name = cur_file->file;
  //find file's path
  char* file_path = (current->dir)[cur_file->dir];
  //combine path and name
  int start = strlen(file_path);
  strcpy(filename, file_path);
  filename[start] = '/';
  start++;
  strcpy(filename + start, single_name);
  sprint("Runtime error at %s:%d\n", filename, cur_line->line);

  //find error line
    //error instruction's line
  int error_line = cur_line->line;
  //open file
  spike_file_t* file = spike_file_open(filename, O_RDONLY, 0);
  if (IS_ERR_VALUE(file)) panic("open file failed!\n");
  //get file's content
  char file_detail[FILE_MAX];
  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      //sprint("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    sprint("%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。

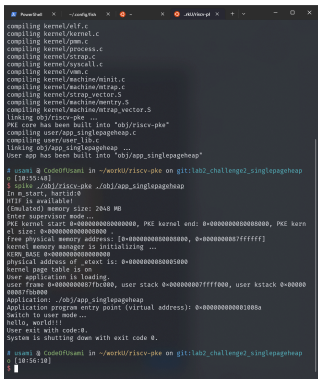

图 2: 正确的输出 lab2_challenge2

```
# usami @ CodeOfUsami in ~/workU/riscv-pke on git:lab2_challenge2_singlepageheap o [11:22:30]
$ spike ./obj/riscv-pke ./obj/app_singlepageheap
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x0000000080008000, PKE kernel size: 0x0000000000008000 .
free physical memory address: [0x0000000080008000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080005000
kernel page table is on
User application is loading.
user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user kstack 0x0000000087fbb000
Application: ./obj/app_singlepageheap
Application program entry point (virtual address): 0x000000000001008a
Switch to user mode...
hello, world!!!
User exit with code:0.
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。

```
//
// maybe, the simplest implementation of malloc in the world ... added @lab2_2
//
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;
}
```

原始的 *malloc* 函数实现

```
// added @lab2_challenge2
// allocate a block to user,size is n.  return virtual address
uint64 sys_user_better_allocate_page( long n ) {
    uint64 va = user_better_allocate( n );
    return va;
}
//
//
//

// added @lab2_challenge2
uint64 user_better_allocate( long n ) {

    // 以 8 为倍数向上取整
    n = ROUNDUP( n, 8 );
    if ( n > PGSIZE ) {
        panic( "fail to allocate PGSIZE due to too big size.\n" );
    }

    // 遍历块链表取取到一个合适的块
    BLOCK *cur = current->free_start, *pre = current->free_start;
    while ( cur ) {
        if ( cur->size >= n ) break;
        pre = cur;
        cur = cur->next;
    }

    // 页表耗尽则触发分页
    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,
(uint64) pa,
35                       prot_to_type( PROT_WRITE | PROT_READ, 1 ) );
36          // turn this new page to a block
37          BLOCK *temp = (BLOCK *) pa;
38          temp->start = (uint64) pa + sizeof( BLOCK );
39          temp->size = PGSIZE - sizeof( BLOCK );
40          temp->va = va + sizeof( BLOCK );
41          temp->next = NULL;
42          // 插入新块
43          if ( pre == NULL ) {
44              current->free_start = temp;
45              cur = temp;
46          } else {
47                if ( pre->va + pre->size >= ( ( pre->va ) / PGSIZE +
PGSIZE ) ) {
48                  pre->size += PGSIZE;
49                  cur = pre;
50              } else {
51                  pre->next = temp;
52                  cur = temp;
53              }
54          }
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
75      // 块置入已用链表
76      cur->size = n;
77      cur->next = current->used_start;
78      current->used_start = cur;
79      return cur->va;
```

```
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
```

```
        BLOCK *free_cur = current->free_start, *free_pre = current->used_start;
    while ( free_cur ) {
        if ( free_cur->va > cur->va ) {
            break;
        }
        free_pre = free_cur;
        free_cur = free_cur->next;
    }
    if ( free_cur == NULL ) {
        if ( free_pre == NULL ) {
            cur->next = current->free_start;
            current->free_start = cur;
        } else {
            if ( free_pre->va + free_pre->size >= cur->va ) {
                free_pre->size += cur->size;
            } else
                free_pre->next = cur, cur->next = NULL;
        }
    } else {
        if ( free_cur == current->free_start ) {
            if ( cur->va + cur->size >=
                 free_cur->va )
            {
                cur->size += free_cur->size;
                cur->next = free_cur->next;
                current->free_start = cur;
            } else {
                cur->next = current->free_start;
                current->free_start = cur;
            }
        } else {
            if ( free_pre->va + free_pre->size >= cur->va &&
                 cur->va + cur->size >= free_cur->va )
            {
                free_pre->size += cur->size + free_cur->size;
                free_pre->next = free_cur->next;
            } else if ( cur->va + cur->size >=
                        free_cur->va )
            {
                cur->size += free_cur->size;
                cur->next = free_cur->next;
                free_pre->next = cur;
            } else if ( free_pre->va + free_pre->size >=
                        cur->va )
            {
                free_pre->size += cur->size;
            } else
            {
                free_pre->next = cur;
                cur->next = free_cur;
            }
        }
```

```
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。