华中科技大学

操作系统原理课程设计报告

姓　　名：　沈子旭

学　　院：　计算机科学与技术学院

专　　业：　计算机科学与技术

班　　级：　计算机 2004 班

学　　号：　U202015396

指导教师：　胡贯荣

分数	
教师签名	

2023 年 4 月 7 日

目　录

实验一 打印用户程序调用栈

1.1 实验目的

通过完善 PKE 内核来实现 app_print_backtrace()函数。具体来说就是通过寻找用户态栈和 elf 文件的相应信息来打印用户态栈中所存储的函数调用情况，所打印函数名的数量由传入参数的大小决定。

1.2 实验内容

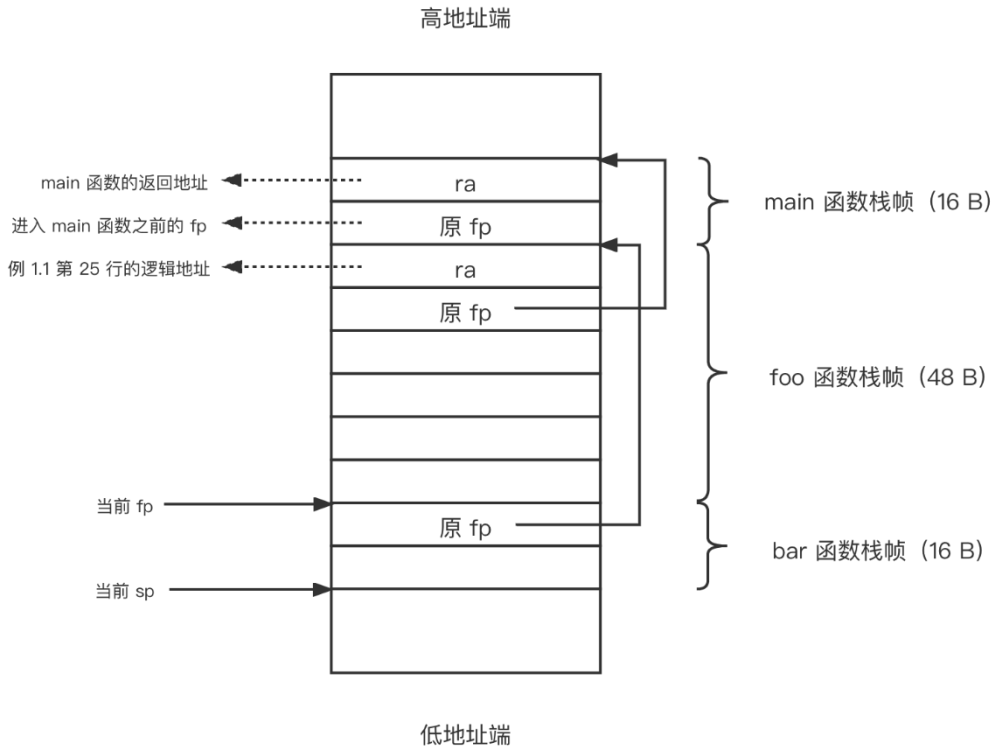
本次实验的思路主要包括三个主要部分。首先，第一部分的目标是完善函数 app_print_backtrace() 的系统调用路径，即从 user_lib 文件到 syscall 文件中的完整调用路径，这与基础实验中的类似。由于该部分内容过于简单且已在基础实验中介绍过，因此在此不再赘述。其次，第二部分的任务是从当前进程的结构中获取用户态栈，并确定回溯的层数和调用函数的返回地址。最后，第三部分的目标是从 ELF 文件中找到与返回地址对应的函数符号名称，并将其打印出来。有关第二部分和第三部分的详细信息将在下面的段落中进行阐述。

第二部分的任务需要掌握两个基础知识，一个是用户态栈在进程结构中的存储位置，另一个是用户态栈的基本结构。为了找到用户态栈的存储位置，首先需要了解栈底指针存储在哪里。

通过翻阅 RISC-V 寄存器对应表，我们不难发现栈底指针存储在通用寄存器中的 sp 寄存器中，故用户态栈可以通过以下语句寻得。

```
unit64 u_sp = current -> trapframe ->reg.sp;
```

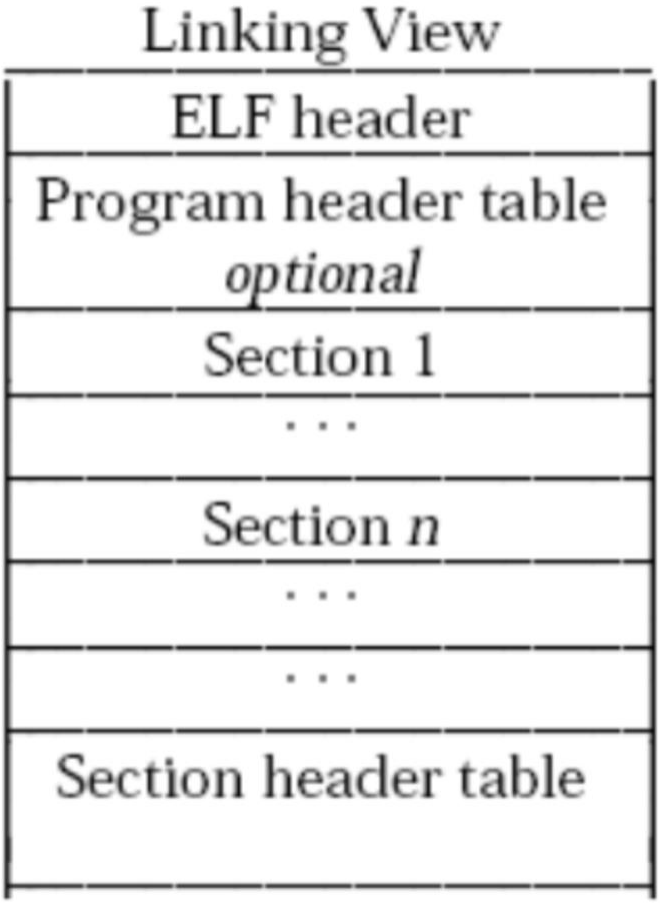
接着就是需要了解用户态栈的结构来找到各个调用函数的返回地址，即 ra，首先我们能通过指导文档找到用户态栈的结构：



从图中不难得出，如果已知每个调用函数的栈帧大小，则可以通过当前的 sp 向高地址累加栈帧大小来获取每个调用函数的返回地址 ra。由于本实验进行了简化，因此所有调用函数的栈帧大小都为 16B。因此，如果想要提取每个调用地址，只需要找到最底部的 ra，然后不断累加 16 即可得到每个调用地址。下面是相关的代码段：

```
unit64 u_sp = current -> trapframe -> regs.sp;
unit64 ra = u_sp + 40;
unit64 i = 0;
for(unit64 tmp_ra = ra; i < depth; tmp_ra += 16, i++)
```

第三部分中，首先先要了解 elf 文件的具体结构，依然根据指导文档可知，其结构图如下：



不难从图中发现，每个部分的内容对应于每个区段，每个区段的相关属性（如大小、偏移量、段名等）都存储在称为 section header 的结构中。这些结构组合在一起形成 section header table 结构，存储在文件的末尾。以下是 section header 的结构：

```
// elf section header
typedef struct elf_sect_header_t{
    uint32 name;
    uint32 type;
    uint64 flags;
    uint64 addr;
    uint64 offset;
    uint64 size;
    uint32 link;
    uint32 info;
    uint64 addralign;
    uint64 entsize;
} elf_sect_header;
```

提取 ELF 文件中相关地址对应的符号名，需要大致分为三个步骤。首先，需要

在 ELF 文件尾部的 section header table 中找到名为.shstrtab 的段的 section header。具体步骤是，先通过 shoff 查找 section header table 的偏移值，然后通过 shstrndx 找到.shstrtab 段对应的 section header 的索引，最后通过 shentsize 找到每个 section header 的大小。如此一来，就能够使用以下代码找到.shstrtab 段的 section header：

```
unit64 offset = ctx -> edhr.shoff + ctx -> edhr.shstrndx * ctx -> edhr.shentsize;
```

第二步是需要了解 .shstrtab 段的结构，该段中存储了每个段名的 ASCII 码，并使用换行符 \n 进行分隔。每个段名所对应的在 .shstrtab 段的偏移值与其对应的 section header 中的 name 段相等。因此，我们可以通过在 .shstrtab 段中查找 .strtab 和 .symtab 这两个字符串，再通过它们的偏移值找到对应的 section header，最后从 section header 中找到相应的偏移值从而找到这两个段。

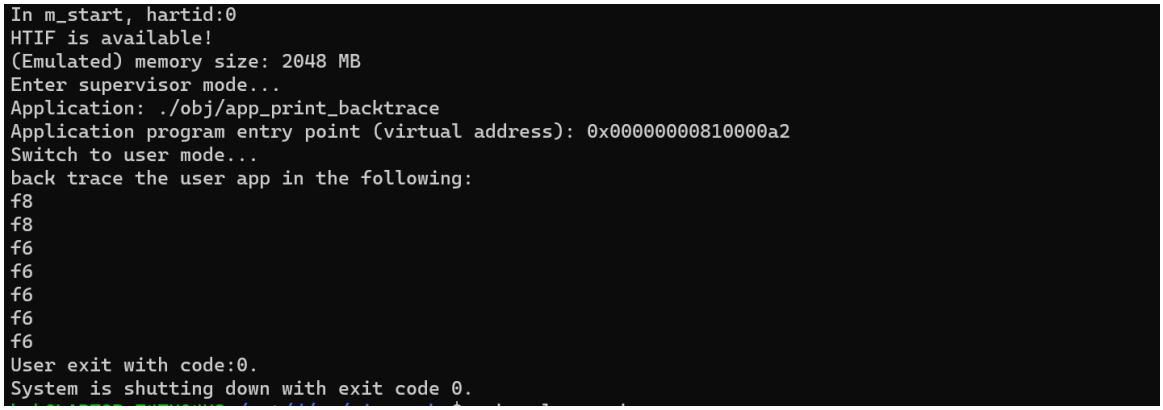
第三步是如何找到返回地址所对应的调用函数的符号名。首先，需要了解.strtab 和.symtab 这两个段的结构。其中，.symtab 段的基础结构如下所示，其中的 value 项对应了该符号的地址：

```
typedef struct {
    uint32 st_name;
    unsigned char st_info;
    unsigned char st_other;
    uint16 st_shndx;
    uint64 st_value;
    uint64 st_size;
} Elf32_Sym;
```

.strtab 段存储着每一个符号名的 ascall 码，并用‘\n’所隔离，每一个符号名所对应的在.strtab 段的偏移值与其 symbol table 中每一个结构的 name 段相等。因此，可以通过在.symtab 段中比较返回地址 ra 是否在某一个调用函数的开始地址和结束地址之间来找到其所对应的调用函数，然后再通过其 name 字段在.strtab 段找到相应的对应符号名。

1.3 实验调试及心得

这次实验所出现的问题主要是出在最后通过 ra 找对应的 symbol 结构中出现了问题，当时的问题输出如下：



经过调试后发现，问题出在对每一个 symbol 结构的开始地址和结束地址的判断上，并没有考虑到它们是连续的结构，导致地址没有被正确累加。在增加了一个 boundary 来处理地址后，问题得到了解决。

本次实验难度较大，主要在于对 ELF 文件结构的理解。为了理解该结构，我在网络上查找相关的博客和阅读了 Linux 相关文档。经过多次尝试后，我才最终理清了相关的结构。在这一点上，我认为实验文档中的提示还有很大的改进空间。但总的来说，通过本次实验，我对用户态栈的理解以及对 ELF 文件结构的理解都更深入了。我对整个程序的运行过程也有了更细致的了解，因此受益匪浅。但是，我希望课题组能够在后续实验中继续努力，对于实验文档中涉及到提示的相关结构，例如 section header 和 symbol table 段的结构，可以直接给出，这样有助于学生更方便地调用和学习。

实验二 打印异常代码行

2.1 实验目的

修改内核（包括 machine 文件夹下）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。

2.2 实验内容

本次实验的思路主要是分为两部分，分别是找到并存储 elf 文件中的.debug_line 段和通过其中的信息和寄存器中存储的中断代码对应的地址找到相应的文件和异常代码的行数并输出相关信息。

第一部分是找到并解析 elf 文件中的.debug_line 段，并将其存储。为了实现这一点，我们需要先在 elf 文件的 section header table 中找到.shstrtab 段的 section header。具体步骤是，通过 shoff 找到 section header table 的偏移值，再通过 shstrndx 找到.shstrtab 段对应的 section header 的索引，再通过 shentsize 找到每一个 section header 的大小，从而就可以找到.shstrtab 段的 section header。接着需要了解.shstrtab 段的结构，其中存储着每一段名的 ASCII 码，并用'\n'所隔离，每一个段名所对应的在.shstrtab 段的偏移值与其 section header 中的 name 段相等。因此，我们可以通过在.shstrtab 段中寻找.debug_line 这一字符串，再分别通过其偏移值找到对应 name 的 section header，然后从 section header 中找到相应的 offset，最终找到这个段。然后我们调用所提供的 make_addr_line()函数并传入相应参数进行解析即可，主要代码如下：

```
char name[16]; ((elf_info *)ctx->info)->p->debugline = NULL;
elf_sect_header name_seg, tmp_seg;
// read name segment
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;
// find ".debug_line" segment
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 *)name, 20, name_seg.offset + tmp_seg.name);
    if (strcmp(name, ".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;
    }
}

return EL_OK;
```

第二部分的任务是从已解析的数据中找到与中断地址对应的指令、其所在文件的路径和文件名，以及输出错误代码。首先，需要从 MEPC 寄存器中读取中断地址 addr，然后在 line 数组中找到具有相同地址的行和对应的文件索引。接着，

在 file 数组中找到该文件对应的路径索引，并从 dir 数组中找到完整路径。这样，我们就可以获得错误代码所在的文件名和路径，以及代码在该文件中的行数。。其代码如下：

```
void error_print() {
    uint64 mepc = read_csr(mepc);
    for (int i = 0; i < current->line_ind; i++) {
        // find the exception line table entry
        if (mepc < current->line[i].addr)
        {
            line_print(current->line + i - 1);
            break;
        }
    }
}
```

然后就是如何打印错误代码的步骤了，我们需要使用 spike_file_pread()等函数来打开文件并读取其中的代码。具体思路是，我们先读取文件中超过一行数量的字符，然后以换行符'\n'为分隔符，记录下每一行对应的字符偏移值，并进行存储。最后，我们只需要输出指定行的代码即可。

2.3 实验调试及心得

这次实验过程中在当时存储.debug_line 段的时候出现了问题，当时的报错情况如下：

```
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
Application: ./obj/app_errorline
Load access fault!
System is shutting down with exit code -1.
```

通过搜索相关资料查明可能原因是在没有相应权限的情况下调用了高权限的内存或者是所使用的物理内存不存在导致，经过排查，发现是留给.debug_line 段的空间大小不够，在尝试申请更多空间后得到了解决。

关于这次实验的心得和建议，首先我觉得这次实验能够充分的用到前面实验涉及到的知识是非常的有意义的，这不仅能够加深理解，也能拓宽对于某一知识的运用，此外，对于相应的文件读取函数的提供也是非常合适的。总而言之，通过这一实验，我对于中断、错误信息调试和字符串匹配等知识都有了更加深入的理解。但是在提交至 educoder 平台后发现本机环境和 educoder 环境略有些不同，导致需要微调代码，不知道这是否需要对 educoder 的环境进行进一步的完善。

实验三 堆空间管理

3.1 实验目的

通过修改 PKE 内核中的 malloc 和 free 函数使得当程序申请分配空间时能够相对紧凑的分配，并且修改分配空间的算法使得分配空简尽量分配最符合需求大小的空间。

3.2 实验内容

为了更细化存储空间的基础空间单位大小，并对此设计一套新的结构和分配、释放算法，本次实验的主要思路如下。

首先，我们基于原本对页进行分配操作的基础上，引入一个内存控制块（Memory Control Block，MCB）的设计。具体来说，对于每一页中，我们会为每个申请的空间附加一个 MCB 结构，用于记录相关信息以及指向下一块可用空间，MCB 的结构如下：

```
typedef struct mem_control_block{
  uint64 flag;
  uint64 size;
  uint64 addr;
  struct mem_control_block* next;
}mcb;
```

在这个设计中，flag 是一个标识位，其中 0 表示空间已经被释放，可以被分配给需要的内存，而 1 则表示该空间仍然被占用。size 表示该段空间的大小，addr 表示该段空间的开始地址（逻辑地址）。此外，我们需要在进程中引入三个地址，分别表示用户当前可分配地址的物理地址和逻辑地址，以及当前已分配页，即堆（heap）的上限。每次进行内存分配时，首先比较所需空间加上当前可分配地址是否小于堆顶。如果大于堆顶，则需要申请新的页并修改堆顶；如果小于堆顶，则在已有的进程块中寻找是否有合适大小的可用空间。如果没有合适的空间，则需要新申请一个 MCB 并进行存储。这其中涉及到的相关函数包括申请页面、新建 MCB 以及寻找合适的 MCB 块，其中的代码分别如下：

```
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;
}
```

```
uint64 sys_user_allocate_block(uint64 n) {
  uint64 va = -1;
  int sys_user_allocate_block_i,sys_user_allocate_block_j,sys_user_allocate_block_k;
  sys_user_allocate_block_k=0;
  mem_block block;
  if(!mem_blocks_numbers) {
    block.flag = 1;
    block.start = (uint64)alloc_page();
    block.end = block.start + n;
    mem_block_start = block.start;
    mem_block_end = block.end;
    mem_blocks[0] = block;
    mem_blocks_numbers++;
    va = g_ufree_page;
    va_start = g_ufree_page;
    g_ufree_page += PGSIZE;
    user_vm_map((pagetable_t)current->pagetable, va, PGSIZE, (uint64)block.start, prot_to_type(PROT_WRITE | PROT_READ, 1));
  }
```

```
  else {
    for(sys_user_allocate_block_i = 0; sys_user_allocate_block_i < mem_blocks_numbers; sys_user_allocate_block_i++)
    {
      if(!mem_blocks[sys_user_allocate_block_i].flag && mem_blocks[sys_user_allocate_block_i].end - mem_blocks[sys_user_allocate_block_i].
start > n)
      {
        block.start = mem_blocks[sys_user_allocate_block_i].start;
        block.end = block.start + n;
        block.flag = 1;
        mem_block block1;
        block1.start = block.end;
        block1.end = mem_blocks[sys_user_allocate_block_i].end;
        block1.flag = 0;
        for(sys_user_allocate_block_j = sys_user_allocate_block_i; sys_user_allocate_block_j < mem_blocks_numbers - 1;
sys_user_allocate_block_j++) mem_blocks[sys_user_allocate_block_j] = mem_blocks[sys_user_allocate_block_j + 1];
        mem_blocks_numbers--;
```

经过实验可知，本次实验使用了首次匹配算法进行空间分配。具体地，在寻找可用空间块时，从第一个可用块开始匹配，找到满足要求的空间块后进行分配。虽然这种算法可能会造成一定的空间浪费，但可以避免其他算法中对空间进行再排序所产生的时间浪费。

在空间释放方面，本次实验采用了二级回收策略。具体来说，当一个空间块被释放后，它会被标记为可用状态（flag 置零），而不是立即回收空间。当剩余可用空间满足一页时，再进行回收。这种设计避免了频繁对内存进行操作，同时考虑到往往连续申请的空间大小相近，这种策略也可以从某种程度上减少对内存的开销，具体代码如下：

```
uint64 sys_user_free_block(uint64 va) {
  uint64 pa = (uint64)user_va_to_pa(current->pagetable, (void *)va);
  for(int i = 0; i < mem_blocks_numbers; i++) {
    if(mem_blocks[i].start == pa) {
      mem_blocks[i].flag = 0;
      return 0;
    }
  }
  panic("free false!");
  return 0;
}
```

3.3 实验调试及心得

在本次实验中，我遇到了一个问题，就是一开始没有考虑到地址对齐的问题，导致程序出现了相关的报错。后来，我通过增加 size + 8 – size%8 的操作，将地址与逻辑地址对齐，解决了这个问题。

通过本次实验，我对 malloc 和 free 的原理有了更深入的理解，也学习和掌握了 Linux 分配内存空间的相关知识。实践操作使我更深入地理解了段页式存储的概念和基础实现，收获颇丰。总之，本次实验让我从实践中学到了很多可以补充理论的知识，收益匪浅。

实验四 实现信号量

4.1 实验目的

通过完善主函数中对于信号灯操作相关的代码的系统调用，实现信号灯相关操作的完整内容。

4.2 实验内容

本次实验的思路比较简单，需要实现三个内容，分别是信号灯的初始化、信号灯的 P 操作和 V 操作。

首先，要实现信号量结构体，主要由值、占位符以及所在进程队列的头尾两指针组成，其结构如下：

```
typedef struct signal_light {
  int value;
  int flag;
  process *head;
  process *tail;
}signal;
```

接着就是新建信号量的操作，也就是对每个值进行初始化，没什么特别多说明的必要：

```
int init signal(int value){
  for (int i = 0; i < NPROC; i ++)
  {
    if(!signals[i].flag){
    signals[i].flag =1;
    signals[i].value = value;
    signals[i].head =signals[i].tail =NULL;
    return i;
    }
  }
return 0;
}
```

然后便是信号灯的 P 操作和 V 操作，当执行 V 操作时，首先将信号量的值加一，然后判断该信号量是否有等待的进程，如果有，就从等待队列中取出队头进程，将其状态修改为 READY，并加入到调度队列中。而执行 P 操作时，先将

信号量的值减一，如果减一后信号量的值小于 0，说明当前进程需要被阻塞并等待信号量的增加。此时，当前进程的状态被修改为 BLOCKED，并被加入到该信号量的等待进程队列中。然后，触发 CPU 的调度，由于当前进程已经不在调度队列中，CPU 会执行其他用户程序。相关代码如下：

```
int sem v(int sem) {
  signals[sem].value ++;
  if(signals[sem].head)
  {
    process*tmp=signals[sem].head;
    signals[sem].head=tmp->queue next;
    if(tmp->queue_next == NULL) signals[sem].tail =NULL;
    insert_to_ready_queue(tmp);
  }
  return 0;
}
```

```
int sem_P(int sem) {
  signals[sem].value --;
  if(signals[sem].value<0)
  {
  if(signals[sem].head ==NULL)
  {
    signals[sem].head =signals[sem].tail = current;
    current->queue_next =NULL;
  }
  else
  {
    signals[sem].tail->queue next=current->queue next;
    signals[sem].tail = current;
  }
  current->status=BLOCKED;
  schedule();
  }
  return 0;
}
```

4.3 实验调试及心得

由于本次实验的逻辑和要点都是课程的重点内容，因此最终实现起来相对容易，并且没有出现调试方面的问题。通过本次实验，我对信号量的相关知识有了

更深入的了解。总的来说，学习理论知识时，实践是必不可少的，只有通过实践，才能更加深刻地理解和掌握所学的知识。