



# 华中科技大学

## 操作系统原理课程设计报告

姓 名： rai ndaydream  
学 院： 计算机科学与技术学院  
专 业： 计算机科学与技术专业  
班 级： CS2009  
学 号：  
指导教师： 周正勇

分数	
教师签名	

2023 年 3 月 7 日

# 目 录

<b>实验一 打印用户程序调用栈 .....</b>	<b>1</b>
1.1 实验目的 .....	1
1.2 实验内容 .....	1
1.3 实验调试及心得 .....	4
<b>实验二 打印异常代码行 .....</b>	<b>6</b>
2.1 实验目的 .....	6
2.2 实验内容 .....	6
2.3 实验调试及心得 .....	7
<b>实验三 堆空间管理 .....</b>	<b>9</b>
3.1 实验目的 .....	9
3.2 实验内容 .....	9
3.3 实验调试及心得 .....	11
<b>实验四 实现信号量 .....</b>	<b>12</b>
4.1 实验目的 .....	12
4.2 实验内容 .....	12
4.3 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 `print_backtrace()` 函数，使其可以回溯并打印用户进程的函数调用情况。应用程序调用 `print_backtrace()` 函数时，应能够通过控制输入的参数控制回溯的层数，使其最深回溯至 `main` 函数。

因此在该实验中需要掌握以下知识：

1. 了解系统调用的方法，并实现回溯函数的系统调用路径；
2. 掌握用户栈的使用，可以在操作系统内核中获取用户程序的栈；
3. 掌握 `elf` 文件头结构，了解文件头数据信息并掌握其中关键数据；
4. 掌握符号表和字符串表的使用，学习字符串表头结构、符号表结构并实现，并实现虚拟地址与源程序符号的转换。

## 1.2 实验内容

### 1.2.1 系统调用

系统调用主要是实现 `main` 中调用的函数并通过系统调用函数进入操作系统内核调用内核的代码来实现想要的功能。

#### 1. `print_backtrace()` 函数

函数 `f8` 调用 `print_backtrace` 函数并向其传递了一个参数，该参数表示回溯的深度，如果深度不超过 `main` 的深度，将按照该参数回溯；如果超过 `main`，将回溯到 `main` 停止。

而在回溯的时候，操作系统会陷入到内核切换到 `S` 模式，因此需要通过 `do_user_call` 函数进入 `S` 模式。`do_user_call` 函数需要一个系统参数表示当前系统调用所调用的功能，因此可以定义一个参数 `SYS_user_backtrace` 来表示 `backtrace` 的参数，同时需要将深度参数传递给该函数，这里我们将其传递给 `a1`。

#### 2. 系统调用函数

`do_user_call` 函数将参数传递给 `a0-a7`，并调用 `do_syscall` 函数实现系统调用，在 `print_backtrace` 中我们传递了系统功能参数和深度参数，因此在 `syscall` 中，我们需要根据这两个参数实现回溯功能。由于系统功能参数需要自己定义，因此需要对 `SYS_user_backtrace` 进行定义，在 `syscall.h` 中定义：

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

并在 `do_syscall` 函数中根据该参数值实现功能，这里我们将实现回溯功能的代码封装在 `ssize_t sys_user_backtrace(int num)` 函数中，将深度传递给回溯函数实现回溯。因此在系统调用函数中，对该功能的实现为：

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

## 1.2.2 回溯函数的实现

### 1. syscall 中的回溯函数

回溯函数需要调用 ELF 头文件，为了方便，我在这里将回溯的函数实现在了 ELF 结构同文件下，也就是 `elf.c` 中。因此需要在 `syscall.c` 文件中包含 `elf.h` 以方便调用函数。

在这里，我们首先对深度进行了判断，由于深度的合法范围为大于 0，因此在这里对深度进行了判断。

### 2. elf 中的回溯函数 backtrace

要打印函数调用过程中的函数名，需要找到用户函数在调用的时候所使用的栈、文件的符号表、字符串表。

#### (1) 用户栈的获取

通过文档和查询的资料，我们可以知道用户栈的栈顶指针保存在寄存器 `s0` 中，因此可以通过查询当前进程 `current` 的寄存器变量找到用户栈。

然而 `s0` 所指向的并非 `f8` 的函数地址，而是当前压栈的叶子节点的父节点的 `fp` 变量，最后压栈的是 `do_user_syscall` 函数，因此在函数调用过程中，`main->f1->f2->f3->f4->f5->f6->f7->f8->print_backtrace` 的返回地址 `ra` 和该函数所使用栈的地址 `fp` 都会压栈，而 `s0` 所指向的栈顶保存的即是 `print_backtrace` 的 `fp`。

在 `main->f1->f2->f3->f4->f5->f6->f7->f8` 中，分别调用了 `f1->f2->f3->f4->f5->f6->f7->f8->print_backtrace`，因此后者的返回地址将处于前者的代码段中，因此我们可以获取后者的返回地址并利用该特性进行范围比较来获取当前回溯的函数的函数名。

#### (2) elf 文件结构、符号表和字符串表的获取

要想获取函数名，需要找到字符串表以及函数名对应的偏移地址，该偏移地址记录在符号表中，因此需要获取符号表，这两个信息都包含在 `elf` 文件头中。

在 `elf` 文件头中包含了很多 `section` 的信息，其中的 `symtab` 和 `strtab` 分别是符号表表头的信息和字符串表表头的信息，这正是实现回溯函数所需要的信息。

字符串表表头和符号表表头都是 `section header table`，需要定义，查询资料知道表头结构为：

```
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;

```

字符串表就是简单的字符串，只需要 char 结构即可，符号表需要保存一些信息，查询资料知道其结构为：

```

typedef struct elf_sym_t {
    uint32    st_name;
    uint8     st_info;        /* the type of symbol */
    uint8     st_other;
    uint16    st_shndx;
    uint64    st_value;
    uint64    st_size;
} elf_sym;

```

其中符号表保存了一个函数的起始地址、代码段长度、名字，而该名字即为字符串表中的偏移地址。

此外还有一些信息，比如符号表表头和字符串表头的 type 值、字符串表的长度、函数占用用户栈的长度、函数类型转换操作，都需要在头文件中完成定义。

### (3) backtrace 函数的实现

首先需要将进程初始化的时候读取的 elf 文件头变量全局化从而使得 backtrace 函数可以使用。

①获取字符串表表头信息：表头信息以 elfboarder 中的 shoff 为起始地址开始，因此可以从该起始地址开始，调用 elf\_fpread 函数遍历所有表头，查询表头类型为符号表表头（SHT\_STRTAB）的表头并将其保存在 elf\_sect\_header 的 strtab 表头中；

②获取字符串表：字符串表表头包含了字符串的地址信息 offset，因此可以通过 elf\_fpread 函数以及 offset 的起始地址读取到字符串表的内容；

③获取符号表表头信息：与获取字符串表表头信息步骤相同，但是需要查询类型为 SHT\_SYMTAB 的表头；

④获取符号表：符号表表头中包含了单个符号表项的大小和整个符号表的大小，将其中所有的符号表表项都读取出来，因此可以从表头的 offset 起始地址开始，逐个的调用 elf\_fpread 函数读取符号表表项并保存下来，由于这里需要保存的是 main-f8 函数的信息，因此可以通过判断函数类型来决定是否保存，因此需要调用 ELF64\_ST\_TYPE 宏并传递该函数的标志信息 info 并与 STT\_FUNC 比较来判断当前的函数是否为 main-f8 中的函数，如果是就将其保存到符号表数组中；

⑤获取返回第一个返回地址：s0 指向 print\_backtrace 的 fp，因此需要将其向高地址移动 8 个字节来指向 print\_backtrace 的返回地址 ra；

⑥循环实现回溯：以 f8 的打印举例，由于 print\_backtrace 函数的返回地址在 f8 的代码段中，以及每一个符号表表项中保存了该函数的起始地址和代码段长度，因此遍历查找哪个符号表表项的代码段地址包含 print\_backtrace 的返回地址，该符号表表项就是 f8 的符号表表项，找到该表项后，根据表项中的 name 偏移地址，从字符串表中找到“f8”的保存位置并打印；

⑦根据深度和当前查询到的函数名是否为“main”来判断回溯是否结束。

## 1.3 实验调试及心得

该实验最重要的就是最终的 backtrace 的实现。

第一个是符号表的获取，在一开始，我只知道 elf 文件中给了字符串表表头的偏移信息，并没有提供符号表表头的信息，在和同学交流并查阅了资料以后我知道了表头中包含了 type 信息可以判断表头信息，其中符号表表头的 type 值为 2，字符串表表头的 type 值为 3，因此解决了寻找表头的问题。

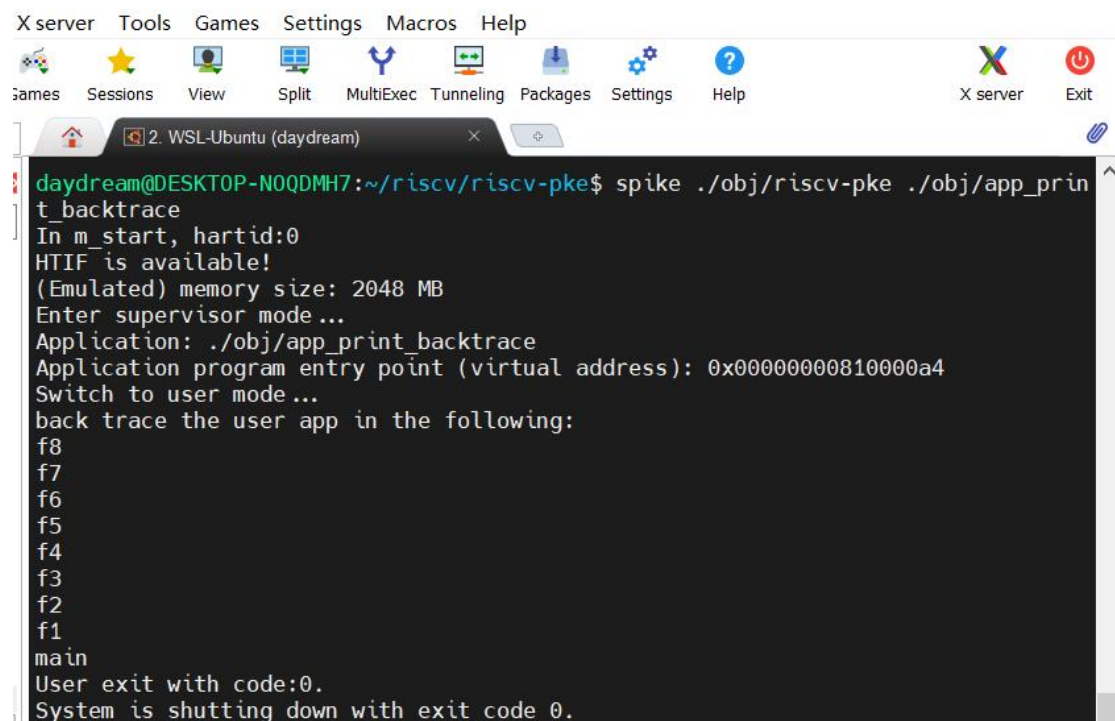
第二个是用户栈的获取，在该程序中，每个函数在调用的时候都使用了栈的两个 uint64 的位置，因此在遍历的过程中每次对指针加 2 即可，但是这个方法只适用于使用栈两个字节的情况，不具有普适性；另外一点就是返回地址获取后的判断，在一开始我使用的是反汇编找到函数开头和返回地址的关系进行判断的，但是该情况只适合函数参数满足一定条件的情况，也不具有普适性。在知道了这些问题之后，我对实现方案进行了调整。

在符号表中会保存一个函数的函数地址和代码段所使用字节的长度，另外在调用过程中，这些函数是由高地址向低地址分配代码段空间的，因此可以利用这一特点，将函数寻找的判定条件由匹配函数地址改为匹配返回地址是否在函数代码段中，这样不管参数如何变化，都可以准确的找到函数；而对于函数的返回地

址，根据上述方法首先可以找到 f8 的符号表，在这里我们可以找到 f8 的函数地址并保存，之后通过对该地址增加代码段长度来找寻下一个函数的代码起始地址，这样就可以避免由于参数压栈也导致栈指针无法通过单纯的加 2 获取下一个返回地址。

在写这个实验的过程中，我也遇到了一些问题，比如 elf 头文件一开始并不太清楚，看了教程也有点糊涂，特别是符号表和字符串表以及它们的表头之类的信息，需要多次查询，有点绕，有点类似于中断向量表的原理，但是在这个过程中，我和同学朋友一起交流，大家交流自己的困惑并一起学习，最终这些问题得以解决。

```
2  * Below is the given application for lab1_challenge1_backtrace.
3  * This app prints all functions before calling print_backtrace().
4  */
5
6  #include "user_lib.h"
7  #include "util/types.h"
8
9  void f8() { print_backtrace(100); }
10 void f7() { f8(); }
```



```
daydream@DESKTOP-NOQDMH7:~/riscv/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): 0x00000000810000a4
Switch to user mode...
back trace the user app in the following:
f8
f7
f6
f5
f4
f3
f2
f1
main
User exit with code:0.
System is shutting down with exit code 0.
```

图 1.1 回溯深度和运行结果

# 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。

需要进行的工作：

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（可以保存在用户程序的地址空间中）；
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，注意，连续行号对应的不一定是连续的地址，因为一条源代码可以对应多条指令；
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源代码行号和文件名输出。

## 2.2 实验内容

### 2.2.1 读取 debug 段信息

在该实验的文件结构中，每一段都有自己的头部信息，保存了段名、偏移地址和长度等信息，其中段的名称并非直接保存在头部信息中，而是保存在字符串表中，段名保存了其段名相对于字符串表起始的相对偏移地址。

因此读取 debug 段的信息需要：

①读取字符串表：根据文件头保存的字符串表头部位置 shstrndx 找到该头部的起始地址，并调用 elf\_fpread 函数读取出头部信息；根据该头部信息再次调用 elf\_fpread 函数读出字符串表并将其保存在字符串数组中。

②循环查找 debug 段：对段的头部信息进行遍历，检测出段名为“.debug\_line”的头部信息即可。

③读取 debug 段的信息：调用 elf\_fpread 函数读取出②中头部信息表示的 debug 段并将其保存在字符串数组中。

### 2.2.2 对 debug 段信息进行解读

在 1 中读取出 debug 段的信息后，需要调用 make\_addr\_line 函数解析 debug 段信息的数组，在解析的过程中，会将“指令地址-代码行-文件表”保存在当前



进程的 line 信息中。

该解读操作需要在读取 debug 段之后进行，因此需要在合适的位置调用两个函数：load\_debugline（读取 debug 段）和 make\_addr\_line（解析 debug 段）函数。

通过编译运行代码可以发现，在执行某个文件之前，会调用 load\_bincode\_from\_host\_elf 函数加载文件再进行接下来的操作，因此可以在该函数中调用这两个函数将文件的相关信息保存在进程中。

这两个函数的调用位置应在 load\_bincode\_from\_host\_elf 函数的最后，同时，由于两个函数一前一后的一次执行，因此可以在 load\_debugline 函数的最后直接调用 make\_addr\_line 实现读取和解析，并在 load\_bincode\_from\_host\_elf 函数中只调用 load\_debugline 即可。

### 2.2.3 异常处理：打印异常代码行

在运行该文件之前 load\_bincode\_from\_host\_elf 函数已经将相关信息保存在了进程 process 结构体中，其中最重要的信息就是 line 保存的“指令地址-代码行-文件表”。

整个处理的流程为：

①遍历进程的 line，找到指令地址与异常代码的指令地址相同的指令信息 (cur\_line)；

②根据 cur\_line 中的 file 地址和进程中的 file 信息起始地址找到该代码行所在文件的信息 (cur\_file)；

③根据该文件信息中的 dir 和进程中的文件路径保存位置 dir 找到该文件的文件路径；

④将文件路径和文件名拼接得到文件的整体路径，同时异常代码的所在位置即为①中得到的信息中的 line；至此得到代码出错的文件+路径+位置；

⑤根据④中得到的路径，打开文件，调用函数将其中的内容读出来；

⑥根据 cur\_line 中的 line 知道出错代码行的位置 line，对⑤中得到的内容进行遍历：以换行符为一行的结束进行判断找到 line 的起始位置；

⑦将该文件从⑥中找到的起始位置开始打印出来，直到遇到换行符。

## 2.3 实验调试及心得

在整个实验中，我遇到了如下问题：

1. 不可以对字节头进行遍历判断 type 找字符串表：因为一个文件有多个字符串表。在本实验中，保存段的名字的字符串表为第二个字符串表，因此如果采用遍历查找想要的字符串表的头部信息只能查找到第一个字符串表也就是文件名等字符串从而导致出错。

2. 保存 debug 段信息的 char 数组的数组大小应该开的大一些, 8000 能过 educoder, 否则就会出现各种各样的问题, 包括但不限于: make\_addr\_line 死循环、加载失败、AMO 等等。

3. 找当前指令地址的时候, 不可以直接使用 epc, 需要调用宏 read\_csr(mepc), 二者的值不相同。

4. 构造表被破坏: 指令地址-代码行-文件这个表被破坏, 原因: 保存 debug 段文件的字符串用的是 char, 用 static char 就正确了。

5. 代码行位置从 1 开始。

在写该实验的过程中, 问题 4 是让我 debug 了最久的一个错误。由于不能设置断点调试, 因此只能在某些位置调用 sprintf 函数输出不同的信息进行调试。在调试该问题的过程中, 发现在读取解析 debug 段信息结束的时候“指令地址-代码行-文件表”是正确的, 即在 elf.c 文件执行的最后输出了该表; 但是在打印异常代码时输出该表发现出现了错误, 该表的前面几个表项被改动了, 即在 mtrap.c 运行的开始对该表输出。在调试后总结下来, 可能是某些数组开的太大导致了数据冲突, 将 debug 段的信息所在的数组定义为静态变量便解决了这个问题。

```
daydream@DESKTOP-NOQDMH7:~/riscv/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 运行输出

# 实验三 堆空间管理

## 3.1 实验目的

挑战任务是修改内核(包括 machine 文件夹下)的代码,使得应用程序的 malloc 能够在一个物理页中分配,并对各申请块进行合理的管理。通过修改 PKE 内核(包括 machine 文件下的代码),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。

需要做的工作:

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,后续对于 heap 的扩展,需要对新增的虚拟地址添加对应的物理地址映射。
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。

## 3.2 实验内容

在该实验中,可以使用首次适应算法实现空闲块的分配,也可以采用最佳适应算法,在我的实现中,我采用了首次适应算法分配并从低地址开始分配。

需要实现的有:

### 3.2.1 增加 block 数据结构

在该块中需要保存该块的起始位置、块大小和下一个块的起始位置,同时由于在内核中访问的都是物理实址,因此起始位置和下一块都保存的是物理实址,而用户态使用的是逻辑地址,因此在结构中保存了一个虚拟地址,该虚拟地址与起始物理实址相对应。

具体结构为:

```
typedef struct block{
    uint64 start;//block's physical address
    uint64 size;//block's size
    uint64 va;//block's virtual address
    struct block *next;
}BLOCK;
```

### 3.2.2 对 process 增加空闲 block 队列和使用 block 队列

一个进程不再以一个页面为单位，而是以块为单位，同时块不是固定大小的，因此需要在进程 process 结构中加一个空闲块链表和使用块链表保存使用的信息和释放的信息。

### 3.2.3 对 malloc 和 free 进行系统调用完善

将原本调用的函数该为我自己编写的 do\_user\_allocate 和 do\_user\_free 函数即可。

### 3.2.4 Malloc 实现：

①判断 size 是否合法：即一次性所请求的大小是否超出了一个页面，如果是的话，调用 panic 输出提示；

②查找是否有满足的空闲 block：对 free 的块链表进行遍历，找到第一个可以满足需求大小的块，查找成功进行③，失败进行④；

③如果查找到有的话，分配该空闲块并判断该块全部分配还是部分分配，如果全部分配，直接将该块从 free 链表中删除并插入到 used 链表中去；如果部分分配，需要更新该空闲块的大小并产生一个新的块（包括起始地址、大小、下一块地址、虚拟地址）将其插入到 used 链表中去；

④没有的话新分配一个页面，并判断是否需要合并，即上一页的最后是否与新分配的块处于连续的状态，如果是的话就合并并将该块分配给当前的需求使用，同时需要和③一样做部分分配和全部分配的判断；

⑤更新空闲 block 队列和 used 的 block 队列，对需要删除的 free 块直接删除，需要修改的块进行修改；对 used 队列进行插入，直接插入最前面即可。

### 3.2.5 Free 实现：

①从 used 的 block 队列中找该位置对应的块：对该链表进行遍历，找到位置处于保存的块信息中的范围的块；

②没有的话报错：当前请求释放的地址为非法地址，直接调用 panic 输出提示；

③有的话 free 掉，将其从 used 队列中删除：将该块信息从 used 的链表中删除；

④插入该块到 free 的 block 队列中，合并相邻的 block：从头开始遍历 free 的块，查找到该块应该插入的位置，对插入位置的前后进行判断，来查看前中，中后，前中后是否需要合并，如果需要合并直接合并，不需要的话直接插入。

### 3.3 实验调试及心得

在本次实验中，我遇到了如下问题：

1. 内核态访问的都是物理地址，构造的块指针一定要指向物理地址；
2. 不要在进程中对块初始化，因为在执行过程中会发生进程调度导致原本保存的块被损坏；
3. 起始地址保存好，块的结构信息也要保存在页中；
4. 申请的大小要对齐，与 8 对齐，否则会触发异常。

在起始的时候，我只保存了虚拟地址从而使得访问发生了越界，之后将虚拟地址修改为物理地址便成功运行了。

```
9 int main(void) {
10
11     char str[20] = "cross page";
12     char *m = (char *)better_malloc(100);
13     char *p = (char *)better_malloc(4096); // cross page
14     if((uint64)p - (uint64)m > 512 ){
15         printu("you need to manage the vm space precisely!");
16         exit(-1);
17     }
18     better_free((void *)m);
19
20     strcpy(p, str);
21     printu("%s\n", p);
22     exit(0);
23     return 0;
24 }
```

C:\Users\xuuuuu\AppData\Local\Temp\Mxt221\RemoteFiles\2\ UNIX C/C++ 24 lines Row #1

```
daydream@DESKTOP-NOQDMH7:~/riscv/riscv-pke$ 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): 0x000000000010078
Switch to user mode...
cross page
User exit with code:0.
System is shutting down with exit code 0.
```

图 3.1 实验测试的代码与输出

# 实验四 实现信号量

## 4.1 实验目的

通过控制信号量的增减，控制主进程和两个子进程的输出按主进程，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，使得输出达到预期。

需要做的工作：

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。

## 4.2 实验内容

### 4.2.1 构造信号等的结构：

信号灯需要有一个初始值和与其他信号灯能够区分开来的标志，同时对于同一个信号灯，有多个进程与其有关，因此信号的数据结构可以用如下结构表示：

```
typedef struct semaphore{
    int signal;
    process *waiting_queue;
}semaphore;
```

其中 `signal` 表示该信号灯的初始值，区分标志为数组下标，即定义一个信号灯数组保存信号灯的信息，`waiting_queue` 表示因为该信号灯而等待的进程。

### 4.2.2 实现三个函数：sem\_num,sem\_P,sem\_V:

(1) `sem_num`:

函数功能：参数为信号灯的初始值，返回信号灯的信号值，该信号值由操作系统为其分配并初始化。

函数实现：判断当前分配的信号灯是否已经达到上限，如果已经达到上限，调用 `panic` 输出相关信息；否则就初始化其初始值并将信号灯量加一。

(2) `sem_P`:

函数功能：参数为信号灯的信号值； 对该信号进行 P 操作，即对该信号的值-1，如果减完之后小于 0，将该进程置于该信号的等待队列中，并调度一个新

的进程；否则继续执行。

函数实现：首先判断信号灯的标志是否合法，即其是否满足数组下标的要求；如果不满足就调用 `panic` 中止；否则就找到该信号灯对应的结构，将其信号量-1，-1 后如果小于 0 表示当前资源不够，调用 `insert_to_waiting_queue` 函数将该进程插入到该信号灯的等待队列并调度一个新的进程执行，否则就继续执行。

### (3) `sem_V`:

函数功能：参数为信号灯的信号值； 对该信号进行 V 操作，即对该信号的值+1，如果加了之后大于 0，不管；否则，就调度该信号的等待队列中的一个进程将其加入就绪队列。

函数实现：先判断信号灯的标志是否合法，即其是否满足数组下标的要求；如果不满足就调用 `panic` 中止；否则就找到该信号灯对应的结构，将其信号量+1，+1 后如果信号量 $\leq 0$ ，表示当前有进程在等待该信号灯，因此需要从该等待队列中去掉队首的进程将其状态置为 `READY` 并调用函数将其插入就绪队列中去；否则就继续执行。

### (4) `insert_to_waiting_queue`:

函数功能：参数为信号灯的下标，将当前进程插入该信号等的等待队列中去。

函数实现：找到该队列的队尾，将该进程插入到队尾中去并就状态置为 `BLOCKED` 状态。

## 4.2.3 完善系统调用

系统调用实现上述三个函数调用的部分，即首先需要对每一个功能定义一个系统调用号，从而可以找到服务例程。在服务例程中，需要用到进程的相关信息，因此可以将详细实现放在 `process` 中，这里服务例程的实现直接调用 `process` 将封装好的服务程序。

在 `syscall` 中包含了 `process` 的头文件，因此需要使用上述三个函数的时候直接调用即可，因此需要在 `syscall` 中增加三个 `case` 并调用 2 中所实现的三个函数。

## 4.3 实验调试及心得

在这个实验中，需要实现的具体功能为：

1. 定义信号灯的数据结构：信号值和等待进程队列；
2. 具体实现三个函数，达到 1 中所想要的功能。

在分析其实现的过程中，需要从 `app` 程序->系统功能实现一步步的进行逻辑推理，确定每一步需要实现的功能；而在具体实现的过程中，需要从系统功能实现->`app` 程序进行一步步程序实现，也就是进程函数功能实现->系统调用完善->用户函数完善。



因此在这个实验过程中，需要根据“实现->封装->调用”的流程对所想要实现的功能进行实现。

```
daydream@DESKTOP-NOQDMH7:~/riscv/riscv-pke$ spike ./obj/riscv-pke ./obj/app_semaphore
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x0000000080009000, PKE kernel size: 0x0000000000009000.
free physical memory address: [0x0000000080009000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x0000000080005000
kernel page table is on
Switch to user mode...
in alloc_proc. user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user kstack 0x0000000087fbb000
User application is loading.
Application: ./obj/app_semaphore
CODE SEGMENT added at mapped info offset:3
Application program entry point (virtual address): 0x0000000000010078
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
going to insert process 0 to ready queue.
going to schedule process 0 to run.
Parent print 1
going to insert process 1 to ready queue.
going to schedule process 1 to run.
Child0 print 1
going to insert process 2 to ready queue.
going to schedule process 2 to run.
Child1 print 1
going to insert process 0 to ready queue.
going to schedule process 0 to run.
Parent print 2
going to insert process 1 to ready queue.
going to schedule process 1 to run.
Child0 print 2
going to insert process 2 to ready queue.
going to schedule process 2 to run.
Child1 print 2
```

图 4.1 运行输出部分截图