



华中科技大学

操作系统原理课程实验报告

姓 名： 徐雨梦
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： U202015562
指导教师： 周正勇

分数	
教师签名	

2022 年 12 月 31 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	6
实验二 复杂缺页异常	6
2.1 实验目的	8
2.2 实验内容	8
2.3 实验调试及心得	9
实验三 进程等待和数据段复制	9
3.1 实验目的	11
3.2 实验内容	11
3.3 实验调试及心得	16

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`。

因此该函数的实现为：

```
void print_backtrace(int num){
    do_user_call(SYS_user_backtrace,num,0,0,0,0,0);
}
```

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，代码如下：

```
ssize_t sys_user_backtrace(int num) {
    if(num>=0)
        return backtrace(num);
    else
        panic("backtrace_depth is illegal\n");
}
```

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` \rightarrow `f1` \rightarrow `f2` \rightarrow `f3` \rightarrow `f4` \rightarrow `f5` \rightarrow `f6` \rightarrow `f7` \rightarrow `f8` \rightarrow `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` \rightarrow `f1` \rightarrow `f2` \rightarrow `f3` \rightarrow `f4` \rightarrow `f5` \rightarrow `f6` \rightarrow `f7` \rightarrow `f8` 中，分别调用了 `f1` \rightarrow `f2` \rightarrow `f3` \rightarrow `f4` \rightarrow `f5` \rightarrow `f6` \rightarrow `f7` \rightarrow `f8` \rightarrow `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：

```
#define SHT_SYMTAB 2
#define SHT_STRTAB 3
#define STT_FUNC 2
#define MAX_DEPTH 20
#define STRTAB_MAX 400
#define ELF64_ST_TYPE(info) ((info) & 0xf)
```

(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。

代码为：

```
long backtrace(int depth){
    int i,off;
```

```

//string table head address
//claim string table header
elf_sect_header strtab;
//string table head address
//get string table header's info
for(i=0,off=elfloader.ehdr.shoff;i<elfloader.ehdr.shnum;i++,off+=sizeof(strtab))
{
    if(elf_fpread(&elfloader,(void*)&strtab, sizeof(elf_sect_header), off) != sizeof
(elf_sect_header)) panic("string table header get failed!\n");
    if(strtab.type == SHT_STRTAB) break;
}
//save string table
char strtab_info[STRTAB_MAX];
if (elf_fpread(&elfloader,(void*)strtab_info, sizeof(strtab_info), strtab.offset) !=
sizeof(strtab_info)) panic("string table get failed!\n");
//symbol table header
elf_sect_header symtab;
//look up for symbol table header's info
for(i=0,off=elfloader.ehdr.shoff;i<elfloader.ehdr.shnum;i++,off+=sizeof(symtab))
{
    if(elf_fpread(&elfloader,(void*)&symtab, sizeof(elf_sect_header), off) != size
of(elf_sect_header)) panic("symbol table header get failed!\n");
    if(symtab.type == SHT_SYMTAB) break;
}
//save symbol table's info
int sym_num=0;
elf_sym symbols[MAX_DEPTH];
elf_sym temp;
off = symtab.offset;
for(i=0;i<symtab.size/symtab.entsize;i++)
{
    if(elf_fpread(&elfloader,(void*)&temp, sizeof(temp), off) != sizeof(temp)) pa
nic("symbol table get failed!\n");
    if ((ELF64_ST_TYPE(temp.st_info)) == STT_FUNC){
        symbols[sym_num] = temp;
        sym_num++;
        //sprintf("symbol:%s\n",temp.st_name+strtab_info);
    }
    off += sizeof(temp);
}
//f8 address
uint64 *cur_s0 = ((uint64*)current->trapframe->regs.s0+1);
uint64 place = *cur_s0;
//backtrace

```

```

for(i=0;i<depth;i++){
    for(int j=0;j<sym_num;j++){
        if(symbols[j].st_value <= place && symbols[j].st_value+symbols[j].st_size>place){
            off = symbols[j].st_name;
            sprintf("%s\n",off+strtab_info,off);
            if(strcmp(strtab_info+off,"main")==0) return i+1;
            place = symbols[j].st_value;
            place+=symbols[j].st_size;
            break;
        }
    }
}
return i;
}

```

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。

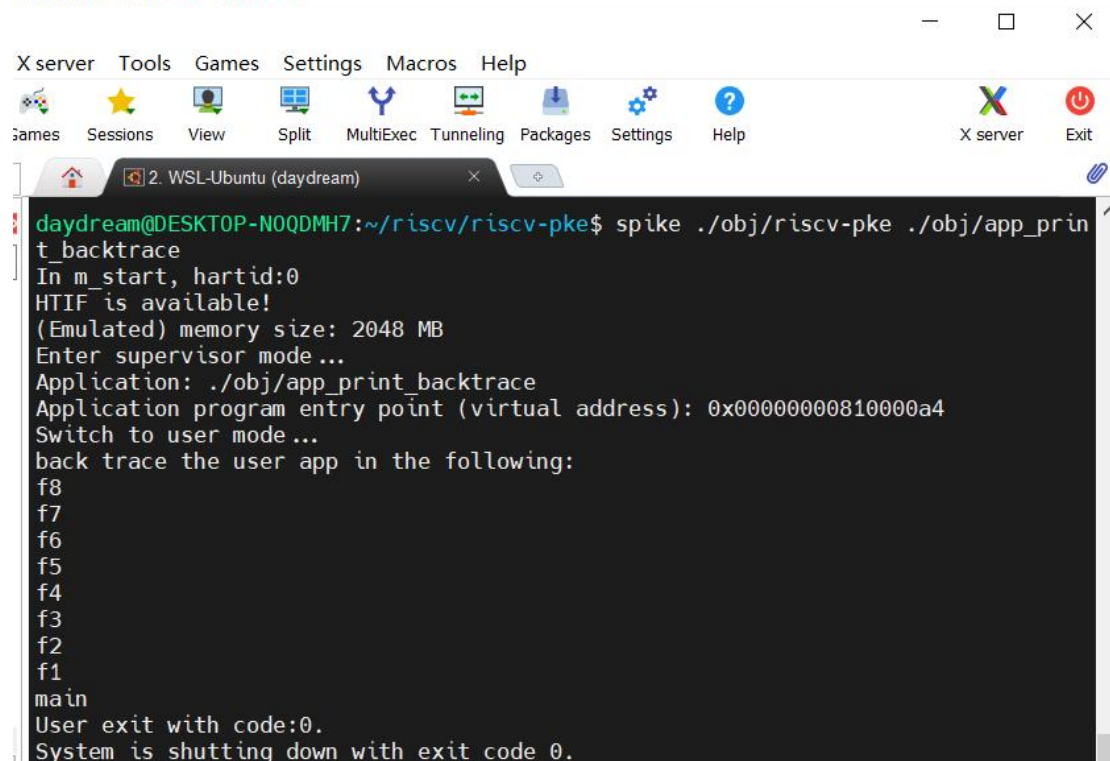
第二个是用户栈的获取，在该程序中，每个函数在调用的时候都使用了栈的两个 uint64 的位置，因此在遍历的过程中每次对指针加 2 即可，但是这个方法只适用于使用栈两个字节的情况，不具有普适性；另外一点就是返回地址获取后的判断，在一开始我使用的是反汇编找到函数开头和返回地址的关系进行判断的，但是该情况只适合函数参数满足一定条件的情况，也不具有普适性。在知道了这些问题之后，我对实现方案进行了调整。

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 复杂缺页异常

2.1 实验目的

本实验要求对合法的缺页异常进行重新分配的处理，但是对于不合理的缺页异常（在本实验中为非法地址访问）需要进行异常处理。

因此需要完成以下工作：

1. 扩展异常处理功能，可以判断当前异常类型并进行相应的正确处理；
2. 实现虚拟地址空间监控，掌握当前进程使用了多少空间。

2.2 实验内容

2.2.1 虚拟地址监控

对于一个程序，会从虚拟空间地址 `USER_STACK_TOP` 开始进行空间分配，每次分配一页，因此对虚拟地址进行监控，也就是掌握该进程所占的虚拟空间地址范围，可以转换成对分配页数的监控，`USER_STACK_TOP` 至 `USER_STACK_TOP+PGS_SIZE*页数` 即为所分配的虚拟地址空间。

我的解决办法就是在 `process` 结构体中加一个变量 `malloc_page_num`，记录分配的页数，该变量在 `switch_to` 函数中会初始化为 1，在缺页异常处理函数中会在分配空间后进行自增操作。

2.2.2 异常处理

对于合法的缺页异常，将对其分配一个新页并将虚拟地址与该空间进行映射，而不合法的缺页异常，应该调用 `panic` 对用户进行提醒。

是否合法的判断条件便是虚拟地址的访问范围，在该程序中，递归函数随着递归深度的增加会需要更多的空间来保存调用过程，该爆栈行为是合理的，属于合法的缺页异常，其缺页的虚拟地址范围也将是当前栈栈顶；而对于数组爆栈，属于非法地址访问，因此需要阻止，而数组的分配是在堆上的，其数据地址和代码地址并不在一起。因此可以通过范围进行判断，在函数压栈的过程中，有一个 `sp` 寄存器保存了栈顶指针，在一页压满的时候，`sp` 指针所指向的地址会超过该页指向想压栈的下一页，而代码段需要的虚拟地址会比该指针所指的地址高并且比栈底地址低，因此可以以此作为条件判断当前的虚拟地址是否合法。

```
if(stval>=current->trapframe->regs.sp&&stval<=USER_STACK_TOP)
```

```

    {
        void* pa = alloc_page();
        user_vm_map((pagetable_t)current->pagetable, ROUNDDOWN(stval, PGSIZE),
        PGSIZE, (uint64)pa, prot_to_type(PROT_WRITE | PROT_READ, 1));
        current->malloc_page_num++;
    }
    else { //illegal
        panic("this address is not available!");
    }
}

```

2.3 实验调试及心得

在本实验，最重要的就是找到合法异常的虚拟地址范围，由于数组是分配在堆上的，其地址比代码段地址要低许多，因此采用了范围比较的方式解决。

在调试过程中，遇到了一个异常问题，在虚拟地址和实际物理地址之间进行映射的时候，发生了错误，引发了异常处理。在这里我才用了 `user_vm_map` 函数实现了映射，其中需要提供两个参数，分别是虚拟地址起始地址和分配的大小，我最初仿照着其他地方的映射函数对其提供参数，也就是 `stval` 和 `PGSIZE` 两个参数，然而发生了错误，在观察了 `user_vm_map` 函数以及其调用的 `map_pages` 函数后，我发现，在分配空间的时候，首先会调用 `ROUNDDOWN` 对虚拟地址进行 `PGSIZE` 大小的对齐，接下来再通过原虚拟地址计算要分配的最大范围，在输出调试后我发现，该虚拟地址并非是 `PGSIZE` 大小对齐的，如果提供 `PGSIZE` 大小的页面，会使得 `map_pages` 分配两个页面而导致错误，因此需要将调整虚拟地址和大小这两个参数，有两种解决方案。

方案一为提供一个合适的分配大小，使得虚拟地址加上该大小仍然与该虚拟地址处于同一个界面，这样就不会导致多分配而引发异常。方案二为将虚拟地址对齐，在 `map_pages` 中也会对起始虚拟地址对其，因此在这里我们可以直接先对齐并传递一个 `PGSIZE` 大小的分配大小参数。

在调试的过程中，我对虚拟地址与实际物理地址之间的映射有了更深的了解，并且深入了解学习了分配界面的函数在底层是如何实现的。还学习了栈和堆的相关知识以及其在内存中的保存方式和范围。对缺页异常处理也有了更深入的掌握，学习了多种缺页异常及其正确的处理方式。

```
app_sum_sequence.c
21 }
22
23 int main(void) {
24     // FIRST, we need a large enough "n" to trigger pagefaults in the user stack
25     uint64 n = 1024;
26 }

tu (daydream)
sions View Xserver Tools Games Settings Macros Help
Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help X server Exit
t...
2. WSL-Ubuntu (daydream)
daydream@DESKTOP-NOQDMH7:~/riscv/riscv-pke$ spike ./obj/riscv-pke ./obj/app_sum_
sequence
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x0000000080007000, PKE ker
nel size: 0x0000000000007000 .
free physical memory address: [0x0000000080007000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN BASE 0x0000000080000000
physical address of _etext is: 0x0000000080004000
kernel page table is on
User application is loading.
user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user kstack 0x0000
000087fbb000
Application: ./obj/app_sum_sequence
Application program entry point (virtual address): 0x00000000000100a2
Switch to user mode...
handle_page_fault: 000000007ffffdf8
handle_page_fault: 000000007ffffcf8
handle_page_fault: 000000007ffffbf8
handle_page_fault: 000000007ffffaf8
handle_page_fault: 000000007ffff9f8
handle_page_fault: 000000007ffff8f8
handle_page_fault: 000000007ffff7f8
handle_page_fault: 000000007ffff6f8
handle_page_fault: 0000000000401000
this address is not available!
System is shutting down with exit code -1.
```

图 2.1 递归参数和运行结果（溢出的情况）

```
app_sum_sequence.c
21 }
22
23 int main(void) {
24     // FIRST, we need a large enough "n" to trigger pagefaults in the user stack
25     uint64 n = 512;
26 }

tu (daydream)
sions View Xserver Tools Games Settings Macros Help
Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help X server Exit
t...
2. WSL-Ubuntu (daydream)
PKE core has been built into "obj/riscv-pke"
compiling user/app_sum_sequence.c
linking obj/app_sum_sequence ...
User app has been built into "obj/app_sum_sequence"
daydream@DESKTOP-NOQDMH7:~/riscv/riscv-pke$ spike ./obj/riscv-pke ./obj/app_sum_
sequence
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x0000000080007000, PKE ker
nel size: 0x0000000000007000 .
free physical memory address: [0x0000000080007000, 0x0000000087ffffff]
kernel memory manager is initializing ...
KERN BASE 0x0000000080000000
physical address of _etext is: 0x0000000080004000
kernel page table is on
User application is loading.
user frame 0x0000000087fbc000, user stack 0x000000007ffff000, user kstack 0x0000
000087fbb000
Application: ./obj/app_sum_sequence
Application program entry point (virtual address): 0x00000000000100a2
Switch to user mode...
handle_page_fault: 000000007ffffdf8
handle_page_fault: 000000007ffffcf8
handle_page_fault: 000000007ffffbf8
handle_page_fault: 000000007ffffaf8
Summation of an arithmetic sequence from 0 to 512 is: 131841
User exit with code:0.
System is shutting down with exit code 0.
```

图 2.2 递归参数和运行结果（不溢出的情况）

实验三 进程等待和数据段复制

3.1 实验目的

实现 wait 函数，使其根据传递的参数实现相应的功能，比如进程同步等。该函数需要满足：①当 pid 为-1 时，父进程等待任意一个子进程退出即返回子进程的 pid；②当 pid 大于 0 时，父进程等待进程号为 pid 的子进程退出即返回子进程的 pid；③如果 pid 不合法或 pid 大于 0 且 pid 对应的进程不是当前进程的子进程，返回-1。

因此需要做以下工作：

1. 修改 PKE 内核和系统调用，为用户程序提供 wait 函数的功能；
2. 实现 wait 函数的功能；
3. 补充 do_fork 函数，实现数据段的复制并保证 fork 后父子进程的数据段相互独立；
4. 进程执行、退出函数的完善。

3.2 实验内容

3.2.1 系统调用

进程等待需要进入操作系统内核 S 态进行，因此需要调用 do_user_syscall 函数进入操作系统内核。

类似于挑战实验一，wait 函数提供一个参数，因此该参数也应该传递给 do_user_syscall 函数的一个参数中，这里我传递给了 a1，而 do_user_syscall 函数本身需要一个参数 a0 来表示系统功能，在这里需要定义一个新的变量 SYS_user_wait 表示需要系统进行等待，这个变量应该在 syscall.h 中定义为

```
#define SYS_user_wait (SYS_user_base + 6)
```

因此 wait 在用户使用中的函数定义为：

```
void wait(uint64 pid){  
    do_user_call(SYS_user_wait,pid,0,0,0,0,0);  
}
```

接下来便进入 do_syscall 函数，根据 a0 选择当前的系统调用，在这里将该系统调用的操作封装到 sys_user_wait 中，sys_user_wait 函数实现要求的功能，将在 3.2.3 与 3.2.4 详细说明。

3.2.2 数据段复制

在 `do_fork` 函数中，会为父进程生成一个子进程，该子进程与父进程使用相同的代码段，但是使用不同的数据段，在前面的实验中，以及实现了代码段的映射，这里需要实现数据段的复制。

由于父进程和子进程的数据段相互独立，这里需要分配新的页面给子进程的数据段，并将父进程的数据都复制给该新分配的位置，子进程的虚拟地址与父进程相同，但是物理地址不同，需要建立映射。在建立映射的时候，由于数据段可以被代码修改和使用，因此在权限这里，需要对其赋予可读可写可执行的权限。

而子进程还有一些其他数据需要赋值或更新，比如页面数量、虚拟地址，在这里也需要赋值。

因此代码为：

```
case DATA_SEGMENT:{
    //copy parent's DATA_SEGMENT to child process's DATA_SEGMENT
    for(int j=0; j<parent->mapped_info[i].npages; j++)
    {
        //look up for parent process's data segment
        uint64 parent_pa = lookup_pa(parent->pagetable,parent->mapped_info[i].va+j
*PGSIZE);
        if(parent_pa){
            void* child_pa=alloc_page();
            if(child_pa){
                memcpy((void*)child_pa,(void*)parent_pa,PGSIZE);
                user_vm_map(child->pagetable, parent->mapped_info[i].va+j*P
GSIZE,PGSIZE, (uint64)child_pa,prot_to_type(PROT_EXEC | PROT_READ |PR
OT_WRITE, 1));
            }
        }
    }
    child->mapped_info[child->total_mapped_region].va=parent->mapped_info[i].va;
    child->mapped_info[child->total_mapped_region].npages=parent->mapped_info
[i].npages;
    child->mapped_info[child->total_mapped_region].seg_type=DATA_SEGMENT;
    child->total_mapped_region++;
    break;
}
```

3.2.3 wait 功能函数的实现

如果 pid 的值合法并且父进程需要等待子进程使其同步的话，需要让父进程进入阻塞状态并保存阻塞原因，因此我在这里修改了父进程的数据结构，增加了一个变量 block_id，表示该进程阻塞是由另外的某个进程引起的。

对于 wait 函数来说，父进程需要等待某个子进程，如果该子进程结束父进程才会继续执行，而子进程是否结束的标志就是子进程的状态。进程结束的函数时 exit，观察 exit 函数发现，进程结束即将进程状态标记为僵尸状态，等待后续再释放。因此在这里，判断一个进程是否结束的条件就是判断该进程的状态是否为僵尸状态。

因此 wait 函数的算法步骤为：

①判断参数 pid 的值，pid 为-1 时执行②；

②pid 的值为-1 时，首先设置一个标志参数 child 表示该进程是否拥有子进程，并初始化为-1；

③接下来遍历所有的进程查看是否有进程的父进程为当前进程，如果有，该进程即为当前进程的子进程，更新 child 值为该进程的 pid 值；进而判断该进程的状态是否为僵尸状态，如果是，将该进程释放设置为 FREE 状态并至阻塞原因为无（0），并返回该进程的 pid 值；

④遍历结束，如果 child 仍为-1，表示该进程无子进程，返回标志-1（表示不合法）；如果 child 为其他值，表明该进程有子进程但无进程结束，因此需要将该进程置为阻塞（BLOCKED）状态并设置阻塞原因为该子进程，设置方式为 block_id |= 1<<procs[child].pid，设置完毕后返回标志-2（表示阻塞）；

⑤pid 不为-1 并且为合法值时，找到该 pid 值对应的进程，首先判断该进程的父进程是否为当前进程，如果不是，表示不合法，返回标志-1；

⑥如果是，继而判断该进程的状态是否为僵尸状态，如果是，将该进程释放设置为 FREE 状态并设置当前进程的阻塞原因为无（0），返回该进程的进程号；如果该进程状态并非僵尸状态，当前进程进入阻塞状态并根据该进程的 pid 值设置阻塞原因，返回阻塞标志-2；

⑦如果 pid 值既不是-1，也不合法。将返回不合法标志-1。

函数 wait 实现了文档的要求，具体代码为：

```
uint64 wait(uint64 pid)
{
    if(pid== -1)//look up for child process to return
    {
        int child=-1;
        for(int j=0;j<NPROC;j++)
```

```

{
    if(procs[j].parent == current)//child process
    {
        child=j;
        if(procs[j].status == ZOMBIE)//child process is zombie
        {
            procs[j].status = FREE;
            current->block_id = 0;
            return procs[j].pid;
        }
    }
}

if(child==-1) return -1;//no child process
else //child process all running, parent process turns into blocked
{
    current->block_id |= 1<<procs[child].pid;
    current->status = BLOCKED;
    return -2;
}

}

else if(pid>0&&pid<NPROC){//pid is legal
    if(procs[pid].parent == current)
    {
        if(procs[pid].status == ZOMBIE)
        {
            procs[pid].status = FREE;
            current->block_id = 0;
            return procs[pid].pid;
        }
    }
    else{
        current->block_id |= 1<<procs[pid].pid;
        current->status = BLOCKED;
        return -2;
    }
}
}

```



```

        else return -1;//this pid is not child process
    }
    else return -1;//pid is illegal
}

```

3.2.4 进程相关函数的完善

1. sys_user_wait 函数的完善

函数 wait 实现了文档的要求，然而并未实现进程调用等调用上的操作，该操作在 sys_user_wait 函数中实现。

在进入 sys_user_wait 函数后，调用 wait 函数来获取当前的进程应该设置的状态，该状态可以通过 wait 函数的返回值得到，也可以通过 current 也就是当前进程的状态进行判断。这里我用返回值判断，如果返回值为-2，表示父进程进入了阻塞状态，这时便需要通过 schedule 函数执行下一个进程了；否则返回值为其他值，表明该进程应该继续执行，通过调用 sys_user_yield 函数开始执行进程。

因此代码为：

```

while(1){
    uint64 x=wait(pid);
    if(x==-2)//parent process is blocked
    {
        schedule();
        return x;
    }
    else{//not blocked
        sys_user_yield();
        return x;
    }
}

```

2. sys_user_exit 函数的完善

在父进程进入阻塞状态需要等待子进程的时候，会直接执行下一个进程并且不会将父进程加入到队列中，直到子进程执行完才会将父进程加入到执行等待队列中去，而在子进程执行完后并不会返回 wait 函数来继续执行父进程，因此需要子进程在退出的时候将父进程加入到等待队列中去，该操作实现在 sys_user_yield 函数中。

在 free 了当前的进程之后，会判断当前进程是否拥有父进程且该父进程是否处于阻塞状态并且阻塞原因为当前进程，如果都是的话，将对该父进程重新至

于准备状态（READY）并将其阻塞原因清零，将其加入到等待队列中去等待进程执行。

因此代码为：

```
free_process( current );
process* par=current->parent;
if(par)
{
    if(par->status==BLOCKED&&par->block_id==1<<current->pid)
    {
        par->status = READY;
        par->block_id = 0;
        insert_to_ready_queue(par);
    }
}
schedule();
```

3.3 实验调试及心得

该实验总共需要做四个工作，其中系统调用在挑战实验一中就已经做过了，所以这里实现起来比较简单。

数据段复制也相对来讲比较简单，最重要的就是搞清楚各个地址应该是什么，即子进程的虚拟地址与父进程相同但是实际物理地址不同需要重新分配并映射，以及该数据的权限。在一开始出现了引发异常的情况，在调试之后发现，是由于权限只给了读写，并未给予可执行权限引发的。

函数 wait 的实现需要编写较多的代码，但是也并不算太难，因为已经将 wait 函数的功能在文档中分类讨论详细说明了，因此只需要通过 if 判断再实现相应的功能即可。一开始的时候我并没有详细的了解进程结束的标志，想当然的认为进程结束的标志是 FREE，然而在执行了以后发生了错误。因此我重新从进程调用以及进程结束开始向前推，发现进程结束是通过 exit 实现的，并且具体实现为简单的将进程设置成僵尸状态，在修改了后结果运行正确。此外就是阻塞原因的设置，这里搜寻了网上的一些资料，采用了左移赋值的方式进行阻塞原因保存。

较为难的是相关函数的补充，这一部分在文档中并没有给出来，所以很容易考虑不到，一开始我一直困扰于如何将父进程重新加入队列，最开始的方案考虑的是在 wait 之后直接 schedule 一个进程后将该父进程直接加入进去，然而调式

了之后发现这样在生成一个新的子进程之后，并不能等待子进程结束再调用父进程而是提前执行了父进程，后来思索如何在 wait 函数中实现该功能，然而并没有想出来实现方案。后来上网查询了资料并且和同学进行了交流，发现可以不拘泥于 wait 函数执行所有功能，而是将这些功能实现在调用 wait 的系统函数中和进程结束的系统调用函数中，这样的话，就能在进程结束的时候再将父进程加入到队列中去。

在这次的实验中，我更加深入的学习了进程调用方面的知识，并且和课程中所学的知识结合起来，解答了一些我在理论学习上的问题。并且更加了解了进程等待的实现方式和进程退出的实现方式以及进程等待队列的组成。对操作系统内核更加了解了。在整个实验过程中，我也要感谢解答我问题的老师和同学，大家互相交流学习，达到了更高的学习效率！

```
Application: ./obj/app_wait
CODE_SEGMENT added at mapped info offset:3
DATA_SEGMENT added at mapped info offset:4
Application program entry point (virtual address): 0x0000000000100b0
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 0x0000000087fae000, user stack 0x000000007ffff000, use
r kstack 0x0000000087fad000
going to insert process 1 to ready queue.
pid=1
going to schedule process 1 to run.
pid=0
User call fork.
will fork a child from parent 1.
in alloc_proc. user frame 0x0000000087fa1000, user stack 0x000000007ffff000, use
r kstack 0x0000000087fa0000
going to insert process 2 to ready queue.
going to schedule process 2 to run.
Grandchild process end, flag = 2.
User exit with code:0.
going to insert process 1 to ready queue.
going to schedule process 1 to run.
Child process end, flag = 1.
User exit with code:0.
going to insert process 0 to ready queue.
going to schedule process 0 to run.
Parent process end, flag = 0.
User exit with code:0.
no more ready processes, system shutdown now.
System is shutting down with exit code 0.
```

图 3.1 进程运行结果