华中科技大学

操作系统原理课程实验报告

姓　　名：

学　　院：　计算机科学与技术学院

专　　业：　数据科学与大数据技术

班　　级：

学　　号：

指导教师：　周正勇

分数	
教师签名	

2024 年 1 月 7 日

目　录

实验一 打印用户程序调用栈

1.1 实验目的

本实验的目的是通过修改 PKE 内核，实现从给定应用（user/app_print_backtrace.c）到预期输出的转换。具体而言，实验要求实现一个名为 print_backtrace()的函数，该函数能够根据控制输入的参数来控制回溯的层数。通过调用 print_backtrace()函数并传入不同的参数，可以控制回溯的深度，从而实现输出指定层数的回溯信息。

实验要求：

1. 修改 PKE 内核：首先，需要对 PKE 内核进行修改，以支持 print_backtrace()函数的实现。这可能涉及修改内核的调用堆栈跟踪机制或函数调用约定。
2. 实现 print_backtrace()函数：根据实验要求，在修改后的 PKE 内核中实现 print_backtrace()函数。该函数应接受一个整数参数，表示回溯的层数。根据参数的值，print_backtrace()函数应输出相应层数的回溯信息。
3. 控制回溯的层数：根据实验要求，应用程序调用 print_backtrace()时，应能够通过控制输入的参数来控制回溯的层数。例如，如果调用 print_backtrace(5)，则 print_backtrace()函数应输出 5 层的回溯信息；如果调用 print_backtrace(100)，则应回溯到 main 函数并停止回溯，因为调用的深度小于 100。

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,0,0,0,0,0,0);
}
```

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 sys call 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 ->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

backtrace 函数是一个用于获取当前函数调用链的函数，它可以在程序运行时打印出函数的调用关系，帮助程序员进行调试和错误排查。下面是 backtrace 函数的大致实现过程：

1. 初始化：在函数开始时，需要将进程初始化时读取的 elf 文件头变量全局化，以便 backtrace 函数可以使用。
2. 获取字符串表表头信息：从 elf 文件的表头信息中找到字符串表表头（SHT_STRTAB）的表头，并将其保存在 elf_sect_header 的 strtab 表头中。这可以通过遍历 elf 文件的所有表头，并根据表头类型来判断。
3. 获取字符串表：根据字符串表表头的偏移地址，通过读取 elf 文件的内容，获取字符串表的内容。
4. 获取符号表表头信息：与获取字符串表表头信息的步骤类似，但是需要查询类型为 SHT_SYMTAB 的表头。
5. 获取符号表：根据符号表表头的偏移地址和大小，读取所有的符号表表项，并保存下来。在保存符号表表项之前，可以通过判断函数类型来决定是否保存。通常情况下，backtrace 函数只关注可执行函数的符号表，如 main 函数及其调用链上的函数。
6. 获取返回第一个返回地址：通过将 print_backtrace 的 fp 指针向高地址移动 8 个字节，可以指向 print_backtrace 的返回地址 ra。
7. 循环实现回溯：利用存储的符号表信息，从当前返回地址开始向前回溯调用

链。通过遍历符号表表项，找到包含返回地址所在代码段的符号表表项，然后根据该表项中的 name 偏移地址，在字符串表中找到对应的函数名，并打印出来。

8. 判断回溯是否结束：根据设定的深度和当前查询到的函数名是否为"main"，判断是否继续回溯。当达到设定的深度或者查询到"main"函数时，回溯结束。

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++;
        //sprint("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;
          sprint("%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

在这个实验中，我遇到了一些挑战，尤其是在实现最终的 backtrace 功能时。以下是我在调试过程中遇到的问题以及我采取的解决方案：

1. 符号表的获取：一开始，我只知道 elf 文件提供了字符串表的偏移信息，但没有提供符号表的信息。通过与同学交流并查阅资料，我了解到符号表表头的类型值为 2，而字符串表表头的类型值为 3。因此，我通过解析 elf 文件的表头，根据类型值来确定符号表的位置，从而解决了符号表的获取问题。
2. 用户栈的获取：在这个程序中，每个函数调用时都使用了栈的两个 uint64 位置。因此，在遍历过程中，我每次将指针加 2 来获取下一个栈位置。然而，这种方法只适用于使用两个字节的栈情况，不具有普适性。另外，我遇到了返回地址获取后的判断问题。起初，我使用反汇编来找到函数开头和返回地址之间的关系进行判断，但这种方法只适用于函数参数满

足特定条件的情况，不具有普适性。在意识到这些问题后，我调整了实现方案。

3. 函数地址和代码段长度：在符号表中，保存了函数的地址和代码段使用的字节数。另外，在调用过程中，函数是按照高地址向低地址分配代码段空间的。因此，我修改了函数查找的条件，将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，无论参数如何变化，都可以准确地找到函数。对于返回地址，我首先找到了 f8 的符号表，并保存了其函数地址。然后，通过将该地址与代码段长度相加，找到下一个函数的代码起始地址。这样，即使参数压栈导致栈指针无法通过简单的加 2 获取下一个返回地址，也能正确地获取返回地址。

在编写这个实验的过程中，我遇到了一些其他问题，比如对 elf 头文件的理解不够清晰，阅读教程后仍然感到困惑，特别是符号表和字符串表以及它们的表头信息。我需要多次查询相关资料来弄清楚这些概念，有点类似于中断向量表的原理。然而，在这个过程中，我和同学朋友一起交流，分享彼此的困惑并共同学习。最终，我们成功解决了这些问题。

总的来说，通过这个实验，我学到了很多关于符号表、字符串表以及 ELF 文件的知识。我也体会到了团队合作和与他人交流的重要性，这有助于我们共同解决问题并加深对知识的理解。此外，我也提高了调试技巧和解决问题的能力。

实验二 复杂缺页异常

2.1 实验目的

本实验的目的是通过修改 PKE 内核的代码，实现对不同情况的缺页异常进行不同的处理。具体要求如下：

1. 修改 PKE 内核：需要对 PKE 内核进行修改，包括 machine 文件夹下的代码文件。这些文件包含了处理缺页异常的相关代码。
2. 处理不同情况的缺页异常：根据实验要求，对于不同情况的缺页异常，需要采取不同的处理方式。具体情况可能包括用户源程序发生的错误和调用的标准库内发生的错误。
3. 文件名规范：在进行代码修改时，需要遵循文件名的规范。如果是用户源程序发生的错误，文件名应包含相对路径；如果是调用的标准库内发生的错误，文件名应包含绝对路径。

通过实现上述要求，可以熟悉并理解操作系统内核的异常处理机制，以及对不同情况的异常进行特定处理的能力。此外，通过对 PKE 内核的修改，还可以加深对操作系统内核的理解，并提升对底层代码的编程能力。

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-PGSIZE*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

在异常处理的上下文中，我们将重点关注缺页异常（Page Fault）以及如何判断缺页异常的合法性。

1. 缺页异常的处理：

当程序在执行过程中访问了一个尚未分配的内存页时，会触发缺页异常。对于合法的缺页异常，操作系统需要为该地址分配一个新的物理页，并将虚拟地址与新分配的物理页进行映射，以满足程序的内存访问需求。对于非法的缺页异常，表示程序访问了不允许访问的内存地址，需要采取相应的异常处理措施。

2. 判断缺页异常的合法性：

   在程序中，有两种情况需要进行判断：递归爆栈和数组爆栈。

   - 递归爆栈：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。这种情况下，爆栈行为是合理的，属于合法的缺页异常。缺页的虚拟地址范围将是当前栈的栈顶。由于递归过程中栈的增长是合法且可预测的，系统可以根据栈的大小和增长规律进行动态分配和映射，以满足程序的需求。
   - 数组爆栈：数组的分配是在堆上进行的，其数据地址和代码地址并不在一起。因此，当数组爆栈时，属于非法地址访问，需要阻止并进行异常处理。在处理过程中，可以通过对比虚拟地址的范围来判断地址的合法性。在函数压栈过程中，有一个 SP（Stack Pointer）寄存器保存了栈顶指针。当一页的空间被压满时，SP 指针所指向的地址会超过该页，指向下一页。而代码段需要的虚拟地址会比 SP 指针所指的地址高但比栈底地址低。因此，可以以此作为判断条件，检查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

在本实验中，我遇到了一些问题，并通过调试解决了它们。以下是我在调试过程中遇到的问题以及解决方案：

1. 异常处理问题：在虚拟地址和实际物理地址之间进行映射时，遇到了异常处理问题。通过观察代码和调试输出，我发现在调用 user_vm_map 函数时，提供的虚拟地址和大小参数没有正确对齐，导致分配了多个页面，进而引发异常。为解决这个问题，我采取了两种方案：一是提供合适的分配大小，使得虚拟地址加上该大小仍然处于同一个界面；二是在调用 user_vm_map 之前，对虚拟地址进行对齐，然后提供一个 PGSIZE 大小的分配大小参数。
2. 映射函数的理解：在调试过程中，我深入学习了虚拟地址与实际物理地址之间的映射过程。通过观察 user_vm_map 函数及其调用的 map_pages 函数，我了解到在分配空间时会对虚拟地址进行对齐，并计算出需要分配的最大范围。这个过程帮助我更好地理解了底层的分配界面函数的实现原理。
3. 栈和堆的知识学习：在调试过程中，我深入学习了栈和堆的相关知识，包括它们在内存中的保存方式和范围。这有助于我更好地理解程序中栈和堆的使用，以及在异常处理过程中对它们的影响。

通过这个实验，我不仅解决了代码中的问题，还学到了很多关于异常处理、虚拟地址映射和栈堆的知识。这加深了我对操作系统内核的理解，并提升了我对底层代码的编程能力。此外，通过调试过程，我也锻炼了解决问题和调试代码的能力，这对我今后的编程工作将非常有帮助。

实验三 进程等待和数据段复制

3.1 实验目的

本实验的目的是通过修改 PKE 内核和系统调用，为用户程序提供 wait 函数的功能，并在实现 fork 函数时保证父子进程的数据段相互独立。具体要求如下：

1. 修改 PKE 内核和系统调用：需要对 PKE 内核和系统调用进行修改，以提供 wait 函数的功能。这涉及到对内核代码和系统调用接口的修改。
2. 实现 wait 函数：wait 函数的功能要根据传递的参数实现相应的功能。当 pid 为-1 时，父进程需要等待任意一个子进程退出，并返回该子进程的 pid。当 pid 大于 0 时，父进程需要等待进程号为 pid 的子进程退出，并返回该子进程的 pid。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，则返回-1。
3. 补充 do_fork 函数：在实现 fork 函数时，需要补充 do_fork 函数，以实现数据段的复制并保证父子进程的数据段相互独立。具体的实现需要根据实验 3_1 中已经实现的代码段复制进行扩展。
4. 进程执行、退出函数的完善：在实现 wait 函数和 do_fork 函数的过程中，可能需要对进程的执行和退出函数进行一些修改和完善，以确保进程的正常执行和退出。

通过实现上述要求，本实验旨在让我们熟悉并理解操作系统进程管理的相关概念和原理，以及对内核代码和系统调用的修改和扩展能力。同时，通过实现 wait 函数和补充 do_fork 函数，还可以加深对进程同步和数据段复制的理解，并提升对底层代码的编程能力。

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,0,0,0,0,0,0);
}
```

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

wait 函数是一个用于等待子进程结束并获取其退出状态的函数。它的功能是父进程等待子进程结束，只有当子进程结束后，父进程才会继续执行。下面是 wait 函数的详细实现过程：

1. 判断参数 pid 的值：如果 pid 为-1，执行步骤②，否则继续执行。
2. 初始化标志参数 child 为-1，表示当前进程是否拥有子进程。
3. 遍历所有进程：遍历系统中的所有进程，查看它们的父进程是否为当前进程。如果找到一个子进程，更新 child 的值为该进程的 pid，并继续执行下一步。
4. 判断子进程的状态：判断找到的子进程的状态是否为僵尸状态。如果是僵尸状态，将该子进程释放（设置为 FREE 状态），将阻塞原因设置为无（0），并返回子进程的 pid 值。
5. 遍历结束后的处理：如果 child 仍为-1，表示当前进程没有子进程，返回标志-1（表示不合法）。如果 child 不为-1，表示当前进程有子进程但没有子进程结束，将当前进程设置为阻塞（BLOCKED）状态，并设置阻塞原因为该子进程，阻塞原因的设置方式为 block_id |= 1 << procs[child].pid。设置完毕后，返回标志-2（表示阻塞）。
6. pid 不为-1 且为合法值的处理：找到 pid 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 pid 不合法，返回标志-1。
7. 如果父进程是当前进程，继续判断该进程的状态是否为僵尸状态。如果是僵尸状态，将该进程释放（设置为 FREE 状态），将当前进程的阻塞原因设置为无（0），返回该进程的 pid。
8. 如果该进程的状态不是僵尸状态，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。
9. 如果 pid 既不是-1，也不是合法值，返回不合法标志-1。

wait 函数的实现过程主要是根据 pid 的值和进程的状态来判断是否需要等待子进程的结束。如果 pid 为-1，表示等待任意子进程结束；如果 pid 为具体的进程 ID，表示等待指定的子进程结束。函数会依次遍历进程表，找到符合条件的

子进程，并根据子进程的状态执行相应的操作，包括释放子进程、设置阻塞状态等。最后根据不同的情况返回不同的标志值，以供调用者判断进程的状态和等待结果。

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

在本次实验中，我按照要求完成了四个工作：修改 PKE 内核和系统调用、数据段复制、wait 函数的实现以及相关函数的补充。在调试过程中，我遇到了一些问题，并从中学到了一些经验和教训。

1. 数据段复制的权限问题：在实现数据段复制时，我忽略了给子进程的数据段赋予可执行权限。这导致子进程无法执行数据段中的代码，引发了异常。通过观察代码和调试输出，我发现了这个问题，并在修改代码后

解决了它。这个问题让我意识到在复制数据段时，不仅需要复制数据本身，还需要注意权限的设置。

2. 进程结束标志的理解：在实现 wait 函数时，我一开始错误地认为进程结束的标志是 FREE。然而，经过调试后我发现，进程的结束是通过调用 exit 函数实现的，并将进程设置为僵尸状态。这个错误让我深入理解了进程的调用和结束过程，并明确了进程结束状态的设置方式。
3. 父进程重新加入队列的问题：在补充相关函数的过程中，我一开始困惑于如何将父进程重新加入就绪队列。经过与同学的讨论和查询资料，我意识到可以在调用 wait 的系统函数和进程结束的系统调用函数中实现这个功能。这个经验教训让我明白了在解决问题时，有时候需要转变思路，不拘泥于某个函数的实现，而是考虑将功能分散到多个函数中实现。

通过本次实验，我对进程调用、等待和退出的实现方式有了更深入的理解，并学会了将理论知识与实践相结合。我还学到了调试代码的技巧，如观察代码、输出调试信息以及与他人交流和寻求帮助。这次实验提升了我在操作系统内核开发方面的能力，并且让我更加熟悉和理解底层代码的运行机制。