

**操作系统原理课程实验报告**

|  |  |
| --- | --- |
| 姓 名： | 刘兴元 |
| 学 院： | 计算机科学与技术学院 |
| 专 业： | 计算机科学与技术 |
| 班 级： | 计算机本硕博2101 |
| 学 号： | U202115663 |
| 指导教师： | 谢美意 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2023年 12月 11 日

目 录

[实验一 打印用户程序调用栈 1](#_Toc153360467)

[1.1实验目的 1](#_Toc153360468)

[1.2实验内容 1](#_Toc153360469)

[1.3实验调试及心得 5](#_Toc153360470)

[实验二 复杂缺页异常 7](#_Toc153360471)

[2.1实验目的 7](#_Toc153360472)

[2.2实验内容 7](#_Toc153360473)

[2.3实验调试及心得 8](#_Toc153360474)

[实验三 进程等待和数据段复制 9](#_Toc153360475)

[3.1实验目的 9](#_Toc153360476)

[3.2实验内容 9](#_Toc153360477)

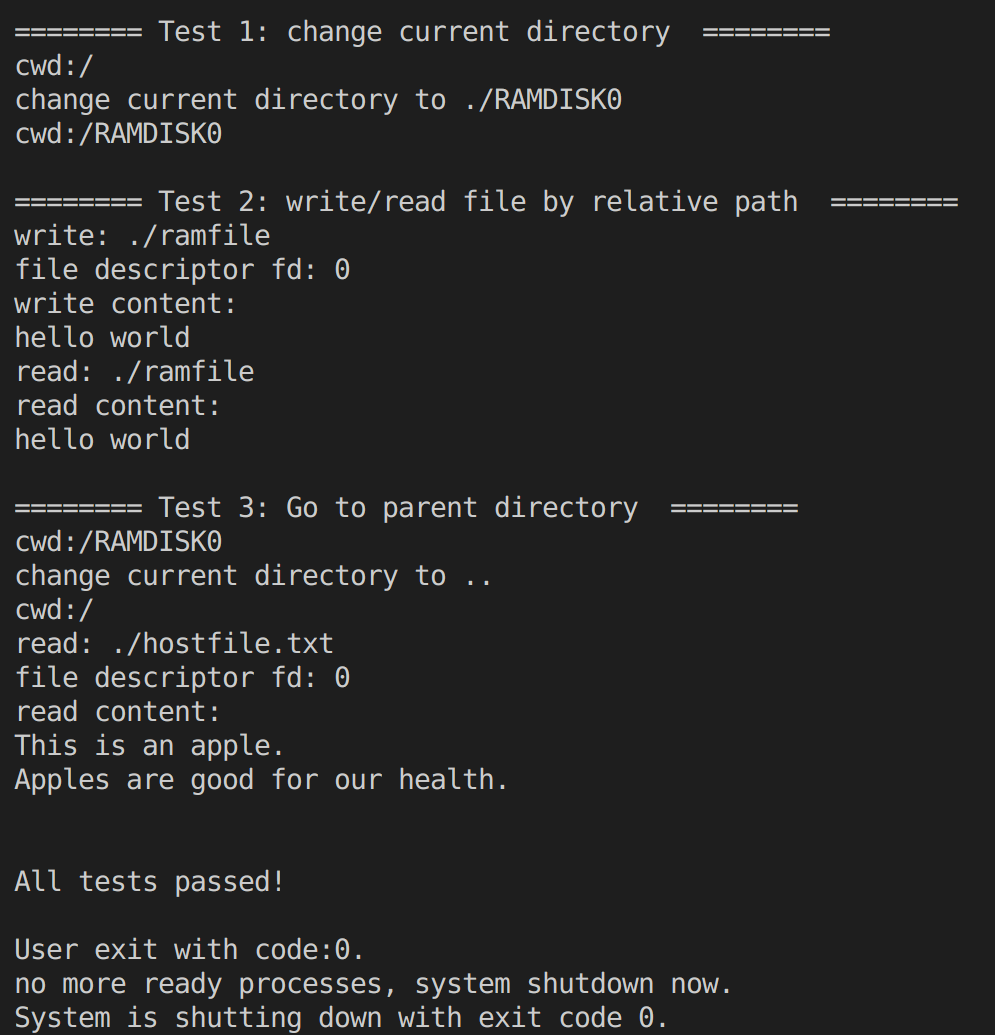
[3.3实验调试及心得 12](#_Toc153360478)

[实验四 多级目录 14](#_Toc153360479)

[4.1实验目的 14](#_Toc153360480)

[4.2实验内容 14](#_Toc153360481)

[4.3实验调试及心得 16](#_Toc153360482)

[ 16](#_Toc153360483)

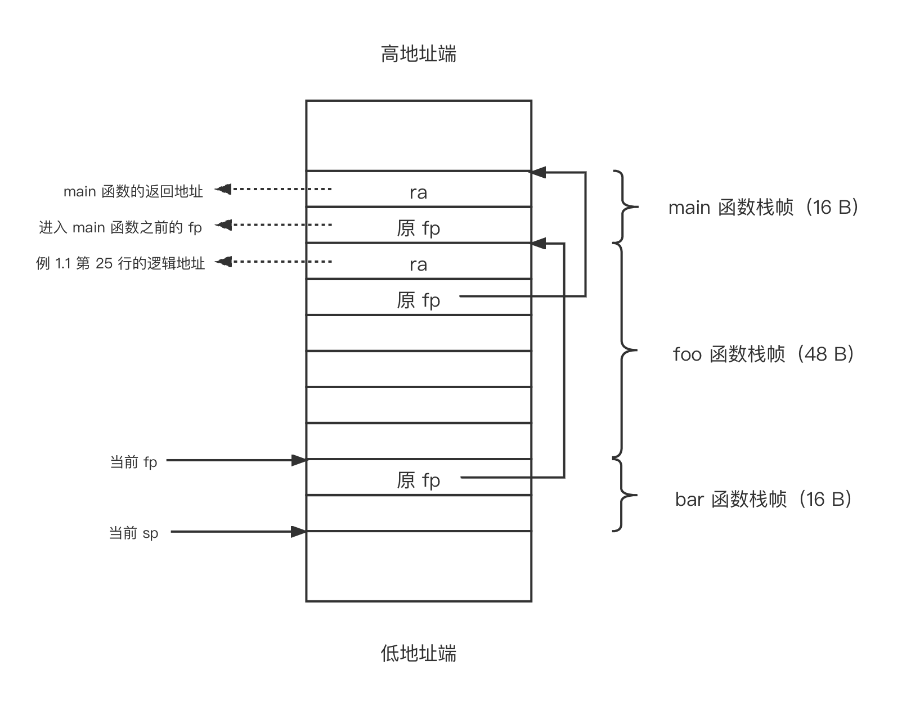
# 实验一 打印用户程序调用栈

## 1.1实验目的

通过修改PKE内核，来实现从给定应用（user/app\_print\_backtrace.c）到预期输出的转换。对于print\_backtrace()函数的实现要求：应用程序调用print\_backtrace()时，应能够通过控制输入的参数（如例子user/app\_print\_backtrace.c中的7）控制回溯的层数。例如，如果调用print\_backtrace(5)则只输出5层回溯；如果调用print\_backtrace(100)，则应只回溯到main函数就停止回溯（因为调用的深度小于100）。我们需要合理地回溯并打印用户进程的函数调用情况。

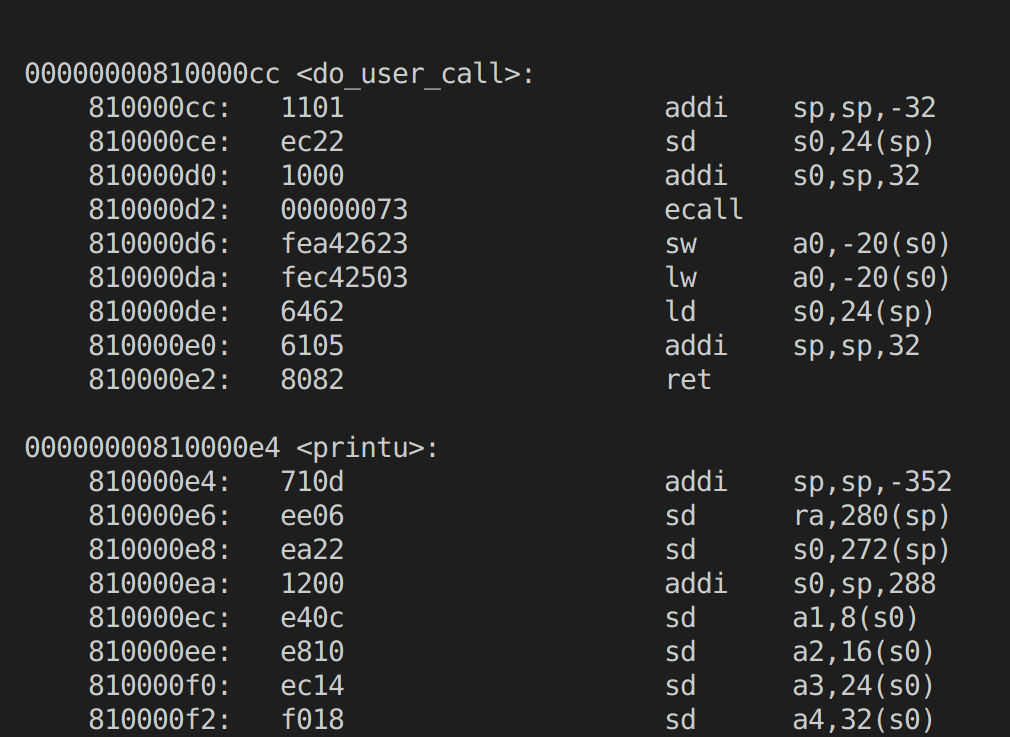
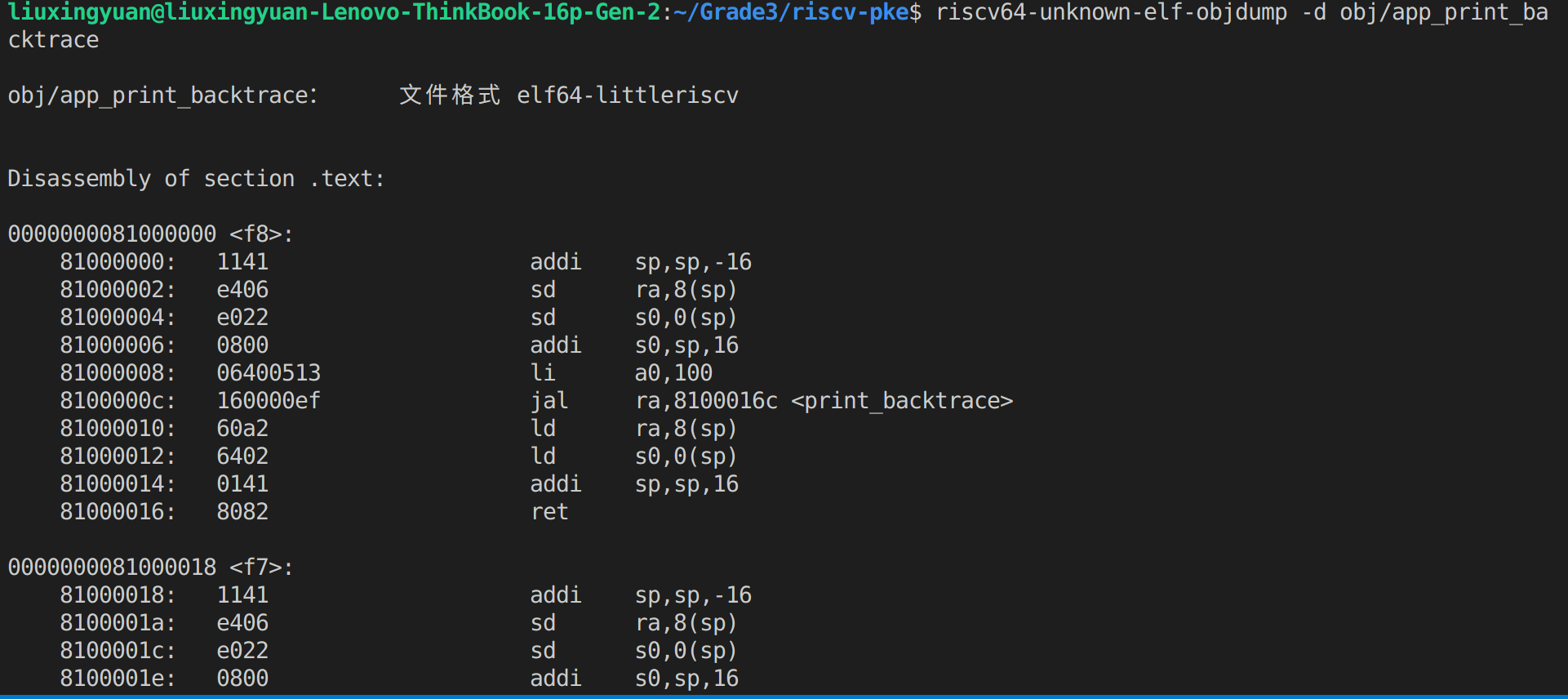
## 1.2实验内容

首先，我们需要弄清楚程序的用户栈帧结构：



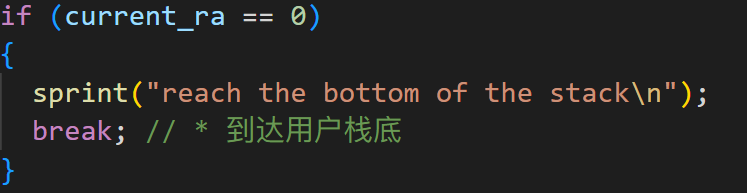
根据上图，我们知道，用户栈是由高地址向低地址增长的，且原fp和ra等指针会储存在栈帧之中。需要注意的是，帧指针 (fp)在函数调用的上下文中，帧指针指向当前函数栈帧的基址。它有助于在执行过程中导航栈。返回地址 (ra)是控制在函数调用后应返回的地址。通常存储在栈上。可以用于识别函数或栈帧中的位置。在后面我们打印调用函数的用户名时，这些指针将会发挥他们的限界的作用。

然后，我们使用反汇编来查看用户栈的实际相互调用的情况：



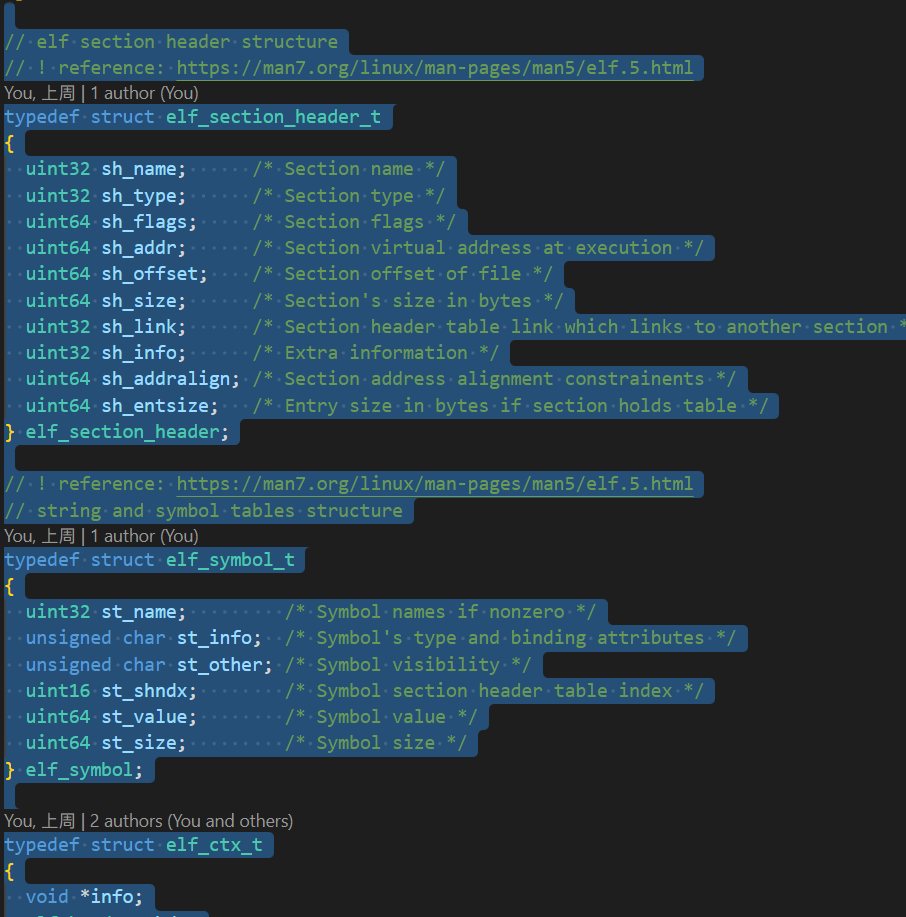
通过使用objdump我们可以输出反汇编代码。我们发现，每一个fx函数占据了16字节的空间，这也是典型栈帧的大小。且是由高地址向低地址不断增长，而user\_lib.h里面的的几个被我们的主函数所调用的函数，则是一开始就被加载好的，处于我们用户栈帧的高地址。

而我们的实验目的，是要回溯并打印用户进程的函数调用情况，例如在我们的backtrace.c 中，函数调用的最末端是f8（不包括外部的函数调用）。且我们发现，f8正好处于用户栈的低地址，且是用户栈的最底部的位置：0x81000000。需要注意的是，这并不是巧合！了解了栈帧结构，我们似乎就能愉快的回溯函数的位置了。值得一提的是，我们需要利用current\_ra限界，如果到达栈底的时候就应该及时停止回溯，不然会发生越界行为（segment fault）。



随后，我们需要打印回溯信息（Printing Backtrace Information），我使用elf\_print\_name函数打印与当前地址相关的信息：

这里也是一个比较难懂的地方，需要在网络上查找学习ELF文件的结构。我通过查阅网址和阅读相关文档。完善了原pke系统中非常不完善的ELF结构。允许程序解析 ELF 文件的内部结构，以便进行各种操作，例如加载程序、执行调试、符号查找等。以便我们对用户栈调用函数名的打印。



elf\_section\_header\_t结构体:定义了ELF文件中节头部的结构。

elf\_symbol\_t结构体:定义了 ELF 文件中符号表项的结构，允许程序检索有关程序中定义和引用的符号的信息。符号表对于链接器和调试器等工具非常重要，能够提供对程序中符号的全面了解。

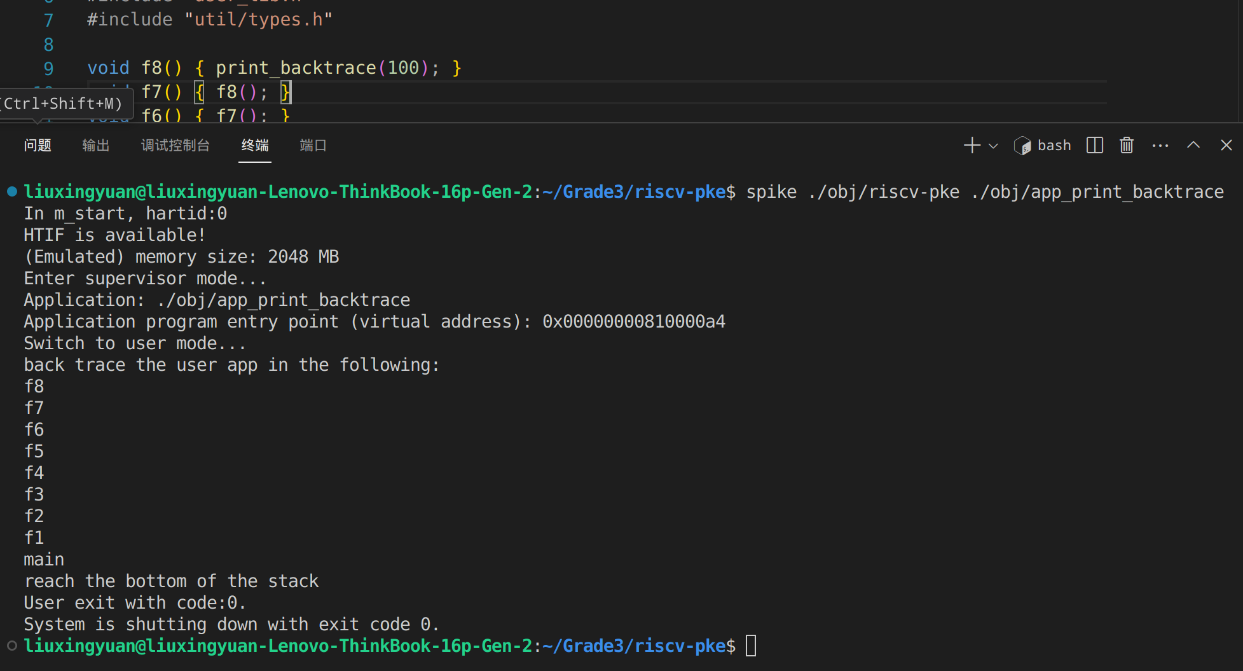
elf\_ctx\_t结构体:定义了 ELF 文件解析的上下文信息。包括了 ELF 文件头部信息、字符串表、符号表等数据结构，以及一个指向任意类型数据的指针，用于存储额外的上下文信息。这种结构提供了一个封装，使得在整个ELF解析过程中能够方便地传递和维护这些信息。

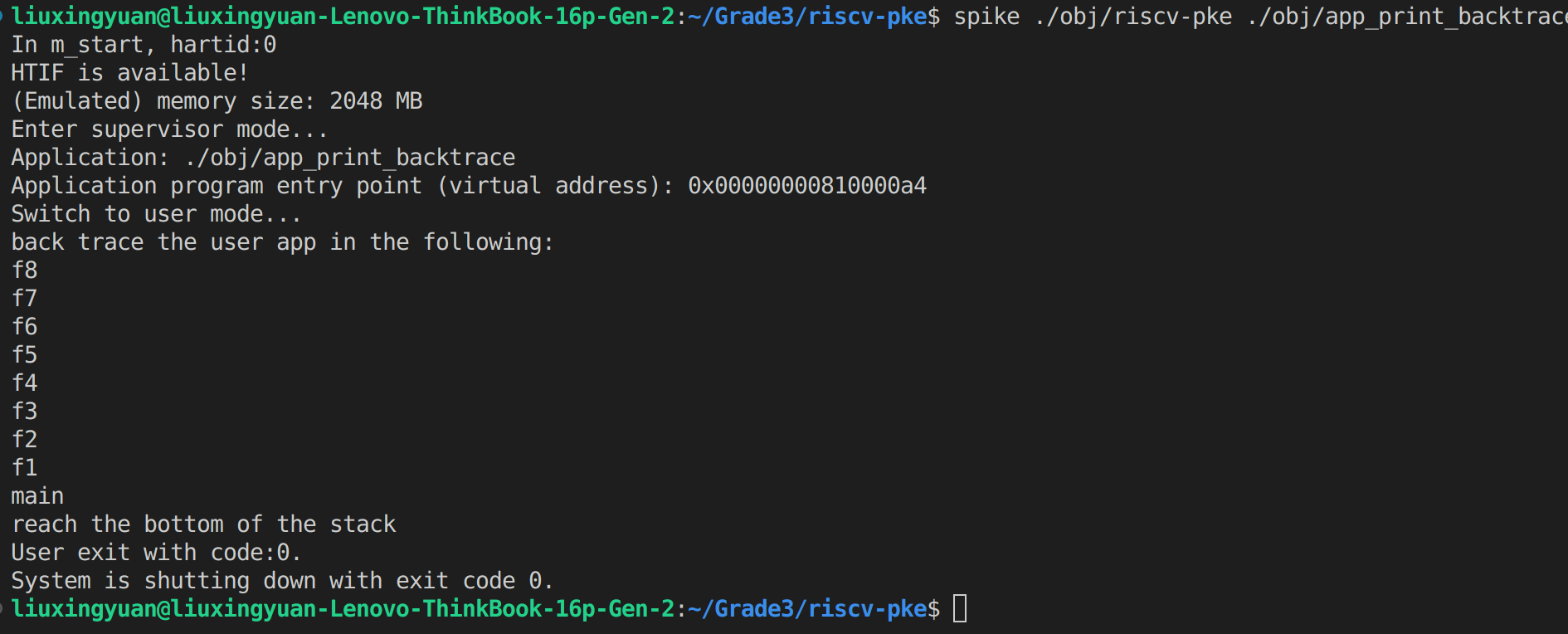
完善了这些，下面的打印就顺风顺水了，其中有一点需要注意的是，Section Header的结构中的name是一个uint32类型的变量，而并不是一个char\*类型的变量。这是因为name本质上来说是相对于某一个地址的偏移。而对于.symtab和.strtab，symbols中的name也又是一个uint32类型的变量。知道这些，我们就知道如何找到函数名了，即如下代码：

&elfloader.str\_table[elfloader.symbol[symbol\_index].st\_name]

随后，我们完善一下elf\_print\_name的相关读取过程，注意边界限界。即可完成此函数的编写。我首先遍历ELF文件的符号表，查找类型为STT\_FUNC且地址范围包含在给定地址ra之前的最大地址的符号。如果找到符号，则打印符号名称（前提是该符号的值大于等于 0x81000000）。如果找不到符号，则输出失败信息。

## 1.3实验调试及心得

完善好相关数据结构，且完善好相关函数，我们就能够准确的回溯并打印我们的用户栈啦：



上面两图分别为回溯层数为100层和7层的结果，我们发现函数能够回溯正确的层数并能正确的停止回溯。符合设计的要求。

在lab1\_challenge1的完成过程中，我实际上遇到了诸多的困难：

一开始在打印用户函数名的时候，出来的是一堆乱码。后面输出各种东西之后才知道，应该需要加上elfloader.symbol[i].st\_info == STT\_FUNC选出符号表中仅为函数（STT\_FUNC）的符号。

此外，我在调试过程中，在能输出用户调用栈函数的同时，也错误的多输出了 “vsnprintf” 的外部函数的函数名，后面debug发现需要限界。限界之后方可解决此问题。

# 实验二 复杂缺页异常

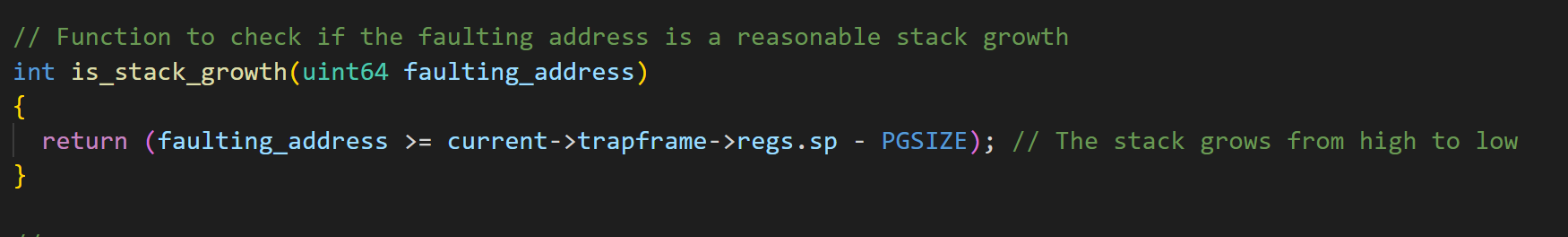
## 2.1实验目的

在创建数组时，我们使用了当前的malloc函数申请了一个页面（4KB）的大小，对应可以存储的个数上限为1024。在函数调用时，我们试图计算1025求和，首先由于n足够大，所以在函数递归执行时会触发用户栈的缺页，你需要对其进行正确处理，确保程序正确运行；其次，1025在最后一次计算时会访问数组越界地址，由于该处虚拟地址尚未有对应的物理地址映射，因此属于非法地址的访问，这是不被允许的，对于这种缺页异常，我们应该提示用户并退出程序执行。

## 2.2实验内容

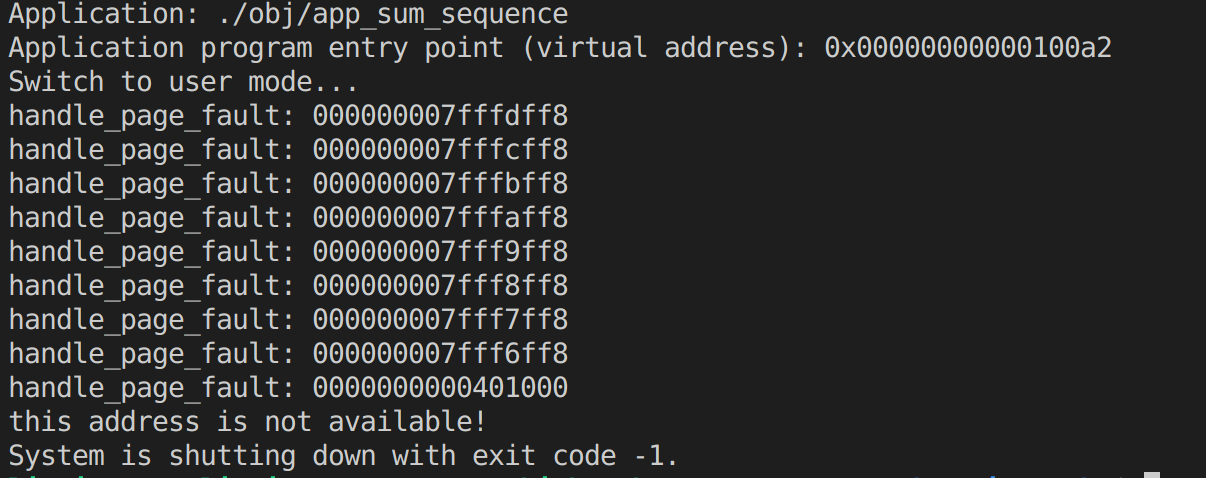
程序如果对动态申请的数组越界访问，我们需要立即退出程序。

观察得到，我们需要修改strap.c中的handle\_user\_page\_fault函数。如果动态分配的地址不在合理的栈范围内，触发 panic，并表示这个地址不可用。

我设计了is\_stack\_growth函数，检查存储页面错误发生的地址stval是否在合理的范围内，以确定是否是合理的栈的增长情况。

如果是合理栈增长，调用map\_pages函数将新分配的物理页映射到发生页面错误的地址stval所在的页。如果地址不在合理的栈范围内，触发panic，并表示这个地址不可用。

## 2.3实验调试及心得



经过修改，现在的pke系统能自动识别新地址在不在合理的栈范围内，如果不在则中止程序。

总的来说，这个challenge比较基础。增加了我继续做下去的信心。

# 实验三 进程等待和数据段复制

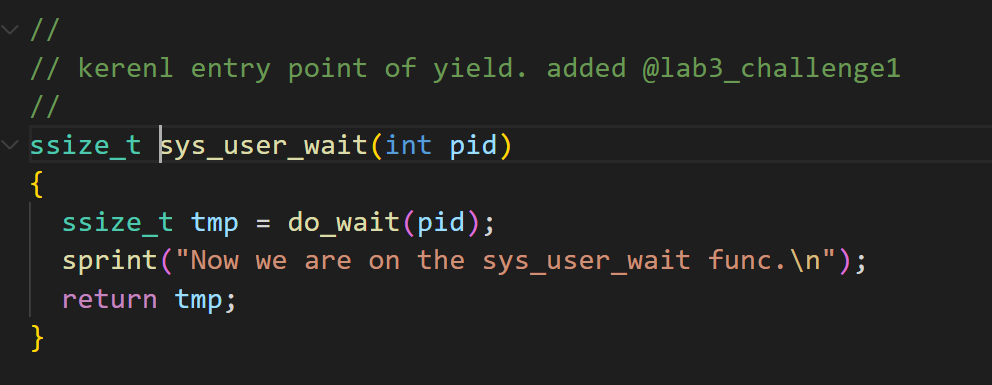
## 3.1实验目的

本实验的要求我们，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，wait函数接受一个参数pid：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；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；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。

同时，需要我们补充do\_fork函数，实验3\_1实现了代码段的复制，需要继续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

## 3.2实验内容

通过完善系统调用，我们现在需要实现do\_wait函数：



对于do\_wait函数，我们需要明确wait函数接受一个参数pid：

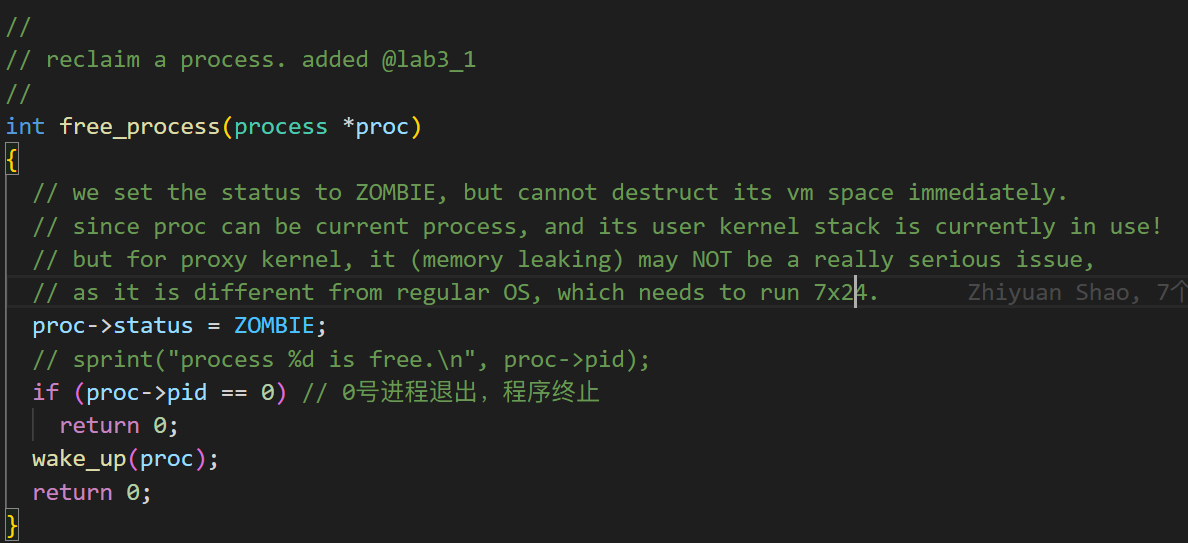
如果 pid 为 -1：循环遍历系统中的所有进程。对于每个进程，检查其状态是否为 FREE 或者其父进程是否为空。如果是，继续下一次循环。如果找到一个子进程的父进程是当前进程，表示找到了一个待检查的子进程。如果该子进程的状态是 ZOMBIE（僵尸状态），则将其状态设置为 FREE，表示该子进程已退出，然后返回该子进程的 pid。如果没有找到僵尸子进程，但找到了子进程，将当前进程插入到阻塞队列，表示当前进程需要等待子进程的退出。然后调用schedule()，挂起当前进程。

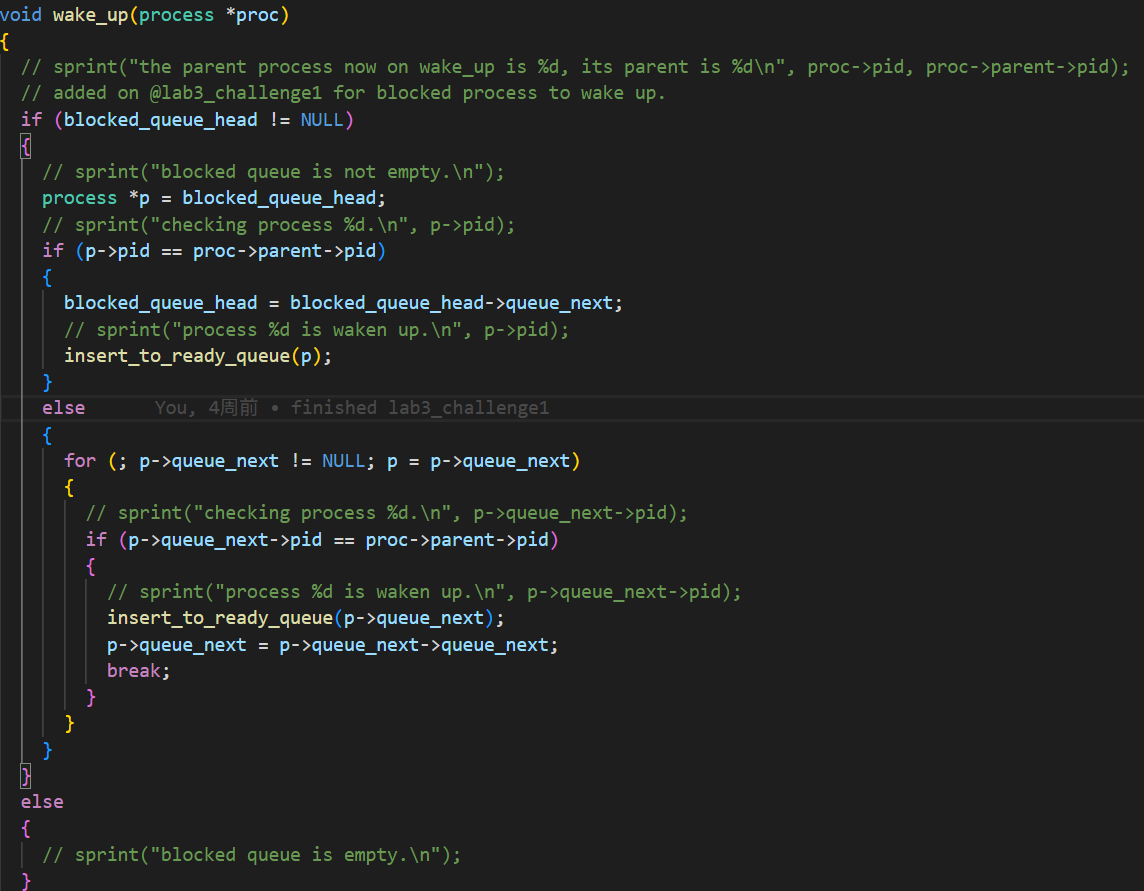
如果 pid 大于 0时，如果传入的 pid 在有效的范围内（大于 0 且小于 NPROC），且该进程的父进程是当前进程。如果该子进程的状态是 ZOMBIE，表示子进程已退出，直接返回该子进程的 pid。如果子进程还没有退出，则将当前进程插入到阻塞队列，表示当前进程需要等待特定子进程的退出。然后调用schedule()，挂起当前进程。

其他情况：如果传入的 pid 不满足上述条件，返回 -1，表示参数无效。

在写完了我们的do\_wait函数之后，显然，我们需要扩充schedual和能够将进程插入blocked\_queue 的函数。我仿照原有的insert\_to\_ready\_queue()函数，完善了相关的数据结构，即增加blocked\_queue和增加相应状态；并写了insert\_to\_ready\_queue()。

此外，我实现了一个函数 wake\_up，其主要作用是将被运行进程阻塞的进程从阻塞队列中唤醒，并移动到就绪队列中。同时修改free\_process: 当一个进程结束时使用wake\_up函数，即current的进程结束时，我们需要将被其阻塞的进程移动到ready\_queue中。Free\_process和wake\_up函数如下两图所示：



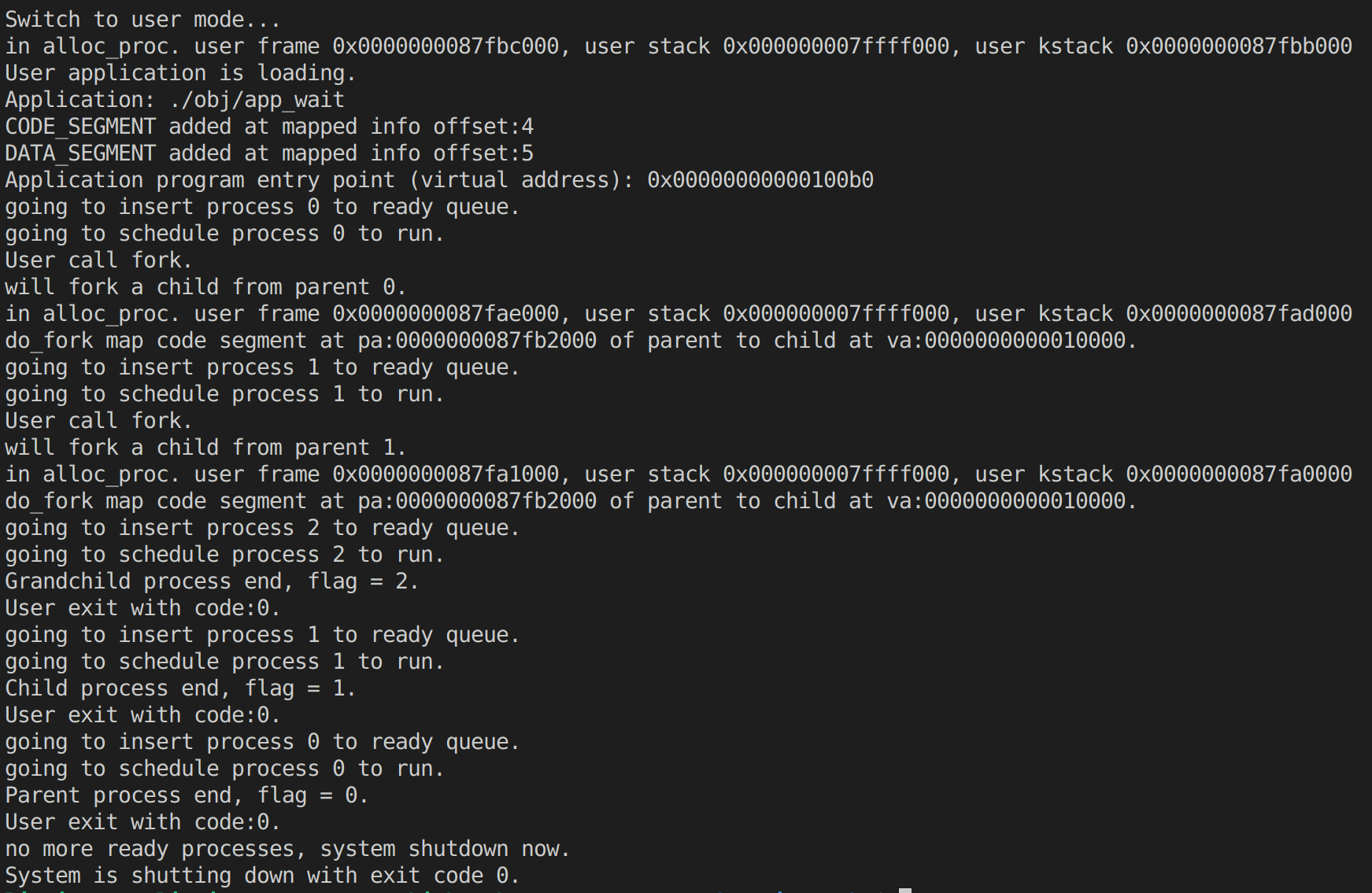


同时，我完善了do\_fork函数，这里只需要完善部分代码。

至此，我们已经成功实现了pke系统的wait功能。

## 3.3实验调试及心得

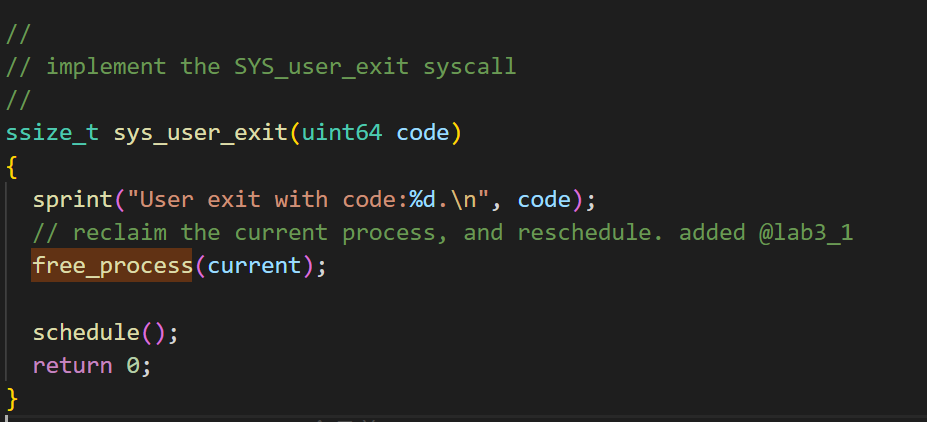
经过测试，我设计的wait()函数和相应数据结构和相关函数能正常运行我们的app\_wait.c函数。



在这个实验过程中，我在单纯的wait()函数的编写时没有遇到什么困难。主要的debug时间集中于我的wake\_up()函数设计部分。

我一开始不知道为什么，我的process0 会先于我的 process1结束，阻塞失效了。后面经过debug发现，不能单纯的使用current->process作为wake\_up的参数，因为此时current->process在free\_process之后立即schedule()完成了切换（如下图）。

故我需要在schedule之前保存现在的将要free的current\_process 。所以我后面将wake\_up放置在free\_process() 返回之前，就成功的解决了这个调度问题。



别的地方总的来说没有什么大的问题，对于一些奇怪的外部引用，我尽量把要用到这个变量，如blocked\_queue\_head。移动到一个.c函数下即可解决。

# 实验四 多级目录

## 4.1实验目的

相对路径即形如：“./file”、“./dir/file”以及“../dir/file”的路径形式。与绝对路径总是从根目录开始逐级指定文件或目录在目录树中的位置不同，相对路径从进程的“当前工作目录“开始，对一个文件或目录在目录树中的位置进行描述。

通过修改PKE文件系统代码，提供解析相对路径的支持。具体来说，用户应用程序在使用任何需要传递路径字符串的函数时，都可以传递相对路径而不会影响程序的功能。同时完成用户层pwd函数（显示进程当前工作目录）、cd函数（切换进程当前工作目录）。

## 4.2实验内容

对于pwd()函数，实现比较简单：使用 strcpy 函数将当前进程的工作目录路径（current->pfiles->cwd->name）复制到由 path 参数指定的用户空间地址。这里需要注意的是，需要使用user\_va\_to\_pa 函数，将用户空间地址转换为物理地址，以便写入由 path 参数指定的用户空间地址。

接下来我们将完成cd函数的实现：

从路径字符串形式的角度来看，相对路径的起始处总是为以下两种特殊的目录之一：”.“，”..“。其中”.“代指进程的当前工作目录，“..”代指进程当前工作目录的父目录。例如，“./file”的含义为：位于进程当前工作目录下的file文件； “../dir/file”的含义为：当前进程工作目录父目录下的dir目录下的file文件。

了解了这些，我们就可以针对上面的要求实现：

首先，检查当前工作目录是否为空：代码检查当前进程的文件描述符表中的当前工作目录是否为 NULL。如果为 NULL，说明当前进程没有有效的工作目录，此时返回错误码 -1。如果有，使用 user\_va\_to\_pa 函数将指定路径 path 的虚拟地址转换为物理地址，得到 pathpa。

如果路径以 ".." 开头，表示需要返回到父目录。首先将当前目录路径设置为其父目录的路径。将当前目录的父目录与当前目录建立连接，并释放原先的父目录占用的页。更新路径指针 pathpa，使其指向 "/" 之后的路径。将 pathpa 指向的字符串拼接到当前目录路径之后。

如果路径以 "." 开头（不是 ".."），表示继续在当前目录下操作。将 pathpa 指针移动到字符串的下一个字符，再将其拼接到当前目录路径之后。

如果路径不是 ".." 也不是 "."，则直接将 pathpa 指向的路径赋值给当前目录路径。

到这里，似乎我们的lab4\_challenge1就已经做完了。但是实际上并没有！我们还需要完成do\_open函数对我们给定的相应路径的解析过程：

do\_open函数处理过程需要考虑了相对路径的情况，包括返回到父目录和继续在当前目录下操作。

如果路径以 ".." 开头，表示需要返回到父目录。首先将当前目录路径设置为其父目录的路径。更新路径指针 path，使其指向 "/" 之后的路径。将 path 指向的字符串拼接到当前目录路径之后，得到解析后的路径 parsed\_path。

如果路径以 "." 开头（不是 ".."），表示继续在当前目录下操作。将 path 指针移动到字符串的下一个字符。将 current->pfiles->cwd->name（当前目录路径）拼接到 path 之后，得到解析后的路径 parsed\_path。

如果路径不是 ".." 也不是 "."，则直接将 path 指向的路径赋值给解析后的路径 parsed\_path。

至此，我们已经完成了lab4\_c1对相对路径解析的实现。

## 4.3实验调试及心得

## 

由上图，我设计的pwd， cd， open函数正常实现。

而在编写代码的过程中，我也遇到了一些问题，一开始我根本没有想到要修改do\_open函数，后面运行时发现无法打开文件，才知道do\_open原来也是需要我去完善的，好在，在学习了前置的知识后，完成比较简单。

整个这次实验的核心函数应该是user\_va\_to\_pa，一个路径是current->pfiles->cwd->name。明确这两个，编写代码应该会比较流畅。