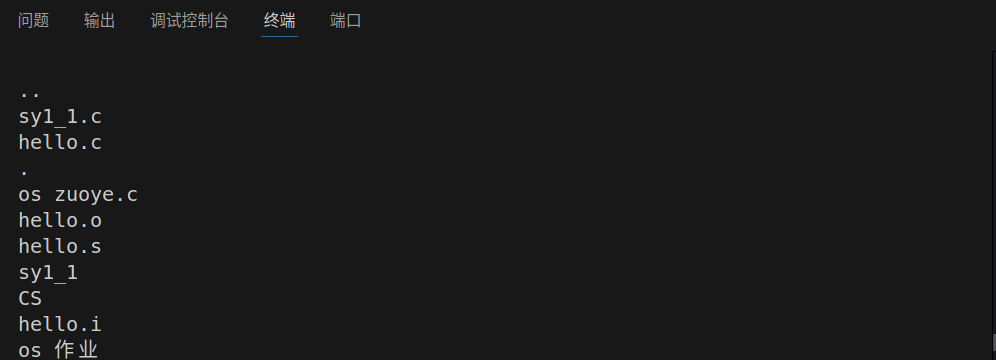
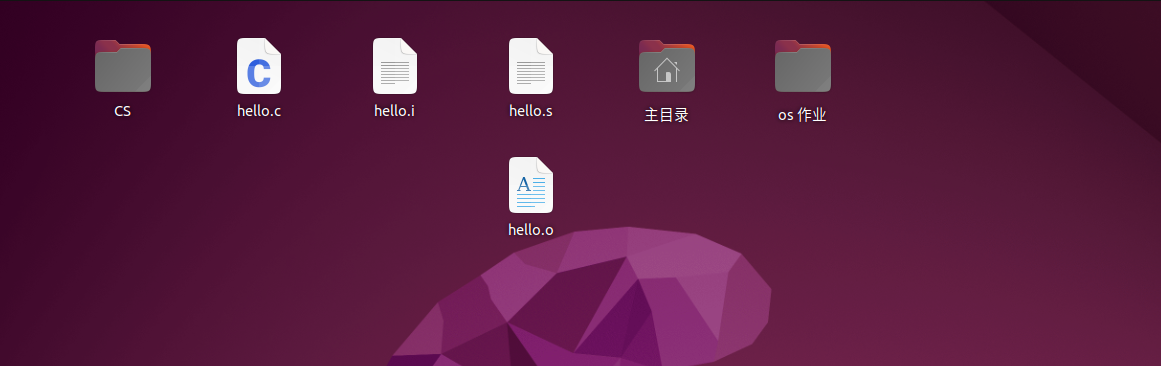
编程作业一：





程序的结果



可以看到正确地表示出了我桌面上的文件。

这段代码使用了C语言中的标准库函数和系统调用，其作用是打开当前工作目录并列出其中的所有文件

和子目录的名称。具体解释如下：

#include <dirent.h> ：包含了用于操作目录的相关函数和数据结构的头文件。

#include <stdio.h> ：包含了用于输入输出的相关函数的头文件。

DIR \*dir = opendir(".");：打开当前工作目录（即“.”表示当前目录），并返回一个指向该目录

流的指针（即DIR \*类型的变量dir）。

struct dirent \*entry;：定义了一个指向目录项的指针。

while ((entry = readdir(dir)))：循环读取dir指向的目录中的每个目录项，并将其赋值给

entry指针，当读取结束后readdir()函数会返回NULL，循环将结束。

printf("%s\n", entry->d\_name); ：输出entry指向的目录项的名称。

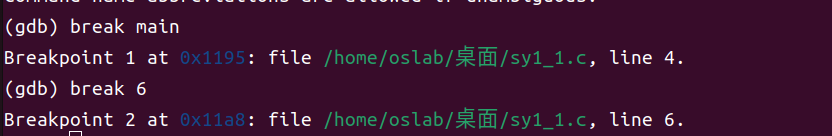
return 0; ：程序正常退出。

接下来我们使用gdb来调试该文件：

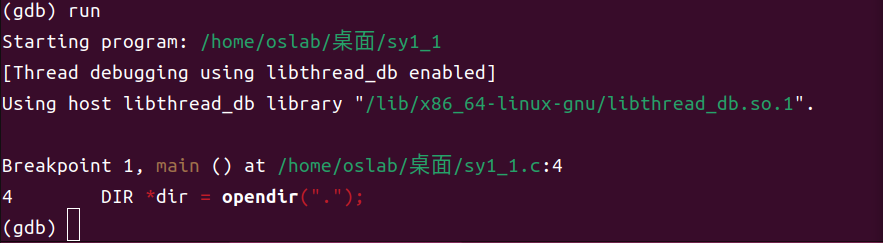


先使用gdb打开编译后的文件。

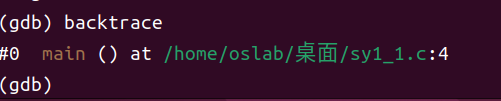
接着在main和whlile循环出设置断点：



Run程序：



看到在第一个断点处停下，下一行代码为DIR \*dir = opendir(".");



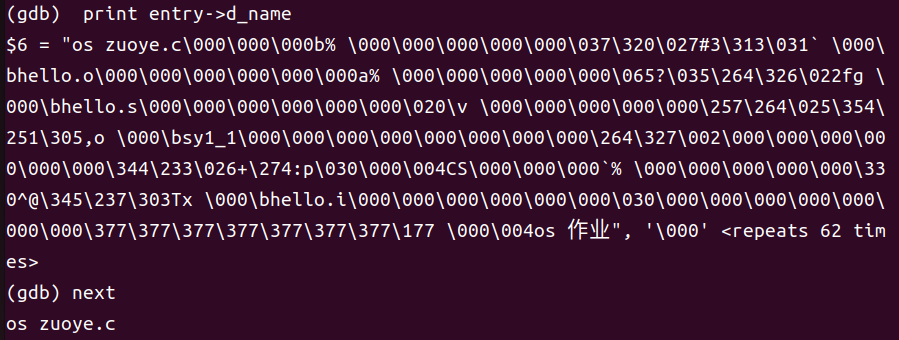
我们使用backtrace来查看调用栈的信息，这里返回了main。

我们继续往下运行：



看到程序开始打印结果也就是我桌面上面的文件了。

我们还可以使用print entry->d\_name来看当前处在循环中的文件名字：

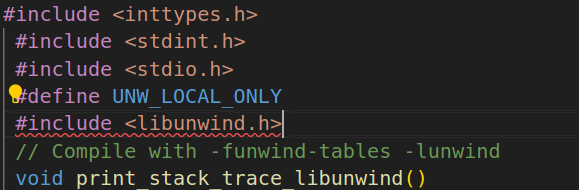


这里有个注意的点：GDB 有时会显示完整的数组内容，特别是在用 print 命令查看结构体成员时，它会显示数组中所有元素，而不仅仅是作为 C 字符串解析的部分。

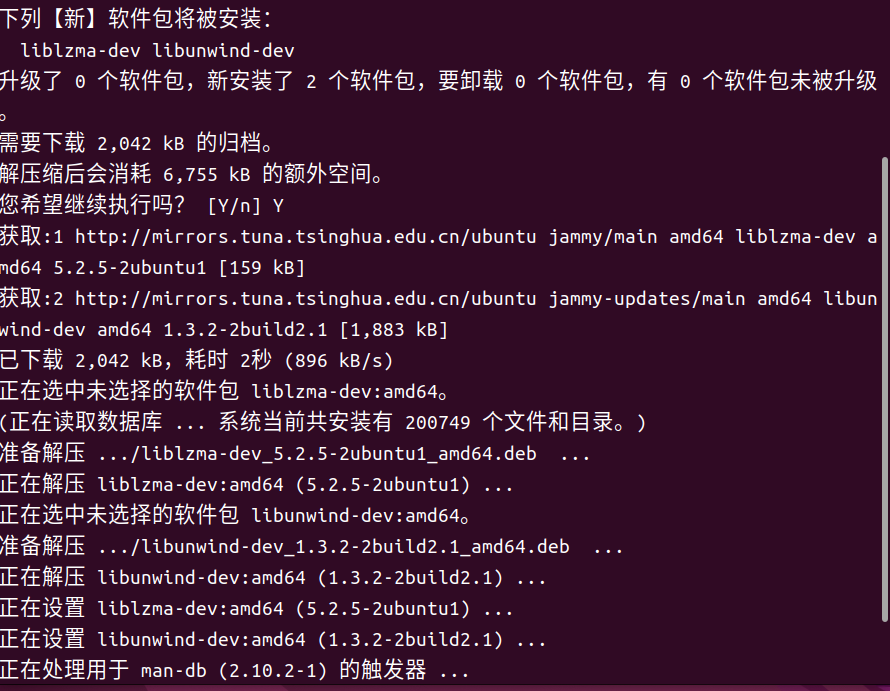
因此，虽然看到了一串包含“CS”、"hello.i"、"os 作业"等内容以及大量的 '\000'（空字符）和其他数据，但实际上有效的文件名只在第一个 '\0' 之前，后面的数据可能是内存中的残留数据或未初始化部分。

1\_2:

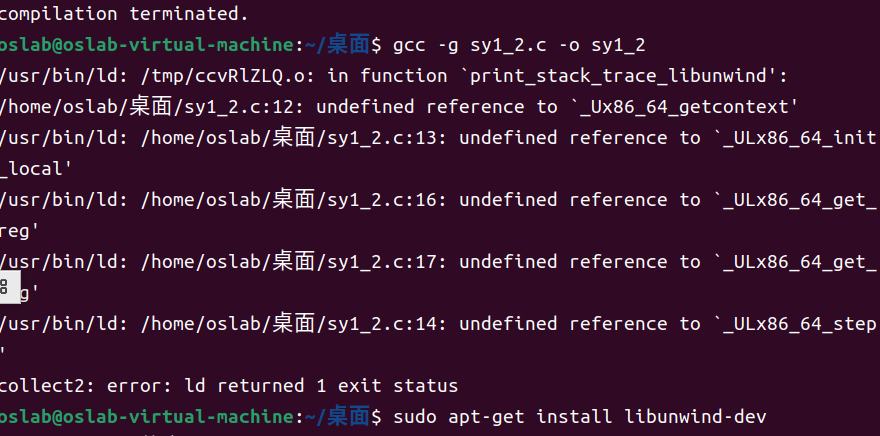
实现一个linux应用程序B，能打印出调用栈链信息。（用C或Rust编程）：



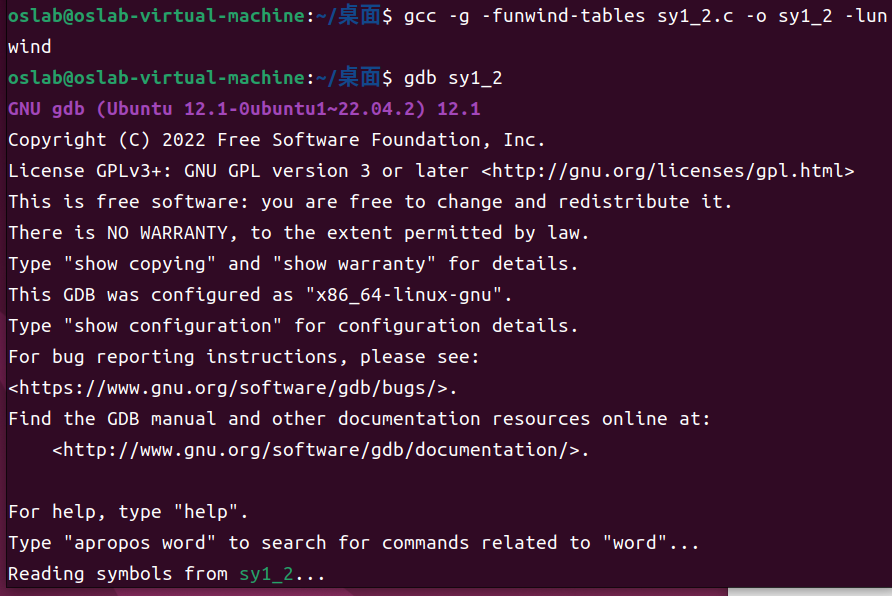
提示我们要先安装libunwind：



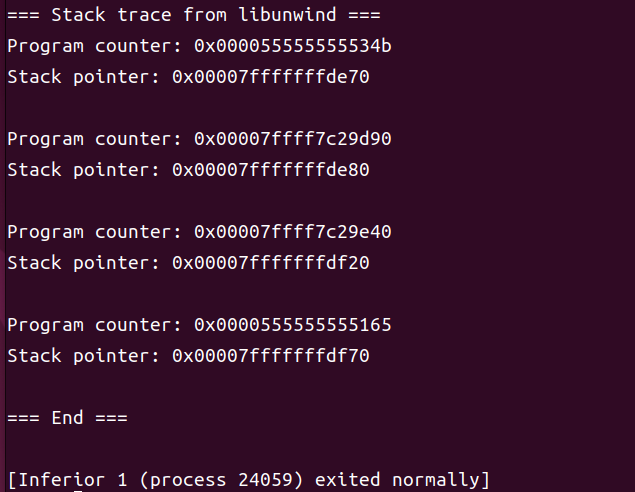
使用gcc编译文件：



发现出现问题，需要在常规的编译指令中加入-lunwind表示按照这个规定编译：



我们直接run看看结果：



对代码的分析如下：

#include <inttypes.h>

#include <stdint.h>

#include <stdio.h>

这些是标准 C 库的头文件，其中包括格式化输出所需的 stdio.h 和整数类型所需的 inttypes.h 和 stdint.h。类似PRIxPTR

#define UNW\_LOCAL\_ONLY：指示 libunwind 只处理本地的栈展开，不考虑远程或其它复杂场景。

#include <libunwind.h>

定义了一个名为 UNW\_LOCAL\_ONLY 的宏并包含了 libunwind.h 头文件，该库提供了访问程序运行时堆栈跟踪信息的功能。

// Compile with -funwind-tables -lunwind

该注释提供了编译该程序时需要添加的选项，其中 -funwind-tables 和 libunwind 库的支持。

声明变量：

unw\_cursor\_t cursor;：用来遍历栈帧的游标。

unw\_context\_t uc;：用于保存当前的执行上下文（context）。

unw\_word\_t pc, sp;：分别用于存储每个栈帧中的程序计数器（PC）和栈指针（SP）

void print\_stack\_trace\_libunwind()

{

printf("=== Stack trace from libunwind ===\n");

unw\_cursor\_t cursor; unw\_context\_t uc;

unw\_word\_t pc, sp;

unw\_getcontext(&uc);

unw\_init\_local(&cursor, &uc);

while (unw\_step(&cursor) > 0)

{

unw\_get\_reg(&cursor, UNW\_REG\_IP, &pc);

unw\_get\_reg(&cursor, UNW\_REG\_SP, &sp);-lunwind 用于启用对

printf("Program counter: 0x%016" PRIxPTR "\n", (uintptr\_t) pc);

printf("Stack pointer: 0x%016" PRIxPTR "\n", (uintptr\_t) sp);

printf("\n");

}

printf("=== End ===\n\n");

}

这个函数使用了 libunwind 库来打印函数调用堆栈。具体实现步骤如下：

1. 首先打印一条开始标记

2. 初始化一个

unw\_getcontext(&uc) 。

unw\_context\_t cursor

3. 初始化一个

uc ，并获取当前线程的上下文信息，即调用

unw\_cursor\_t cursor ，并将其指向当前线程的栈顶，即调用 unw\_init\_local(&cursor, &uc) 。

4. 循环遍历调用堆栈，即调用

unw\_step(&cursor) ，直到遍历完整个调用堆栈。

unw\_step() 返回一个

int 类型的值，表示执行一次堆栈帧的遍历，可能会有三种返回值：

UNW\_OK ：表示遍历成功，下一个堆栈帧已被加载到 cursor 中，返回 1。

UNW\_END\_OF\_FRAME : 表示已到达堆栈的顶部，无法再向上遍历堆栈，返回 0。

UNW\_EUNSPEC : 表示遍历失败，原因不明确，会返回一个负数。

5. 在循环体中，每次调用 unw\_get\_reg(&cursor, UNW\_REG\_IP, &pc) 获取当前函数的程序计数器 (Program Counter) 的值，

调用 unw\_get\_reg(&cursor, UNW\_REG\_SP, &sp) 获取当前函数的栈指针 (Stack Pointer) 的值，并打印这两个值。

6. 循环结束后，打印一条结束标记。

int main()

{

print\_stack\_trace\_libunwind();

return 0;

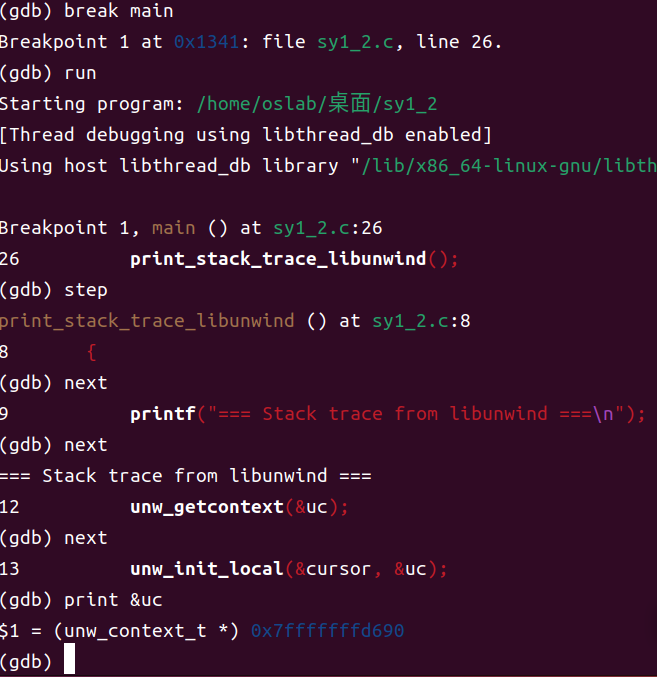
}

这是程序的主函数，它调用

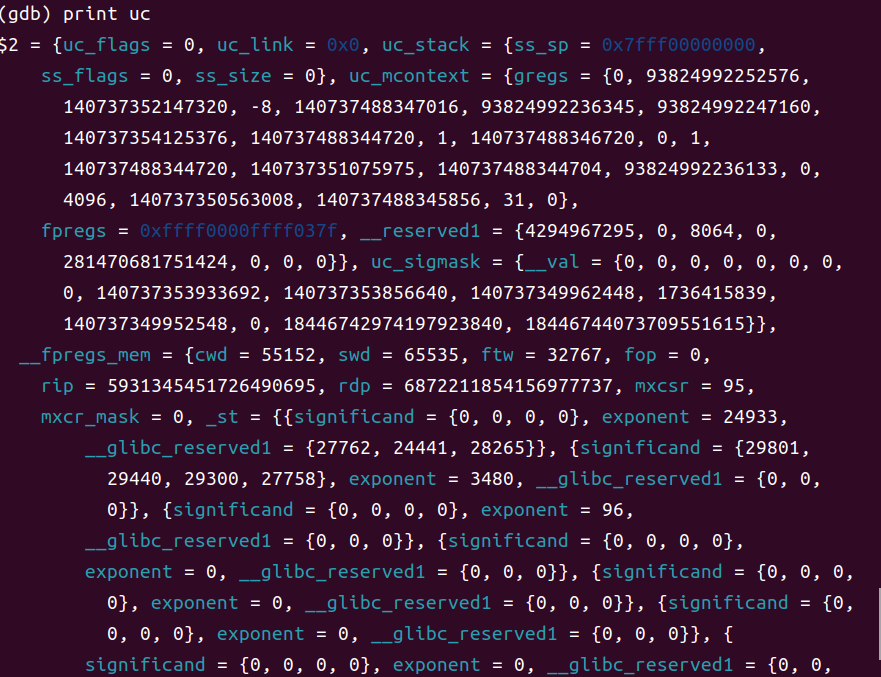
print\_stack\_trace\_libunwind 函数并返回 0，表示程序正常结束.

使用gdb来调试该程序：

首先进入函数，在函数调用开始前设置断点，然后用step指令进入函数，之后使用next指令执行函数代码：

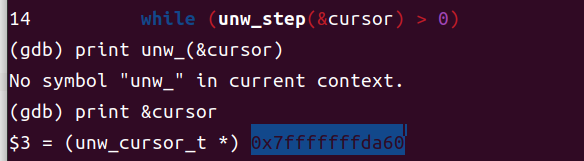


我们打印出当前调用栈最底层的进程状态的地址：0x7fffffffd690，然后我们试着打印uc结构体本身：

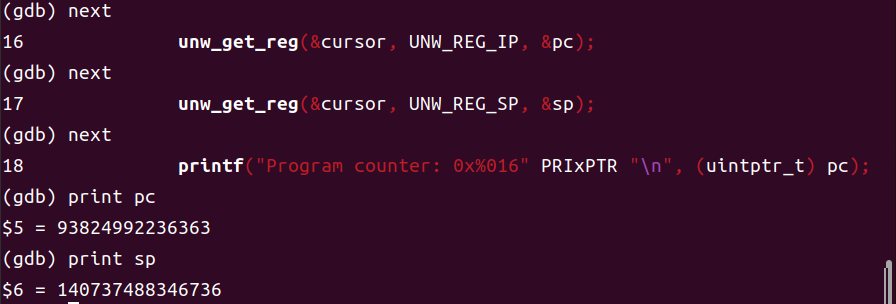


里面存着当前程序在这一帧的状态。

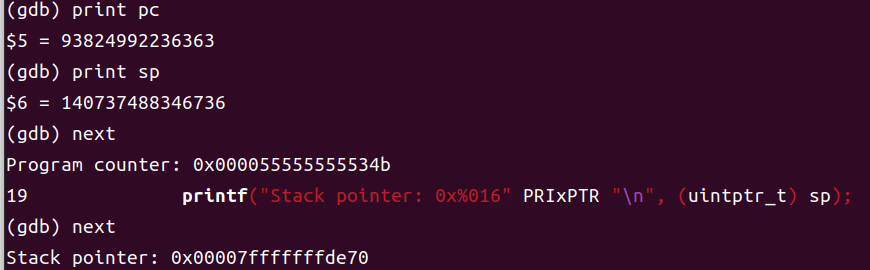
再试着打印&cursor：



打印pc和sp的值：

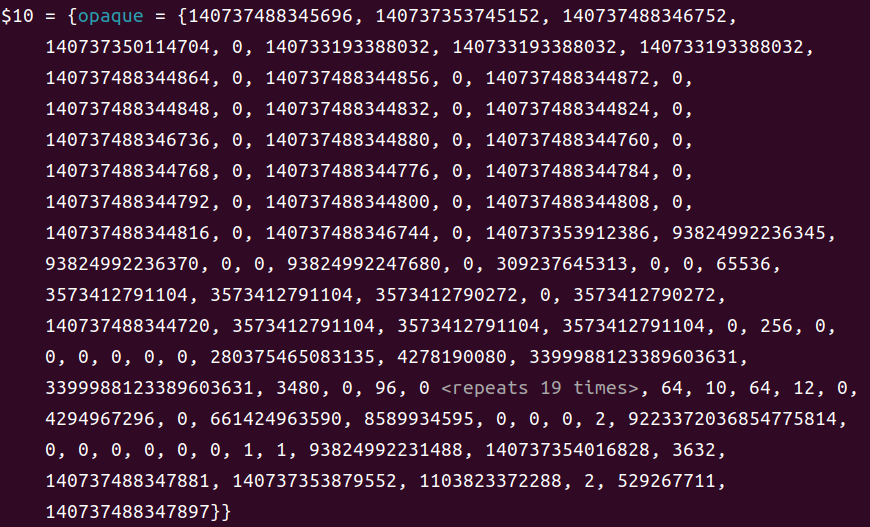


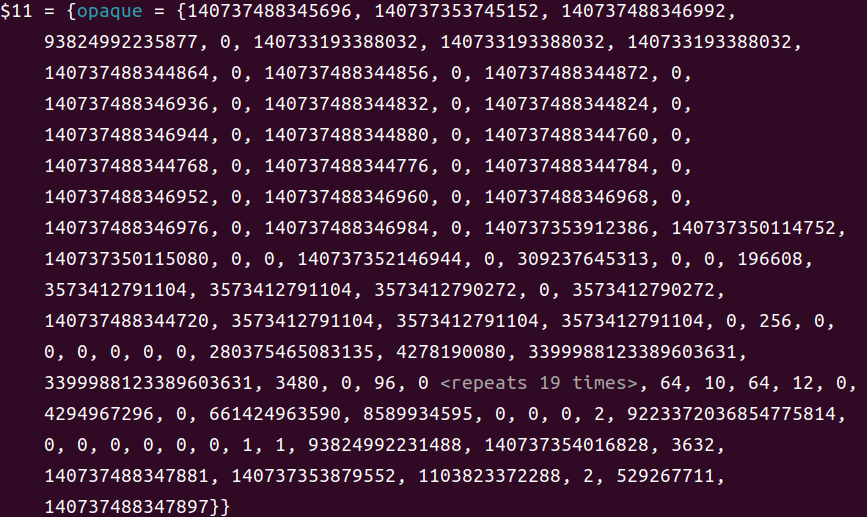
继续执行代码：



自己打印的pc和sp的值是十进制而代码输出的值为十六进制。

对比调用函数前后的cursor的值：





对比两图，可以发现有些值发生了改变：



说明游标cursor在往调用链上层移动。

问答题：

1. 应用程序在执行过程中，会占用哪些计算机资源？

答：CPU资源：用于执行程序的指令和计算任务，占用中央处理器的时间片。

内存（RAM）：存储程序代码、数据以及运行时堆栈，确保程序在运行时有足够的空间存放临时数据。

存储设备资源（磁盘I/O）：用于读写数据、加载程序文件及存储临时或持久性数据。

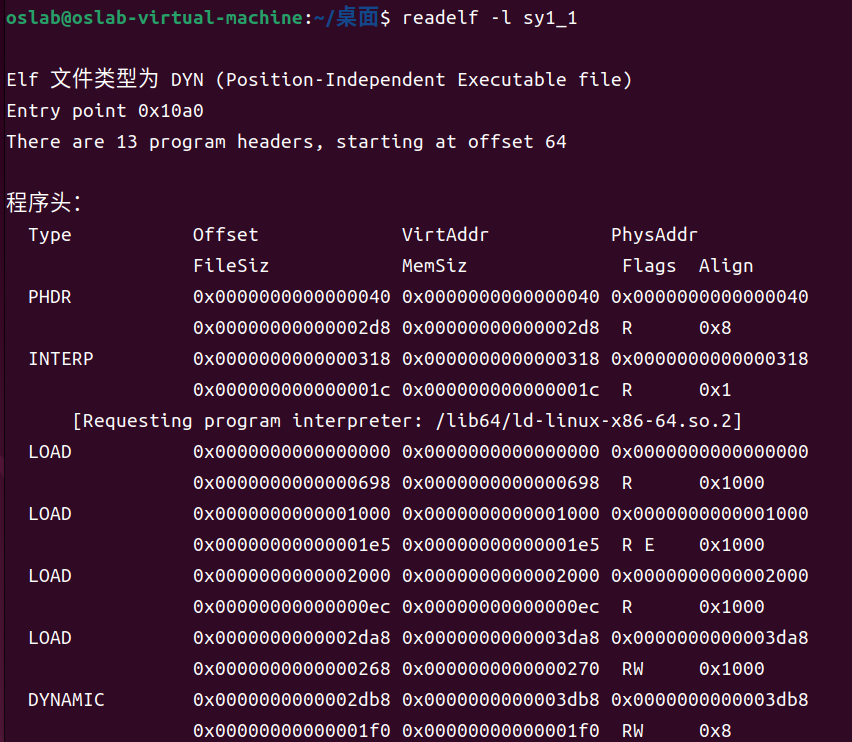
网络资源：当应用程序需要进行网络通信时，会占用网络接口和带宽。

文件描述符：应用程序在打开文件或设备时，会占用系统分配的文件描述符以进行读写操作。

其他资源：根据应用场景，可能还会占用GPU（图形处理单元）资源、系统调用接口、进程/线程管理资源等。

2 请用相关工具软件分析并给出应用程序A的代码段/数据段/堆/栈的 地址空间范围。

我们可以在 Linux/Unix 环境下使用 readelf 工具来查看 ELF 格式文件的程序头信息，程序头信息里面会包含代码段和数据段的信息：



Load表示需要加载到内存中的段（代码段、数据段等）。其中Flag是R E（只读和可执行）的为代码段，flag是R W（可读和可写入）的是数据段。想要知道其地址范围，我们可以利用virtaddr（起始地址）和memsiz（内存大小）来进行判断，其范围就是起始地址到起始地址加上其内存大小：

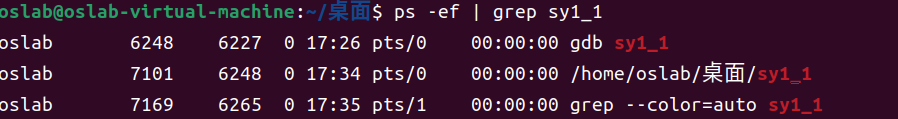
得到代码段的地址范围为：0x0000000000001000到0x00000000000011e5

数据段的地址范围为：0x0000000000003da8到0x0000000000004018

然后找堆和栈的地址范围：

想找到堆和栈的地址，我们需要找到代码一运行时候的进程：

使用指令ps -ef | grep sy1\_1，该条指令前半部分为列举出当前进程，后半部分为找出其中关于sy1\_1的部分。在使用该指令之前，我们需要先为代码1-1打开一个进程。



第二行才是我们的程序本身，第一行是gdb调试进程，第三行是grep本身的程序，看到这里有两个数：7101和6248，前者是其pid后者是其ppid（父进程pid）.

我们用cat指令进入其map文件“cat /proc/7101/maps”。





所以我们得到了程序堆的地址范围为0x5af4560c8000-0x5af45622f000.栈的地址范围为：0x7ffffffde000-0x7fffffffff000.

3请简要说明应用程序与操作系统的异同之处。

相异点：

1. 目的不同：操作系统的主要目的是管理计算机硬件和软件资源，提供用户与计算机系统之间的接口，而应用程序的主要目的是为用户提供特定的功能或服务。

2. 运行环境不同：应用程序是在操作系统上运行的，它们使用操作系统提供的资源和服务，而操作系统则是在计算机硬件上运行的，它们管理计算机硬件和提供对应用程序的支持。

3. 开发过程不同：应用程序通常是由软件开发人员或团队编写的，而操作系统通常是由大型软件公司或组织开发的，需要投入大量的时间和资源。

4. 版本控制不同：应用程序通常有版本控制机制，这样用户可以选择是否升级到新版本，而操作系统通常只有一种版本，在一段时间后由新的操作系统替代。

5. 功能和服务不同：应用程序提供特定的功能和服务，例如办公软件、游戏、音频和视频编辑等，而操作系统提供更基本的服务，例如文件系统、网络支持、安全和用户管理等。

相同点：

1. 都是计算机系统中不可或缺的组成部分。

2. 都需要使用计算机硬件资源和软件服务。

3. 都需要不断更新和升级以适应不断变化的计算机技术和需求。

4.RISC-V中的SBI的含义和功能是啥？

SBI（Supervisor Binary Interface）是RISC-V架构中的一种标准接口，用于在特权模式下进行系统调用和访问系统资源。

在RISC-V架构中，SBI由两部分组成：SBI库和SBI固件。SBI库是一个软件库，它提供了一组标准的接口函数，应用程序可以通过这些函数来访问特权模式下的系统资源和服务。SBI固件则是一个运行在特权模式下的软件，它提供了对SBI库中定义的接口函数的实现。

SBI的功能包括以下几个方面：

1. 系统调用：应用程序可以通过SBI库中的接口函数向操作系统发起系统调用，以访问特权模式下的系统资源和服务，例如文件系统、网络、设备驱动程序等。

2. 中断处理：SBI可以处理来自硬件设备和软件异常的中断，并将控制权传递给操作系统进行处理。

3. 堆栈管理：SBI可以管理特权模式下的堆栈，以确保堆栈的正确性和安全性。

4. 内存管理：SBI可以提供内存映射和访问控制等功能，以确保特权模式下的内存访问的正确性和安全性。

5.为了让应用程序能在计算机上执行，操作系统与编译器之间需要达成哪些协议？

数据类型和内存布局

数据类型大小与对齐方式：定义了各种基本数据类型（如int、float等）的大小、内存对齐规则以及结构体的内部布局。这确保了编译器生成的代码在内存操作上与操作系统的预期一致。

函数调用约定

参数传递与返回值：明确规定函数调用时，参数是通过寄存器还是堆栈传递，返回值存放的位置，以及调用者与被调用者需要保存的寄存器。这一约定确保了函数调用和返回过程中数据传递的正确性。

系统调用接口

调用系统服务：操作系统提供的系统调用接口（如文件操作、进程管理、内存分配等）必须被编译器所识别和支持。编译器需要生成符合操作系统要求的调用指令，以便正确访问系统资源。

可执行文件格式

目标文件和可执行文件格式：编译器生成的目标文件和最终可执行文件需要遵循操作系统支持的文件格式，如ELF（在Unix/Linux系统中）、PE（在Windows系统中）或Mach-O（在macOS系统中）。这关系到文件的加载、链接以及运行时的内存映射。

动态链接和共享库

符号解析与链接方式：在使用共享库或动态链接库时，操作系统和编译器需要协商如何进行符号解析、库加载以及运行时链接，确保动态加载的代码能够正确地与主程序交互。

异常处理机制

错误与中断处理：两者需要在异常（如程序崩溃、系统中断等）处理方式上达成一致，这包括信号处理、异常栈展开以及资源清理机制等。

6.为何应用程序员编写应用时不需要建立栈空间和指定地址空间？

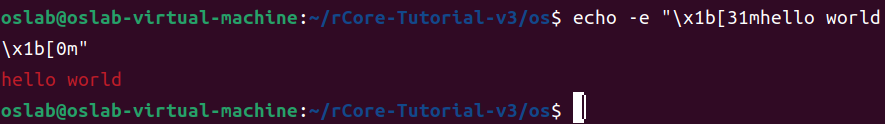
应用程度对内存的访问需要通过 MMU 的地址翻译完成，应用程序运行时看到的地址和实际位于内存中的地址是不同的，栈空间和地址空间需要内核进行管理和分配。应用程序的栈指针在 trap return 过程中初始化。

此外，应用程序可能需要动态加载某些库的内容，也需要内核完成映射.

实践作业：

彩色化log：

我们需要改变输出的hello world的颜色。实验之前我们使用指令“echo -e "\x1b[31mhello world\x1b[0m"”

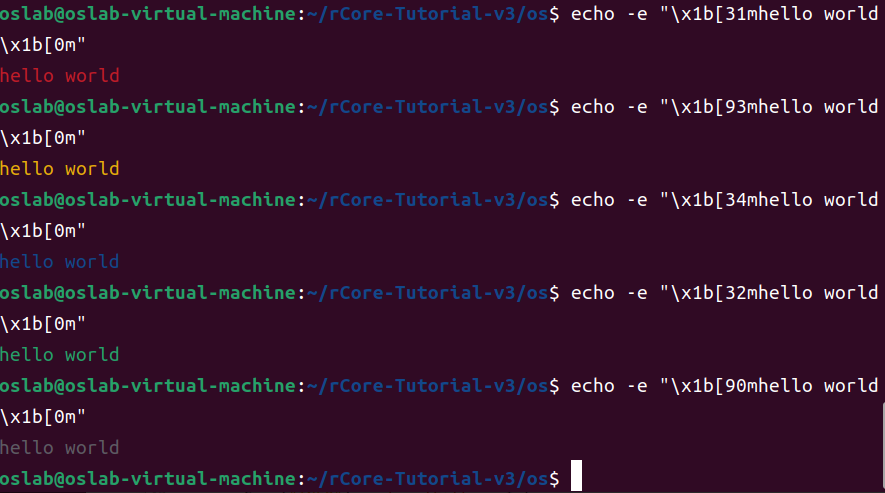


发现输出的hello world的颜色是红色。

我们找到管理颜色的代码部分，src目录下的文件logging。

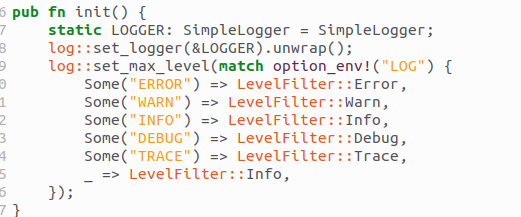


看到color下有不同的颜色对应不同的数字。所以我们找到最简单的修改输出hello world的方式，只要把指令“echo -e "\x1b[31mhello world\x1b[0m"”中的数字“31”改成对应的其他颜色的数字，就可以输出不同颜色hello world：



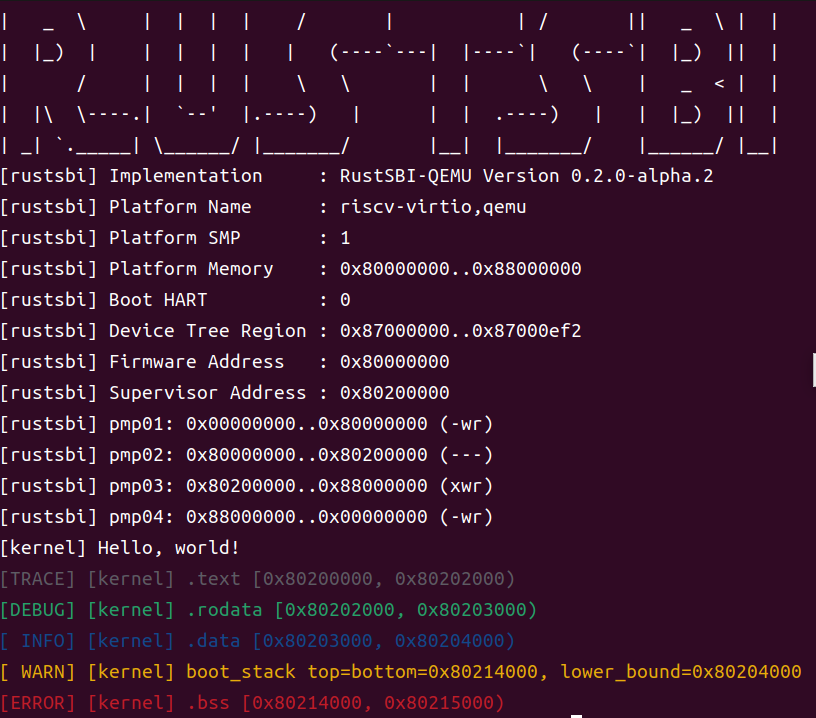
这里的颜色数字是ANSI 颜色码。

要想实现输出优先级，我们要在logging文件中对不同的提示做优先级分级：



看到set\_max\_level函数从上到下设置优先级，比如，日志提示error就只输出error，warn则要同时输出warn和error，即其本身和比本身更高优先级的提示。并且我们设置了每种提示的颜色。

我们设置等级为trace以获得所有提示的输出：



看到其输出了所有的日志提示信息。