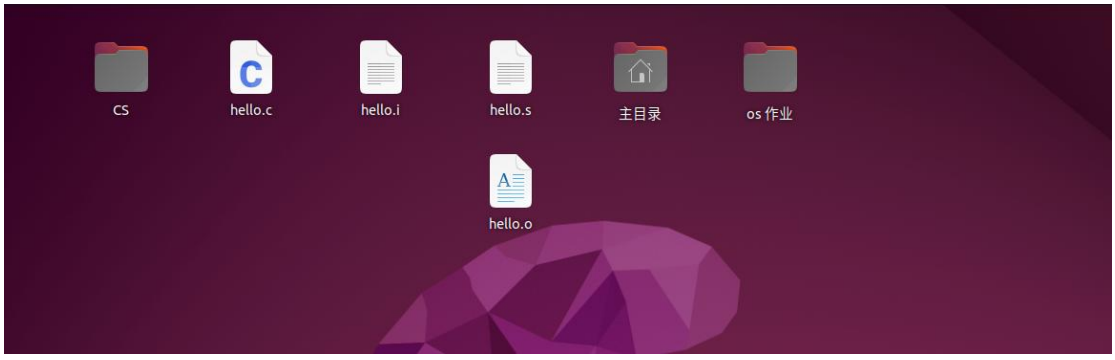
编程作业一：

```c
#include <dirent.h>
#include <stdio.h>
int main() {
DIR *dir = opendir(".");
struct dirent *entry;
while ((entry = readdir(dir)))
{
printf("%s\n", entry->d_name);
}
return 0;
}
```

```
问题  输出  调试控制台  终端  端口

..
sy1_1.c
hello.c
.
os zuoye.c
hello.o
hello.s
sy1_1
CS
hello.i
os 作业
```

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:

```
oslab@oslab-virtual-machine:~/桌面$ gdb sy1_1
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sy1_1...
```

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:

```
(gdb) break main
Breakpoint 1 at 0x1195: file /home/oslab/桌面/sy1_1.c, line 4.
(gdb) break 6
Breakpoint 2 at 0x11a8: file /home/oslab/桌面/sy1_1.c, line 6.
```

Run 程序:

```
(gdb) run
Starting program: /home/oslab/桌面/sy1_1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at /home/oslab/桌面/sy1_1.c:4
4       DIR *dir = opendir(".");
(gdb)
```

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

```
(gdb) backtrace
#0  main () at /home/oslab/桌面/sy1_1.c:4
(gdb)
```

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：

```
(gdb) next

Breakpoint 3, main () at /home/oslab/桌面/sy1_1.c:8
8         printf("%s\n", entry->d_name);
(gdb) next
os zuoye.c
6         while ((entry = readdir(dir)))
(gdb) next

Breakpoint 3, main () at /home/oslab/桌面/sy1_1.c:8
8         printf("%s\n", entry->d_name);
(gdb) next
hello.o
6         while ((entry = readdir(dir)))
(gdb) next

Breakpoint 3, main () at /home/oslab/桌面/sy1_1.c:8
8         printf("%s\n", entry->d_name);
(gdb) next
hello.s
```

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：

```
(gdb)  print entry->d_name
$6 = "os zuoye.c\000\000\000b% \000\000\000\000\000\037\320\027#3\313\031` \000\
bhello.o\000\000\000\000\000\000a% \000\000\000\000\000\065?\035\264\326\022fg \
000\bhello.s\000\000\000\000\000\000\020\v \000\000\000\000\000\257\264\025\354\
251\305,o \000\bsy1_1\000\000\000\000\000\000\000\000\264\327\002\000\000\000\00
0\000\000\344\233\026+\274:p\030\000\004CS\000\000\000`% \000\000\000\000\000\33
0^@\345\237\303Tx \000\bhello.i\000\000\000\000\000\000\030\000\000\000\000\000\
000\000\377\377\377\377\377\377\377\177 \000\004os 作业", '\000' <repeats 62 tim
es>
(gdb) next
os zuoye.c
```

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)：

```c
#include <inttypes.h>
#include <stdint.h>
#include <stdio.h>
#define UNW_LOCAL_ONLY
#include <libunwind.h>
// Compile with -funwind-tables -lunwind
void print_stack_trace_libunwind()
```

提示我们要先安装 libunwind：

```
下列【新】软件包将被安装：
  liblzma-dev libunwind-dev
升级了 0 个软件包，新安装了 2 个软件包，要卸载 0 个软件包，有 0 个软件包未被升级
。
需要下载 2,042 kB 的归档。
解压缩后会消耗 6,755 kB 的额外空间。
您希望继续执行吗？ [Y/n] Y
获取:1 http://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy/main amd64 liblzma-dev a
md64 5.2.5-2ubuntu1 [159 kB]
获取:2 http://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy-updates/main amd64 libun
wind-dev amd64 1.3.2-2build2.1 [1,883 kB]
已下载 2,042 kB，耗时 2秒 (896 kB/s)
正在选中未选择的软件包 liblzma-dev:amd64。
(正在读取数据库 ... 系统当前共安装有 200749 个文件和目录。)
准备解压 .../liblzma-dev_5.2.5-2ubuntu1_amd64.deb  ...
正在解压 liblzma-dev:amd64 (5.2.5-2ubuntu1) ...
正在选中未选择的软件包 libunwind-dev:amd64。
准备解压 .../libunwind-dev_1.3.2-2build2.1_amd64.deb  ...
正在解压 libunwind-dev:amd64 (1.3.2-2build2.1) ...
正在设置 liblzma-dev:amd64 (5.2.5-2ubuntu1) ...
正在设置 libunwind-dev:amd64 (1.3.2-2build2.1) ...
正在处理用于 man-db (2.10.2-1) 的触发器 ...
```

使用 gcc 编译文件：

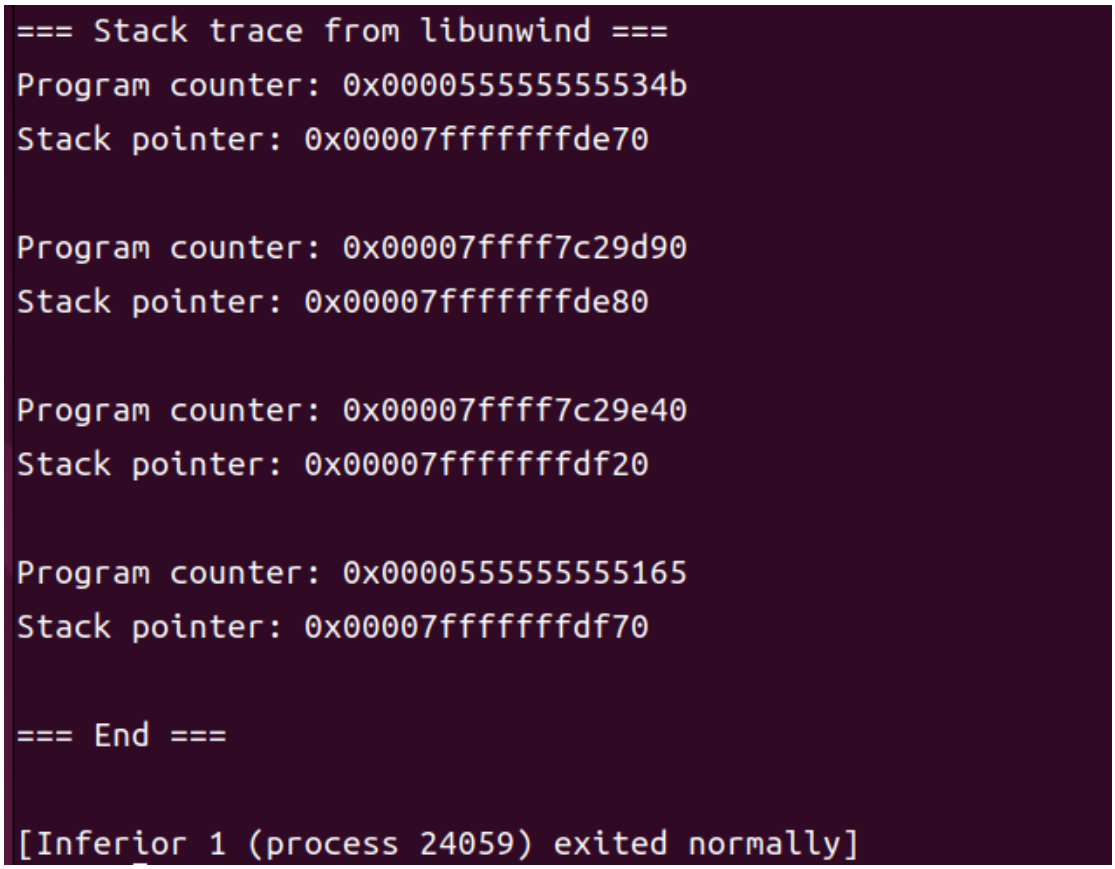
```
compilation terminated.
oslab@oslab-virtual-machine:~/桌面$ gcc -g sy1_2.c -o sy1_2
/usr/bin/ld: /tmp/ccvRlZLQ.o: in function `print_stack_trace_libunwind':
/home/oslab/桌面/sy1_2.c:12: undefined reference to `_Ux86_64_getcontext'
/usr/bin/ld: /home/oslab/桌面/sy1_2.c:13: undefined reference to `_ULx86_64_init
_local'
/usr/bin/ld: /home/oslab/桌面/sy1_2.c:16: undefined reference to `_ULx86_64_get_
reg'
/usr/bin/ld: /home/oslab/桌面/sy1_2.c:17: undefined reference to `_ULx86_64_get_
g'
/usr/bin/ld: /home/oslab/桌面/sy1_2.c:14: undefined reference to `_ULx86_64_step
'
collect2: error: ld returned 1 exit status
oslab@oslab-virtual-machine:~/桌面$ sudo apt-get install libunwind-dev
```

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:

```
oslab@oslab-virtual-machine:~/桌面$ gcc -g -funwind-tables sy1_2.c -o sy1_2 -lun
wind
oslab@oslab-virtual-machine:~/桌面$ gdb sy1_2
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sy1_2...
```

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:

```
(gdb) break main
Breakpoint 1 at 0x1341: file sy1_2.c, line 26.
(gdb) run
Starting program: /home/oslab/桌面/sy1_2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libth

Breakpoint 1, main () at sy1_2.c:26
26          print_stack_trace_libunwind();
(gdb) step
print_stack_trace_libunwind () at sy1_2.c:8
8       {
(gdb) next
9           printf("=== Stack trace from libunwind ===\n");
(gdb) next
=== Stack trace from libunwind ===
12          unw_getcontext(&uc);
(gdb) next
13          unw_init_local(&cursor, &uc);
(gdb) print &uc
$1 = (unw_context_t *) 0x7fffffffd690
(gdb)
```

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：

```
(gdb) print uc
$2 = {uc_flags = 0, uc_link = 0x0, uc_stack = {ss_sp = 0x7fff00000000,
    ss_flags = 0, ss_size = 0}, uc_mcontext = {gregs = {0, 93824992252576,
      140737352147320, -8, 140737488347016, 93824992236345, 93824992247160,
      140737354125376, 140737488344720, 1, 140737488346720, 0, 1,
      140737488344720, 140737351075975, 140737488344704, 93824992236133, 0,
      4096, 140737350563008, 140737488345856, 31, 0},
    fpregs = 0xffff0000ffff037f, __reserved1 = {4294967295, 0, 8064, 0,
      281470681751424, 0, 0, 0}}, uc_sigmask = {__val = {0, 0, 0, 0, 0, 0, 0,
      0, 140737353933692, 140737353856640, 140737349962448, 1736415839,
      140737349952548, 0, 18446742974197923840, 18446744073709551615}},
  __fpregs_mem = {cwd = 55152, swd = 65535, ftw = 32767, fop = 0,
    rip = 5931345451726490695, rdp = 6872211854156977737, mxcsr = 95,
    mxcr_mask = 0, _st = {{significand = {0, 0, 0, 0}, exponent = 24933,
        __glibc_reserved1 = {27762, 24441, 28265}}, {significand = {29801,
          29440, 29300, 27758}, exponent = 3480, __glibc_reserved1 = {0, 0,
          0}}, {significand = {0, 0, 0, 0}, exponent = 96,
        __glibc_reserved1 = {0, 0, 0}}, {significand = {0, 0, 0, 0},
        exponent = 0, __glibc_reserved1 = {0, 0, 0}}, {significand = {0, 0, 0,
          0}, exponent = 0, __glibc_reserved1 = {0, 0, 0}}, {significand = {0,
          0, 0, 0}, exponent = 0, __glibc_reserved1 = {0, 0, 0}}, {
        significand = {0, 0, 0, 0}, exponent = 0, __glibc_reserved1 = {0, 0,
```

里面存着当前程序在这一帧的状态。
再试着打印&cursor:

```
14          while (unw_step(&cursor) > 0)
(gdb) print unw_(&cursor)
No symbol "unw_" in current context.
(gdb) print &cursor
$3 = (unw_cursor_t *) 0x7fffffffda60
```

打印 pc 和 sp 的值:

```
(gdb) next
16              unw_get_reg(&cursor, UNW_REG_IP, &pc);
(gdb) next
17              unw_get_reg(&cursor, UNW_REG_SP, &sp);
(gdb) next
18              printf("Program counter: 0x%016" PRIxPTR "\n", (uintptr_t) pc);
(gdb) print pc
$5 = 93824992236363
(gdb) print sp
$6 = 140737488346736
```

继续执行代码:

```
(gdb) print pc
$5 = 93824992236363
(gdb) print sp
$6 = 140737488346736
(gdb) next
Program counter: 0x000055555555534b
19              printf("Stack pointer: 0x%016" PRIxPTR "\n", (uintptr_t) sp);
(gdb) next
Stack pointer: 0x00007fffffffde70
```

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:

```
$10 = {opaque = {140737488345696, 140737353745152, 140737488346752,
    140737350114704, 0, 140733193388032, 140733193388032, 140733193388032,
    140737488344864, 0, 140737488344856, 0, 140737488344872, 0,
    140737488344848, 0, 140737488344832, 0, 140737488344824, 0,
    140737488346736, 0, 140737488344880, 0, 140737488344760, 0,
    140737488344768, 0, 140737488344776, 0, 140737488344784, 0,
    140737488344792, 0, 140737488344800, 0, 140737488344808, 0,
    140737488344816, 0, 140737488346744, 0, 140737353912386, 93824992236345,
    93824992236370, 0, 0, 93824992247680, 0, 309237645313, 0, 0, 65536,
    3573412791104, 3573412791104, 3573412790272, 0, 3573412790272,
    140737488344720, 3573412791104, 3573412791104, 3573412791104, 0, 256, 0,
    0, 0, 0, 0, 0, 280375465083135, 4278190080, 3399988123389603631,
    3399988123389603631, 3480, 0, 96, 0 <repeats 19 times>, 64, 10, 64, 12, 0,
    4294967296, 0, 661424963590, 8589934595, 0, 0, 0, 2, 9223372036854775814,
    0, 0, 0, 0, 0, 0, 1, 1, 93824992231488, 140737354016828, 3632,
    140737488347881, 140737353879552, 1103823372288, 2, 529267711,
    140737488347897}}
```

```
$11 = {opaque = {140737488345696, 140737353745152, 140737488346992,
    93824992235877, 0, 140733193388032, 140733193388032, 140733193388032,
    140737488344864, 0, 140737488344856, 0, 140737488344872, 0,
    140737488346936, 0, 140737488344832, 0, 140737488344824, 0,
    140737488346944, 0, 140737488344880, 0, 140737488344760, 0,
    140737488344768, 0, 140737488344776, 0, 140737488344784, 0,
    140737488346952, 0, 140737488346960, 0, 140737488346968, 0,
    140737488346976, 0, 140737488346984, 0, 140737353912386, 140737350114752,
    140737350115080, 0, 0, 140737352146944, 0, 309237645313, 0, 0, 196608,
    3573412791104, 3573412791104, 3573412790272, 0, 3573412790272,
    140737488344720, 3573412791104, 3573412791104, 3573412791104, 0, 256, 0,
    0, 0, 0, 0, 0, 280375465083135, 4278190080, 3399988123389603631,
    3399988123389603631, 3480, 0, 96, 0 <repeats 19 times>, 64, 10, 64, 12, 0,
    4294967296, 0, 661424963590, 8589934595, 0, 0, 0, 2, 9223372036854775814,
    0, 0, 0, 0, 0, 0, 1, 1, 93824992231488, 140737354016828, 3632,
    140737488347881, 140737353879552, 1103823372288, 2, 529267711,
    140737488347897}}
```

对比两图，可以发现有些值发生了改变：

`140737488346752` 变为 `140737488346992`

`93824992231488` 变为 `93824992235877`

`140737488344848` 变为 `140737488346936`

`140737488346736` 变为 `140737488346944`

`140737488344792` 变为 `140737488346952`

`140737488344800` 变为 `140737488346960`

`140737488344808` 变为 `140737488346968`

`140737488344816` 变为 `140737488346976`

`140737488346744` 变为 `140737488346984`

`93824992236345` 变为 `140737350114752`

`93824992236370` 变为 `140737350115080`

`93824992247680` 变为 `140737352146944`

`65536` 变为 `196608`

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：

```
oslab@oslab-virtual-machine:~/桌面$ readelf -l sy1_1

Elf 文件类型为 DYN (Position-Independent Executable file)
Entry point 0x10a0
There are 13 program headers, starting at offset 64

程序头:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000698 0x0000000000000698  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x00000000000001e5 0x00000000000001e5  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000000ec 0x00000000000000ec  R      0x1000
  LOAD           0x0000000000002da8 0x0000000000003da8 0x0000000000003da8
                 0x0000000000000268 0x0000000000000270  RW     0x1000
  DYNAMIC        0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
```

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。

```
oslab@oslab-virtual-machine:~/桌面$ ps -ef | grep sy1_1
oslab       6248    6227  0 17:26 pts/0    00:00:00 gdb sy1_1
oslab       7101    6248  0 17:34 pts/0    00:00:00 /home/oslab/桌面/sy1_1
oslab       7169    6265  0 17:35 pts/1    00:00:00 grep --color=auto sy1_1
```

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”。

```
5af44ec92000-5af44ec9d000 rw-p 00000000 00:00 0
5af4560c8000-5af45622f000 rw-p 00000000 00:00 0                          [heap]
```

```
7fffffde000-7ffffffff000 rw-p 00000000 00:00 0                          [stack]
```

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：0x7fffffde000-0x7fffffffff000.

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"”

```
oslab@oslab-virtual-machine:~/rCore-Tutorial-v3/os$ echo -e "\x1b[31mhello world
\x1b[0m"
hello world
oslab@oslab-virtual-machine:~/rCore-Tutorial-v3/os$
```

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。

```rust
use log::{self, Level, LevelFilter, Log, Metadata, Record};

struct SimpleLogger;

impl Log for SimpleLogger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }
    fn log(&self, record: &Record) {
        if !self.enabled(record.metadata()) {
            return;
        }
        let color = match record.level() {
            Level::Error => 31, // Red
            Level::Warn => 93,  // BrightYellow
            Level::Info => 34,  // Blue
            Level::Debug => 32, // Green
            Level::Trace => 90, // BrightBlack
        };
        println!(
            "\u{1B}[{}m[{:>5}] {}\u{1B}[0m",
            color,
            record.level(),
            record.args(),
        );
    }
    fn flush(&self) {}
}
```

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：

```
oslab@oslab-virtual-machine:~/rCore-Tutorial-v3/os$ echo -e "\x1b[31mhello world
\x1b[0m"
hello world
oslab@oslab-virtual-machine:~/rCore-Tutorial-v3/os$ echo -e "\x1b[93mhello world
\x1b[0m"
hello world
oslab@oslab-virtual-machine:~/rCore-Tutorial-v3/os$ echo -e "\x1b[34mhello world
\x1b[0m"
hello world
oslab@oslab-virtual-machine:~/rCore-Tutorial-v3/os$ echo -e "\x1b[32mhello world
\x1b[0m"
hello world
oslab@oslab-virtual-machine:~/rCore-Tutorial-v3/os$ echo -e "\x1b[90mhello world
\x1b[0m"
hello world
oslab@oslab-virtual-machine:~/rCore-Tutorial-v3/os$
```

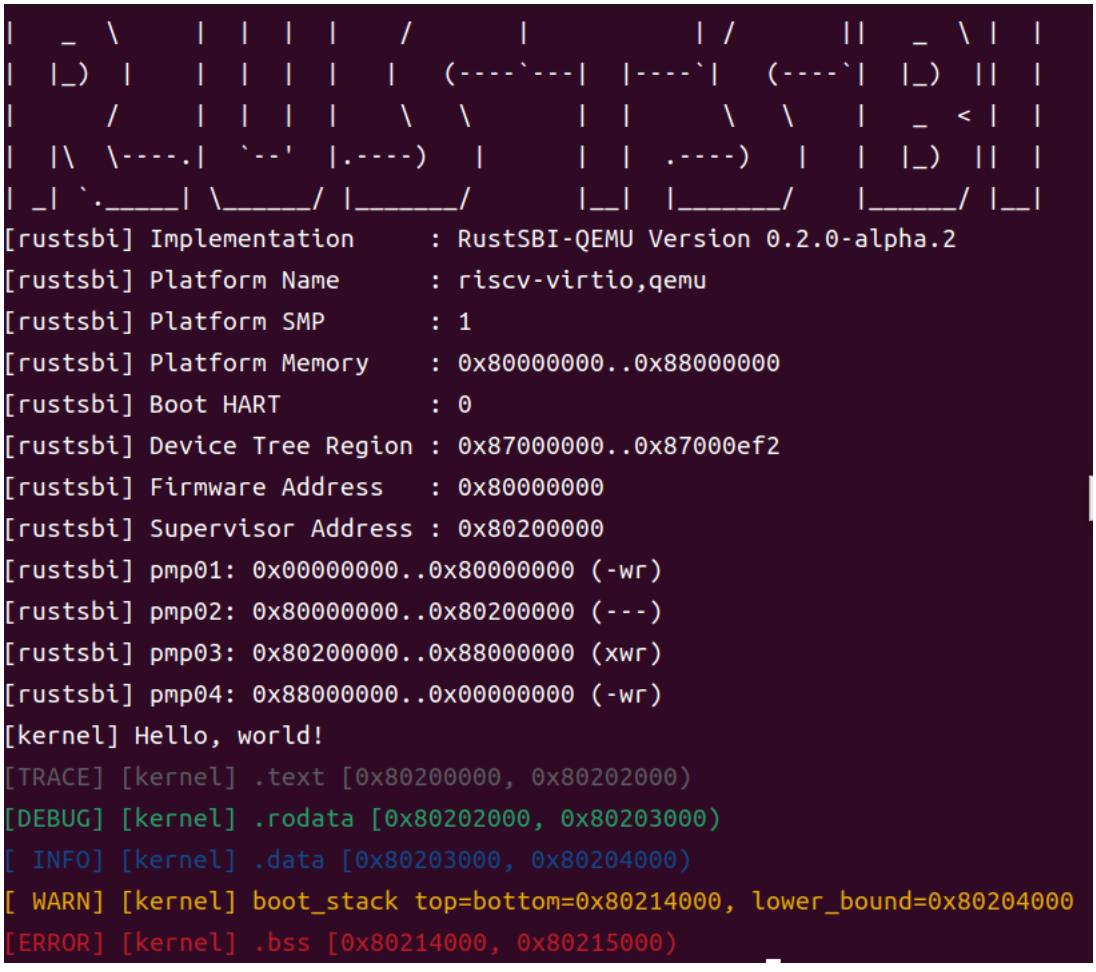
这里的颜色数字是 ANSI 颜色码。

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：

```rust
pub fn init() {
    static LOGGER: SimpleLogger = SimpleLogger;
    log::set_logger(&LOGGER).unwrap();
    log::set_max_level(match option_env!("LOG") {
        Some("ERROR") => LevelFilter::Error,
        Some("WARN") => LevelFilter::Warn,
        Some("INFO") => LevelFilter::Info,
        Some("DEBUG") => LevelFilter::Debug,
        Some("TRACE") => LevelFilter::Trace,
        _ => LevelFilter::Info,
    });
}
```

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。