

课程报告四

杜培绪

2024 年 9 月 15 日

题目 1.

使用 Linux 上的 `journalctl` 或命令来获取最近一天中超级用户的登录信息及其所执行的指令。

解决方法：

在 Linux 系统中输入命令 `journalctl`，显示出最近一天的用户登录信息和执行过的指令。

```
a@ubuntu:~$ journalctl
-- Logs begin at Thu 2024-09-05 19:06:17 PDT, end at Fri 2024-09-13 21:04:24 PDT.
9月 05 19:06:17 ubuntu kernel: Linux version 5.4.0-150-generic (build@bos03-amd
9月 05 19:06:17 ubuntu kernel: Command line: BOOT_IMAGE=/boot/vmlinuz-5.4.0-150-
9月 05 19:06:17 ubuntu kernel: KERNEL supported cpus:
9月 05 19:06:17 ubuntu kernel: Intel GenuineIntel
9月 05 19:06:17 ubuntu kernel: AMD AuthenticAMD
9月 05 19:06:17 ubuntu kernel: Hygon HygonGenuine
9月 05 19:06:17 ubuntu kernel: Centaur CentaurHauls
9月 05 19:06:17 ubuntu kernel: zhaoxin Shanghai
9月 05 19:06:17 ubuntu kernel: x86/fpu: Supporting XSAVE feature 0x001: 'x87 flo
9月 05 19:06:17 ubuntu kernel: x86/fpu: Supporting XSAVE feature 0x002: 'SSE reg
9月 05 19:06:17 ubuntu kernel: x86/fpu: Supporting XSAVE feature 0x004: 'AVX reg
9月 05 19:06:17 ubuntu kernel: x86/fpu: Supporting XSAVE feature 0x200: 'Protect
9月 05 19:06:17 ubuntu kernel: x86/fpu: xstate_offset[2]: 576, xstate_sizes[2]:
9月 05 19:06:17 ubuntu kernel: x86/fpu: xstate_offset[0]: 832, xstate_sizes[0]:
```

图 1: 执行过的指令

题目 2.

学习使用 `pdb` 调试 python 代码。

解决方法：

在代码中添加一行 pdb 对应代码，制造断点，代码如下：当代码允

```
def fact(x):  
    if x == 0:  
        return 1  
    return x * fact(x - 1)  
x=int(input())  
import pdb; pdb.set_trace()  
print (fact(x))
```

图 2: pdb 代码

许到此处时，程序将进入暂停状态，并输出接下来要输出的内容。我们可以在此处输入其他命令进行操作。例如 n 表示继续进行，s 表示允许这一行并在下一处可能的位置停止，l 表示列出当前文件源代码，q 表示退出等。

```
a@ubuntu:~$ python3 3.py  
6  
> /home/a/3.py(7)<module>()  
-> print (fact(x))  
(Pdb) s  
--Call--  
> /home/a/3.py(1)fact()  
-> def fact(x):  
(Pdb) s  
> /home/a/3.py(2)fact()  
-> if x == 0:  
(Pdb) s  
> /home/a/3.py(4)fact()  
-> return x * fact(x - 1)  
(Pdb) n  
--Return--  
> /home/a/3.py(4)fact()->720  
-> return x * fact(x - 1)
```

图 3: pdb 的应用

题目 3.

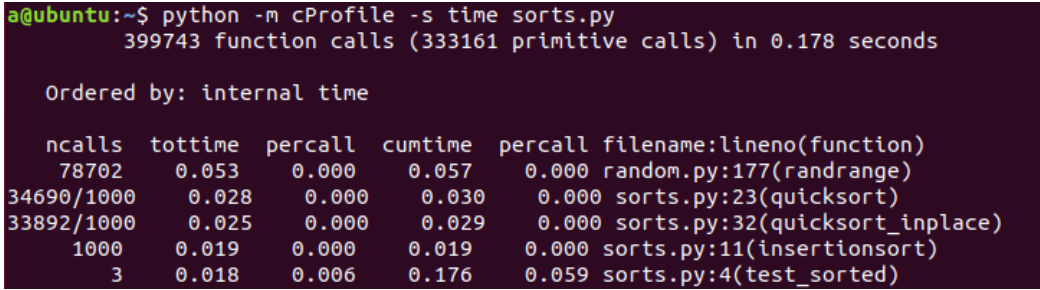
使用 cProfile 来比较插入排序和快速排序的性能。

解决方法:

在 linux 中安装相关工具，使用以下代码运行题目中的排序算法。

```
python -m cProfile -s time sorts.py
```

运行结果如下：可以发现，插入算法的效率最高。



```
a@ubuntu:~$ python -m cProfile -s time sorts.py
399743 function calls (333161 primitive calls) in 0.178 seconds

Ordered by: internal time
```

| ncalls | totttime | percall | cumtime | percall | filename:lineno(function) |
|------------|----------|---------|---------|---------|---------------------------------|
| 78702 | 0.053 | 0.000 | 0.057 | 0.000 | random.py:177(randrange) |
| 34690/1000 | 0.028 | 0.000 | 0.030 | 0.000 | sorts.py:23(quick sort) |
| 33892/1000 | 0.025 | 0.000 | 0.029 | 0.000 | sorts.py:32(quick sort_inplace) |
| 1000 | 0.019 | 0.000 | 0.019 | 0.000 | sorts.py:11(insertionsort) |
| 3 | 0.018 | 0.006 | 0.176 | 0.059 | sorts.py:4(test_sorted) |

图 4: cProfile 比较算法性能

题目 4.

使用 line_profiler 来比较插入排序和快速排序的性能。

解决方法:

首先需要安装 line_profiler 工具，使用 line_profiler 工具比较性能需要在代码中相应位置插入装饰器

```
@profile
```

想要测试文件中单个排序算法的性能，需要在该函数上方加入装饰器。想要测试插入算法的性能，代码如图所示：

```
@profile
def insertionsort(array):

    for i in range(len(array)):
        j = i-1
        v = array[i]
```

图 5: line_profiler 测试插入算法

随后使用代码

```
kernprof -l -v sorts.py
```

运行 python 程序，即可查看插入算法的性能情况。修改装饰器位置，就可

```
Wrote profile results to sorts.py.lprof
Timer unit: 1e-06 s

Total time: 0.201273 s
File: sorts.py
Function: insertionsort at line 10
```

| Line # | Hits | Time | Per Hit | % Time | Line Contents |
|--------|--------|---------|---------|--------|-----------------------------|
| 10 | | | | | @profile |
| 11 | | | | | def insertionsort(array): |
| 12 | | | | | |
| 13 | 25978 | 6586.0 | 0.3 | 3.3 | for i in range(len(array)): |
| 14 | 24978 | 6297.0 | 0.3 | 3.1 | j = i-1 |
| 15 | 24978 | 6478.0 | 0.3 | 3.2 | v = array[i] |
| 16 | 227751 | 66245.0 | 0.3 | 32.9 | while j >= 0 and v < ar |
| 17 | 202773 | 57803.0 | 0.3 | 28.7 | array[j+1] = array[|
| 18 | 202773 | 50489.0 | 0.2 | 25.1 | j -= 1 |
| 19 | 24978 | 7109.0 | 0.3 | 3.5 | array[j+1] = v |
| 20 | 1000 | 266.0 | 0.3 | 0.1 | return array |

图 6: 插入算法性能

以查看并比较所有排序算法的性能了。

题目 5.

使用 `memory_profiler` 来检查排序算法的内存消耗。

解决方法:

与 `line_profiler` 相似，使用 `memory_profiler` 也需要插入装饰器位置。之后使用代码

```
python -m memory_profiler sorts.py
```

内存消耗如下:

| Filename: sorts.py | | | |
|--------------------|------------|------------|--------------------------------|
| Line # | Mem usage | Increment | Line Contents |
| ===== | | | |
| 10 | 14.023 MiB | 14.023 MiB | @profile |
| 11 | | | def insertionsort(array): |
| 12 | | | |
| 13 | 14.023 MiB | 0.000 MiB | for i in range(len(array)): |
| 14 | 14.023 MiB | 0.000 MiB | j = i-1 |
| 15 | 14.023 MiB | 0.000 MiB | v = array[i] |
| 16 | 14.023 MiB | 0.000 MiB | while j >= 0 and v < array[j]: |
| 17 | 14.023 MiB | 0.000 MiB | array[j+1] = array[j] |
| 18 | 14.023 MiB | 0.000 MiB | j -= 1 |
| 19 | 14.023 MiB | 0.000 MiB | array[j+1] = v |
| 20 | 14.023 MiB | 0.000 MiB | return array |

图 7: 内存消耗

题目 6.

使用 `perf` 来查看不同排序算法的循环次数及缓存命中及丢失情况。

解决方法:

安装 `perf` 工具后，修改排序代码，将 `for` 循环删去，改为只运行一种算法。修改后代码如下:

```
if __name__ == '__main__':

    test_sorted(quicksort_inplace)
```

图 8: 修改后代码

由于 Linux 使用的是虚拟机，perf 下一些命令无法执行，显示为 “no supported”。在网上搜索信息尝试修改，但未能修复这一问题。使用 perf 命令返回如下信息：

```
a@ubuntu:~$ sudo perf stat -e cycles,cache-references,cache-misses python3 sorts.py

Performance counter stats for 'python3 sorts.py':

   <not supported>      cycles
   <not supported>      cache-references
   <not supported>      cache-misses

    0.057887908 seconds time elapsed

    0.053237000 seconds user
    0.004095000 seconds sys
```

图 9: perf 运行结果

题目 7.

根据给出的斐波那契数列代码，使用 pycallgraph 和 graphviz 生成图片，显示出程序的执行信息。

解决方法：

输入命令行

```
pip install "setuptools < 58.0.0"
```

`pip install pycallgraph`

安装工具，之后输入代码运行程序 运行完毕后会发现目录下多出来一张图

```
a@ubuntu:~$ pycallgraph graphviz -- ./fib.py
34
```

图 10: 运行结果

片，显示出了程序的运行情况。

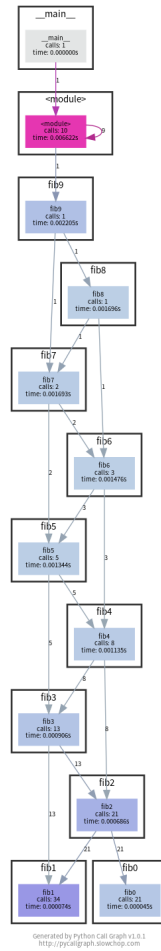


图 11: pycallgraph.png

题目 8.

在 Linux 系统中修改键盘映射，把 Caps Lock 键更换为 Shift L。

解决方法：

使用 xmodmap 修改键盘映射，执行命令后可以查看当前的键盘映射，如图所示：

```
agubuntu:~$ xmodmap -pke
keycode 8 =
keycode 9 = Escape NoSymbol Escape NoSymbol Escape
keycode 10 = 1 exclam 1 exclam 1 exclam
keycode 11 = 2 at 2 at 2 at
keycode 12 = 3 numbersign 3 numbersign 3 numbersign
keycode 13 = 4 dollar 4 dollar 4 dollar
keycode 14 = 5 percent 5 percent 5 percent
keycode 15 = 6 asciicircum 6 asciicircum 6 asciicircum
keycode 16 = 7 ampersand 7 ampersand 7 ampersand
keycode 17 = 8 asterisk 8 asterisk 8 asterisk
```

图 12: 查看键盘映射

接下来打开配置文件.Xmodmap，并在其中加入代码如下：

```
keycode 254 = XF86WWAN NoSymbol XF86WWAN
keycode 255 = XF86RFKill NoSymbol XF86RFKill
keycode 66 = Shift_L NoSymbol Shift_L
```

图 13: 修改键盘映射

随后使用命令

```
xmodmap ~/.Xmodmap
```

重新加载配置文件，之后尝试使用 caps lock 键进行输入，可以看到已经可以发挥与 shift 键相同的效果。

题目 9.

学习 Linux 中的守护进程，查看当前进程并尝试创建一个.service 配置文件。

解决方法：

运行命令行 `systemctl status` 查看正在运行的所有守护进程：

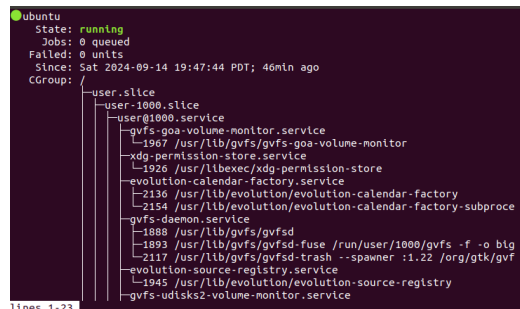


图 14: 守护进程

接下来在相应位置创建.service 文件。如果直接用 vim 创建，会显示权限不足，需要先切换到 root 权限。这一步需要重新设置密码。之后使用 `su` 命令获取 root 权限。

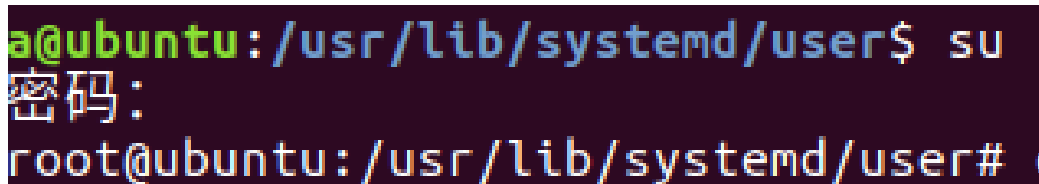


图 15: 获取 root 权限

随后即可创建配置文件并保存。

```
# 配置文件描述
Description=My Custom App
# 在网络服务启动后启动该进程
After=network.target

[Service]
# 运行该进程的用户
User=foo
# 运行该进程的用户组
Group=foo
# 运行该进程的根目录
WorkingDirectory=/home/foo/projects/mydaemon
# 开始该进程的命令
ExecStart=/usr/bin/local/python3.7 app.py
# 在出现错误时重启该进程
Restart=on-failure

[Install]
# 相当于Windows的开机启动。即使GUI没有启动，该进程也会加载并运行
WantedBy=multi-user.target
# 如果该进程仅需要在GUI活动时运行，这里应写作：
# WantedBy=graphical.target
# graphical.target在multi-user.target的基础上运行和GUI相关的服务
```

图 16: 配置文件

题目 10.

nm 命令可以列出一些特定类型文件的符号表，学习使用 nm 调试代码。

解决方法：

nm 命令的对象为库文件、可执行文件、目标文件，首先生成一个.o 目标文件。首先创建一个 C 语言文件，用 g++ 编译后生成目标文件。

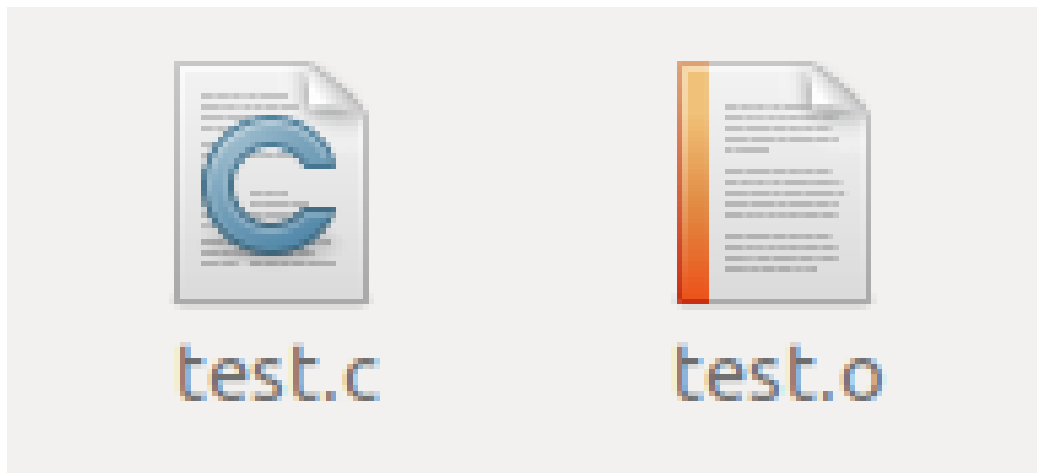


图 17: 目标文件的生成

之后对目标文件使用 nm 命令，即可查看该文件的符号信息。

```

a@ubuntu:~$ nm -n test.o
                 U _GLOBAL_OFFSET_TABLE_
                 U putchar
0000000000000000 R ch
0000000000000000 T function
0000000000000000 D init
0000000000000004 d sta_int.2253
0000000000000004 C uninit

```

图 18: 符号信息

题目 11.

addr2line 是一种常用的调试工具，可以通过地址迅速找到崩溃的位置。学习使用 addr2line 调试程序。

解决方法:

首先编写 C++ 程序如下:

```

#include <stdio.h>

int divide(int a, int b)
{
    return a/b;
}

int main()
{
    fprintf(stdout, "input value\n");
    int a = 3, b = 0;
    int div = divide(a, b);
    fprintf(stdout, "div value: %d\n", div);

    return 0;
}

```

图 19: C++ 程序

运行程序后，使用 dmesg 命令查找错误信息。

可以看到 ip 后面的数字和字母 555f409946d8 就是出错的位置。我们可以通过 addr2line 命令查找代码中的错误位置。

直接查看时会发现只能输出问号，而不是像预期那样标出错误行数。经过搜索资料，了解到是因为输入的地址并非相对偏移地址，减去基地址后即可正常执行。

```

a@ubuntu:~$ dmesg | grep main
[ 2.153029] PCI: MMCONFIG for domain 0000 [bus 00-7f] at [mem 0xf0000000-0xf7
ffff] (base 0xf0000000)
[ 2.523324] ACPI: PCI Root Bridge [PCI0] (domain 0000 [bus 00-7f])
[ 4.991318] iommu: Default domain type: Translated
[ 5.181226] NetLabel: domain hash size = 128
[ 7.074621] platform elsa.0: EISA: Cannot allocate resource for mainboard
[ 8.351624] scsi target32:0:0: Beginning Domain Validation
[ 8.359827] scsi target32:0:0: Domain Validation skipping write tests
[ 8.359828] scsi target32:0:0: Ending Domain Validation
[ 9277.476282] traps: main[3259] trap divide error ip:555f409946d8 sp:7ffdf106a
0 error:0 in main[555f40994000+1000]

```

图 20: dmesg 命令

```

a@ubuntu:~$ addr2line -e main 555f409946d8 -f -a -p -C
0x0000555f409946d8: ?? ??:0

```

图 21: 运行结果

题目 12.

strace 可以跟踪系统调用和信号，能够帮助开发者快速定位问题和优化程序。使用 strace 工具跟踪一些常见的简单命令。

解决方法:

strace 可以跟踪系统中的命令，用这种方式我们可以看到在执行这个命令时系统进行了哪些步骤，可以让开发者了解这些命令的执行情况。例如用 strace 跟踪 cat 命令:

```

a@ubuntu:~$ strace cat 1.txt
execve("/bin/cat", ["cat", "1.txt"], 0x7ffdc9985198 /* 66 vars */) = 0
brk(NULL)                               = 0x55d0a0dd2000
access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=146824, ...}) = 0
mmap(NULL, 146824, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f27b7a15000
close(3)                                 = 0
access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\240\35\2\0\0\0\0\0"...
832) = 832

```

图 22: 跟踪 cat 命令

还可以跟踪许多命令，如复制命令。

```

a@ubuntu:~$ strace cp file1.txt file2.txt
execve("/bin/cp", ["cp", "file1.txt", "file2.txt"], 0x7ffe1dc7f4c0 /* 66 vars */) = 0
brk(NULL)                                = 0x557f9ab38000
access("/etc/ld.so.nohwcap", F_OK)        = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)        = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=146824, ...}) = 0
mmap(NULL, 146824, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff77753f000
close(3)                                  = 0
access("/etc/ld.so.nohwcap", F_OK)        = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3

```

图 23: 跟踪 cp 命令

题目 13. 学习使用 `stare` 命令跟踪文件操作。

解决方法:

可以使用 `stace` 跟踪 `cat` 命令对文件的操作情况。修改参数, 使用如下代码:

```
strace -e trace = file cat 1.txt
```

这样可以之跟踪与文件有关的信息, 过滤到无关信息。结果如下:

```

a@ubuntu:~$ strace -e trace=file cat 1.txt
execve("/bin/cat", ["cat", "1.txt"], 0x7ffc975751f8 /* 66 vars */) = 0
access("/etc/ld.so.nohwcap", F_OK)        = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)        = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
access("/etc/ld.so.nohwcap", F_OK)        = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "1.txt", O_RDONLY)       = 3
Hello, World!
+++ exited with 0 +++

```

图 24: 跟踪 cat 对文件的操作

`strace` 还可以监控进程和网络活动。使用代码如下:

```
strace -p id
```

```
strace -e trace = network curl http://example.com
```

题目 14.

学习使用 lsof 命令。

解决方法:

lsof 命令可以列出当前打开的所有文件信息，可以帮助用户了解资源占用情况和文件信息。

```
790022 /home/a
lsof 5208 a rtd DIR 8,1 4096
2 /
lsof 5208 a txt REG 8,1 163224
918158 /usr/bin/lsof
lsof 5208 a mem REG 8,1 4799968
945898 /usr/lib/locale/locale-archive
lsof 5208 a mem REG 8,1 144976
425204 /lib/x86_64-linux-gnu/libpthread-2.27.so
lsof 5208 a mem REG 8,1 14560
425167 /lib/x86_64-linux-gnu/libdl-2.27.so
lsof 5208 a mem REG 8,1 460728
405613 /lib/x86_64-linux-gnu/libpcr.so.3.13.3
lsof 5208 a mem REG 8,1 2030928
425161 /lib/x86_64-linux-gnu/libc-2.27.so
lsof 5208 a mem REG 8,1 154832
399743 /lib/x86_64-linux-gnu/libselinux.so.1
lsof 5208 a mem REG 8,1 179152
393241 /lib/x86_64-linux-gnu/ld-2.27.so
lsof 5208 a 4r FIFO 0,13 0t0
173705 pipe
lsof 5208 a 7w FIFO 0,13 0t0
```

图 25: lsof 列表

lsof 命令会输出很多信息，支持输入参数进行分页展示。输入

```
lsof | more
```

命令，显示以下界面：

```
COMMAND  PID  TID      USER  FD  TYPE      DEVICE SIZE/OFF
NODE NAME
systemd   1      root  cwd  unknown
/proc/1/cwd (readlink: Permission denied)
systemd   1      root  rtd  unknown
/proc/1/root (readlink: Permission denied)
systemd   1      root  txt  unknown
/proc/1/exe (readlink: Permission denied)
systemd   1      root  NOFD
/proc/1/fd (opendir: Permission denied)
kthreadd  2      root  cwd  unknown
/proc/2/cwd (readlink: Permission denied)
kthreadd  2      root  rtd  unknown
/proc/2/root (readlink: Permission denied)
kthreadd  2      root  txt  unknown
/proc/2/exe (readlink: Permission denied)
kthreadd  2      root  NOFD
/proc/2/fd (opendir: Permission denied)
rcu_gp    3      root  cwd  unknown
/proc/3/cwd (readlink: Permission denied)
rcu_gp    3      root  rtd  unknown
/proc/3/root (readlink: Permission denied)
rcu_gp    3      root  txt  unknown
--更多--
```

图 26: 分页展示

题目 15.

了解 DWARF 调试文件格式的相关内容。

解决方法:

DWARF 是一个用于在可执行程序和其源代码之间进行关联的调试文件格式，当编译时，系统会生成 dwarf 格式的调试信息。编译后使用 readelf 查看 dwarf 信息，如下图所示：这种格式包含关于源代码结构的很多信息，

```
a@ubuntu:~$ readelf --debug-dump=info main
.debug_info 节的内容:

编译单元 @ 偏移 0x0:
  长度:          0x38f (32-bit)
  版本:          4
  缩写偏移量:    0x0
  指针大小:      8
<0><b>: 缩写编号: 1 (DW_TAG_compile_unit)
  <c> DW_AT_producer : (indirect string, offset: 0xef): GNU C++14 7.5.0 -
mtune=generic -march=x86-64 -g -fstack-protector-strong
  <10> DW_AT_language : 4 (C++)
  <11> DW_AT_name : (indirect string, offset: 0x265): main.cpp
  <15> DW_AT_comp_dir : (indirect string, offset: 0x6e): /home/a
  <19> DW_AT_low_pc : 0x6ca
  <21> DW_AT_high_pc : 0x80
  <29> DW_AT_stmt_list : 0x0
<1><2d>: 缩写编号: 2 (DW_TAG_typedef)
  <2e> DW_AT_name : (indirect string, offset: 0x2f): size_t
  <32> DW_AT_decl_file : 2
  <33> DW_AT_decl_line : 216
```

图 27: dwarf 调试信息

比如变量、函数等，还会与其在源代码中的位置进行关联。

题目 16.

学习使用 proc 查看系统中的进程状态。

解决方法:

proc 会收集系统启动后运行时的系统信息，使用 proc 命令就可以直接进行查看。

```
a@ubuntu:~$ ls /proc
```

| | | | | | | | | | |
|------|------|------|------|------|------|------|-----|-----|---------------|
| 1 | 1277 | 17 | 20 | 220 | 253 | 3671 | 6 | 950 | mdstat |
| 10 | 128 | 1702 | 2000 | 2200 | 254 | 375 | 704 | 955 | meminfo |
| 100 | 1281 | 1741 | 2007 | 221 | 255 | 379 | 710 | 956 | misc |
| 101 | 1282 | 1744 | 2008 | 2212 | 256 | 381 | 712 | 96 | modules |
| 1018 | 129 | 1745 | 2010 | 2214 | 2565 | 382 | 716 | 97 | mounts |
| 102 | 130 | 1763 | 2025 | 222 | 257 | 383 | 719 | 98 | mpt |
| 1020 | 1319 | 1768 | 2028 | 223 | 258 | 385 | 725 | 99 | mtrr |
| 1027 | 132 | 1770 | 2031 | 224 | 2585 | 388 | 729 | 992 | net |
| 103 | 1322 | 1776 | 2033 | 225 | 259 | 392 | 730 | | pagetypeinfo |
| 104 | 1325 | 1779 | 2035 | 226 | 26 | 4 | 733 | | partitions |
| 105 | 1327 | 18 | 2037 | 227 | 260 | 404 | 770 | | pressure |
| 106 | 134 | 1856 | 2039 | 228 | 261 | 407 | 772 | | sched_debug |
| 107 | 1392 | 1858 | 2040 | 229 | 262 | 4095 | 78 | | schedstat |
| 1075 | 14 | 1863 | 207 | 23 | 263 | 412 | 780 | | scsi |
| 108 | 143 | 1865 | 2071 | 230 | 264 | 419 | 784 | | self |
| 109 | 146 | 1882 | 208 | 231 | 265 | 425 | 785 | | slabinfo |
| 11 | 15 | 1888 | 209 | 232 | 266 | 426 | 786 | | softirqs |
| 110 | 1572 | 1893 | 2099 | 233 | 267 | 433 | 79 | | stat |
| 111 | 1585 | 1904 | 21 | 234 | 268 | 449 | 8 | | swaps |
| 112 | 159 | 1915 | 210 | 235 | 27 | 466 | 80 | | sys |
| 113 | 1593 | 1919 | 2101 | 236 | 28 | 475 | 803 | | sysrq-trigger |
| 114 | 1598 | 1922 | 2105 | 237 | 29 | 481 | 82 | | sysvipc |
| 115 | 16 | 1924 | 211 | 238 | 2959 | 486 | 83 | | thread-self |

图 28: proc 列表

这个列表只显示主要信息，更详细的信息需要使用文件 id 进一步查看。

```
root@ubuntu:/home/a# ls /proc/127
```

ls: 无法读取符号链接 '/proc/127/exe': 没有那个文件或目录

| | | | | |
|-----------------|-----------|---------------|--------------|---------------|
| arch_status | environ | mountinfo | personality | statm |
| attr | exe | mounts | projid_map | status |
| autogroup | fd | mountstats | root | syscall |
| auxv | fdinfo | net | sched | task |
| cgroup | gid_map | ns | schedstat | timers |
| clear_refs | io | numa_maps | sessionid | timerslack_ns |
| cmdline | limits | oom_adj | setgroups | uid_map |
| comm | loginuid | oom_score | smaps | wchan |
| coredump_filter | map_files | oom_score_adj | smaps_rollup | |
| cpuset | maps | pagemap | stack | |
| cwd | mem | patch_state | stat | |

图 29: 详细信息

此外，proc 还支持其他参数。例如使用 mem 查看内存占用，使用 status 查看当前状态等。

题目 17.

学习使用 ltrace 跟踪程序。

解决方法:

ltrace 的功能是跟踪进程的库函数调用，对程序进行跟踪如图所示：

```
a@ubuntu:~$ ltrace ./main
fwrite("input value\n", 1, 12, 0x7f39452e0760input value
) = 12
--- SIGFPE (Floating point exception) ---
+++ killed by SIGFPE +++
```

图 30: ltrace 跟踪程序

加入参数-S 还可以打印出系统的调用情况。

```
a@ubuntu:~$ ltrace -S ./main
SYS_brk(0) = 0x55bd16c16000
SYS_access("/etc/ld.so.nohwcap", 00) = -2
SYS_access("/etc/ld.so.preload", 04) = -2
SYS_openat(0xffffffff9c, 0x7f669dd48ea8, 0x80000, 0) = 3
SYS_fstat(3, 0x7fff17bda090) = 0
SYS_mmap(0, 0x2400d, 1, 2) = 0x7f669df2a000
SYS_close(3) = 0
SYS_access("/etc/ld.so.nohwcap", 00) = -2
SYS_openat(0xffffffff9c, 0x7f669df51dd0, 0x80000, 0) = 3
SYS_read(3, "\177ELF\002\001\001\003", 832) = 832
SYS_fstat(3, 0x7fff17bda0f0) = 0
SYS_mmap(0, 8192, 3, 34) = 0x7f669df28000
SYS_mmap(0, 0x3f0ae0, 5, 2050) = 0x7f669d935000
SYS_mprotect(0x7f669db1c000, 2097152, 0) = 0
SYS_mmap(0x7f669dd1c000, 0x6000, 3, 2066) = 0x7f669dd1c000
SYS_mmap(0x7f669dd22000, 0x3ae0, 3, 50) = 0x7f669dd22000
SYS_close(3) = 0
SYS_arch_prctl(4098, 0x7f669df294c0, 0x7f669df29e00, 0x7f669df28988) = 0
SYS_mprotect(0x7f669dd1c000, 16384, 1) = 0
SYS_mprotect(0x55bd165c5000, 4096, 1) = 0
SYS_mprotect(0x7f669df4f000, 4096, 1) = 0
SYS_munmap(0x7f669df2a000, 147469) = 0
```

图 31: 调用情况

题目 18.

学习使用 Valgrind 进行调试和分析。

解决方法:

Valgrind 可以用于检测程序的内存使用问题。使用 valgrind 运行程序可以返回其执行信息:

```
a@ubuntu:~$ valgrind --leak-check=full ./hello
==6306== Memcheck, a memory error detector
==6306== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6306== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6306== Command: ./hello
==6306==
Hello Word
==6306==
==6306== HEAP SUMMARY:
==6306==     in use at exit: 0 bytes in 0 blocks
==6306==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==6306==
==6306== All heap blocks were freed -- no leaks are possible
==6306==
==6306== For counts of detected and suppressed errors, rerun with: -v
==6306== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

图 32: 执行信息

它可以用于检测内存泄漏问题, 当程序中存在这类问题时, 执行信息中会将其标出。例如在程序中使用野指针, 再次执行后返回如下结果:

```
==6346== Use of uninitialised value of size 8
==6346==    at 0x108656: main (in /home/a/hello)
==6346==
```

图 33: 野指针的检测

熟练应该这一调试工具可以有效避免内存泄漏问题。

题目 19.

学习使用 uname 查看计算机信息。

解决方法:

uname 可显示电脑以及操作系统的相关信息，例如内核版本、主机名、处理器类型等。使用 uname 默认返回当前计算机的操作类型：

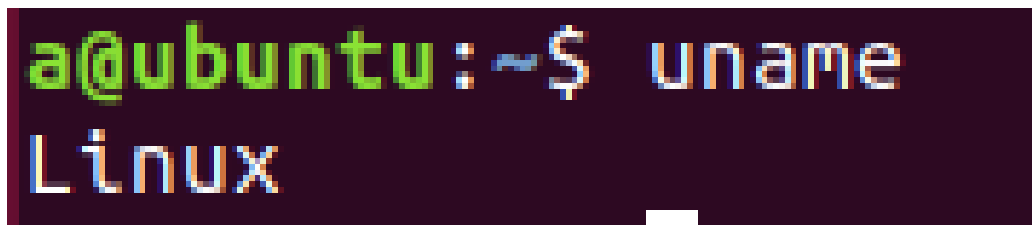


图 34: 操作系统

uname 还可以接受各种参数，例如-a 显示全部内容，-r 显示内核版本号，-p 显示处理器类型等。

```
a@ubuntu:~$ uname -a
Linux ubuntu 5.4.0-150-generic #167~18.04.1-Ubuntu SMP Wed May 24 00:51:42 UTC 2
023 x86_64 x86_64 x86_64 GNU/Linux
```

图 35: 全部信息

题目 20.

学习 readelf 的使用方法。

解决方法:

readelf 用于显示读取 ELF 文件中信息，ELF 文件应用广泛，常见于可执行文件、目标文件和库文件。下面尝试使用 readelf -a 查看 ELF 格式文件的头部信息：

```
a@ubuntu:~$ readelf -h hello
ELF 头:
  Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  类别:      ELF64
  数据:      2 补码, 小端序 (little endian)
  版本:      1 (current)
  OS/ABI:     UNIX - System V
  ABI 版本:   0
  类型:      DYN (共享目标文件)
  系统架构:   Advanced Micro Devices X86-64
  版本:      0x1
  入口点地址: 0x540
  程序头起点: 64 (bytes into file)
  Start of section headers: 6448 (bytes into file)
  标志:      0x0
  本头的大小: 64 (字节)
  程序头大小: 56 (字节)
  Number of program headers: 9
  节头大小:   64 (字节)
  节头数量:   29
  字符串表索引节头: 28
```

图 36: 头部信息

还可以使用其他参数查看程序信息，例如使用-S 查看节区信息，用-s 查看符号表，-w 查看 dwarf 格式的调试信息等。

```
a@ubuntu:~$ readelf -S hello
There are 29 section headers, starting at offset 0x1930:
节头:
[号] 名称      类型      地址      偏移量
     大小      全体大小  旗标  链接  信息  对齐
[ 0]          NULL      0000000000000000 0 0 0 0
     0000000000000000 0000000000000000
[ 1] .interp    PROGBITS  0000000000000238 00000238
     000000000000001c 0000000000000000 A 0 0 1
[ 2] .note.ABI-tag NOTE      0000000000000254 00000254
     0000000000000020 0000000000000000 A 0 0 4
[ 3] .note.gnu.build-id NOTE      0000000000000274 00000274
     0000000000000024 0000000000000000 A 0 0 4
[ 4] .gnu.hash  GNU_HASH  0000000000000298 00000298
     000000000000001c 0000000000000000 A 5 0 8
[ 5] .dynsym    DYSYM     00000000000002b8 000002b8
     00000000000000a8 0000000000000018 A 6 1 8
[ 6] .dynstr    STRTAB    0000000000000360 00000360
     0000000000000084 0000000000000000 A 0 0 1
[ 7] .gnu.version VERSYM     00000000000003e4 000003e4
     000000000000000e 0000000000000002 A 5 0 2
```

图 37: 查看节区信息

心得体会：

Linux 系统中有许多调试测试代码的工具，体现了这一系统的方便性优势。这些工具可以测试程序的各种信息，例如代码性能、出错位置、内存占用、运行进程等。使用这些工具可以让编程事半功倍。

此外 Linux 系统下还有许多其他方便的命令，例如修改按键映射、查看文件列表信息、查找代码错误位置、追踪程序或命令等。熟练使用这些功能和工具，才能真正学会 Linux 系统的应用。

Github 链接及提交记录：

GitHub 链接：<https://github.com/28935/23020007016>

commit 记录截图：



图 38: commit 截图