山东大学 计算机科学与技术 学院

操作系统 课程实验报告

学号：202200101007	姓名：张祎乾	班级：22.3 班
实验题目：实验 1：Nachos 系统的安装与调试		
实验学时：2	实验日期：2025/2/20	
实验目的： 1．安装编译 Nachos 系统，理解 Nachos 系统的组织结构与安装过程； 2．安装测试 gcc MIPS 交叉编译器； 3．掌握利用 Linux 调试工具 GDB 调试跟踪 Nachos 的执行过程； 4．安装成功后，根据 Nachos 的输出结果，分析分析跟踪 Nachos 的 C++ 程序及 汇编代码，理解 Nachos 中线程的创建方法以及上下文切换的过程。 5．阅读 Nachos 的相关源代码，理解 Nachos 内核的启动与停机过程。 6．理解 Nachos 的运行参数的含义与使用。		
实验环境：WSL、Ubuntu		
源程序清单： 无		
编译及运行结果： Threads 运行：		

```
zhang@zhang:~/OS/nachos-3.4/code/threads$ ./nachos
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 130, idle 0, system 130, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

c++example 运行结果：

```
zhang@zhang:~/OS/nachos-3.4/c++example$ ./stack
pushing 17
pushing 18
pushing 19
pushing 20
pushing 21
pushing 22
pushing 23
pushing 24
pushing 25
pushing 26
popping 26
popping 25
popping 24
popping 23
popping 22
popping 21
popping 20
popping 19
popping 18
popping 17
```

关于 2.6 的问题：

经过我反复实践，发现这些问题得到的地址不会改变

不过我得到的地址与指南不同，推测可能是所用虚拟机不同等原因

(1)三种方法：

①info address func 可以找到函数 func()的地址

②disassemble func 可以通过反汇编查看代码并看到 func()地址

③b func 可以在函数打断点，系统会提示断点地址(即函数地址)

```
(gdb) b InterruptEnable
Breakpoint 1 at 0x3021: file thread.cc, line 242.
(gdb) b SimpleThread
Breakpoint 2 at 0x326f: file threadtest.cc, line 26.
(gdb) b ThreadFinish
Breakpoint 3 at 0x2ff6: file thread.cc, line 241.
(gdb) b ThreadRoot
Breakpoint 4 at 0x4e76
```

Ⅰ.InterruptEnable()的地址为 0x3021

Ⅱ. SimpleThread()的地址为 0x326f

Ⅲ.ThreadFinish()的地址为 0x2ff6

Ⅳ.ThreadRoot()的地址为 0x4e76

(2)

简述指南：分别找到主线程和其他线程被创建的地方，打断点从相关变量的值中获取

```
(gdb) d
Delete all breakpoints? (y or n) y
(gdb) b Initialize
Breakpoint 6 at 0x2723: file system.cc, line 79.
(gdb) r
Starting program: /home/zhang/OS/nachos-3.4/code/threads/nachos

Breakpoint 6, Initialize (argc=1, argv=0xffffd364) at system.cc:79
79      {
(gdb) b 155
Breakpoint 7 at 0x5655793a: file system.cc, line 155.
(gdb) c
Continuing.

Breakpoint 7, Initialize (argc=0, argv=0xffffd368) at system.cc:155
155         currentThread->setStatus(RUNNING);
(gdb) p currentThread
$1 = (Thread *) 0x56563ca0
(gdb)
```

Ⅰ.the main thread of the Nachos：地址为 0x56563ca0

```
(gdb) l
39      //      to call SimpleThread, and then calling SimpleThread ourselves.
40      //----------------------------------------------------------------------
41
42      void
43      ThreadTest()
44      {
45          DEBUG('t', "Entering SimpleTest");
46
47          Thread *t = new Thread("forked thread"); // 创建子线程
48
(gdb) l
49          t->Fork(SimpleThread, 1);
50          // 让t线程去执行函数，第一个参数为“执行函数”，第二个参数为“执行函数”所需的参数
51          SimpleThread(0);
52      }
53
(gdb) b 51
Breakpoint 9 at 0x5655833a: file threadtest.cc, line 51.
(gdb) c
Continuing.

Breakpoint 9, ThreadTest () at threadtest.cc:51
51          SimpleThread(0);
(gdb) p t
$2 = (Thread *) 0x56563d00
```

Ⅱ.the forked thread created by the main thread：的地址为

0x56563d00

(3)

```
   0x56559ec5 <+69>:    mov    0x1c(%eax),%edi
   0x56559ec8 <+72>:    mov    0x14(%eax),%ebp
   0x56559ecb <+75>:    mov    (%eax),%esp
   0x56559ecd <+77>:    mov    0x20(%eax),%eax
   0x56559ed0 <+80>:    mov    %eax,(%esp)
   0x56559ed3 <+83>:    mov    0x5655e054,%eax
--Type <RET> for more, q to quit, c to continue without paging--c
   0x56559ed8 <+88>:    ret
   0x56559ed9 <+89>:    xchg   %ax,%ax
   0x56559edb <+91>:    xchg   %ax,%ax
   0x56559edd <+93>:    xchg   %ax,%ax
   0x56559edf <+95>:    nop
End of assembler dump.
(gdb) b *0x56559ed0
Breakpoint 2 at 0x56559ed0
(gdb) c
Continuing.

Breakpoint 2, 0x56559ed0 in SWITCH ()
(gdb) info r
eax            0x56559e72          1448451698
ecx            0x56558021          1448443937
edx            0x1                 1
ebx            0x0                 0
esp            0x56568d50          0x56568d50
```

Ⅰ.当主线程第一次运行 SWITCHO)函数,CPU 返回的地址是 0x56559e72

Ⅱ.该地址对应程序的 ThreadRoot ()

第一次调用 SWITCH(), 是从主线程(main)切换到新建的子线程（forked thread),

因此第一次调用 SWITCH, 其返回值是新建子线程（forked thread）的入口 ThreadRoot(),

即切换到 ThreadRoot()开始执行新建的子线程。

(4)

```
gs             0x63                99
(gdb) c
Continuing.
*** thread 1 looped 0 times

Breakpoint 1, 0x56559e80 in SWITCH ()
(gdb) c
Continuing.

Breakpoint 2, 0x56559ed0 in SWITCH ()
(gdb) info r
eax            0x56556a20          1448438304
ecx            0x4                 4
edx            0x0                 0
ebx            0x5655df40          1448468288
```

Ⅰ.当调用 Fork0 新建的线程首次运行 SWITCH0 函数时，CPU 返回的地址是 0x56556a20

Ⅱ.该地址对应程序的 Scheduler::Run()

子线程开始执行后，后续子线程与主线程 main 发生的上下文切换都是从上次被

中断的地方开始执行，即 Scheduler::Run()中语句 SWITCH(oldThread, nextThread)之后，

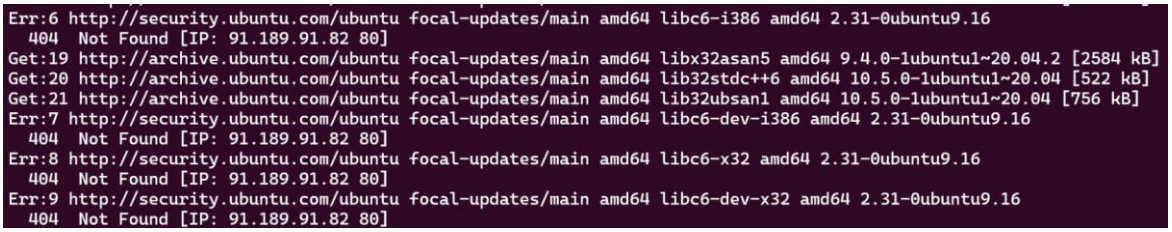
因此断点地址都是相同的，即后续 SWITCH 的返回值都是相同的。

问题及收获：

问题 1

操作位置：2.2-2.2.1-2-(2)

输入“sudo apt-get install build-essential g++-multilib gcc-multilib”，遇到报错信息如下



我讲报错信息丢给 chatGPT，得到了如下反馈

看起来你在安装或更新 `libc6-i386` 包时遇到了 `404 Not Found` 错误。这通常意味着你尝试访问的包版本在指定的 URL 中已经不可用。

你可以尝试以下几种方法来解决这个问题：

1. **更新包列表：** 运行 `sudo apt update` 来确保你的包列表是最新的。这样可以刷新可用的仓库，确保你安装的是最新的包。
2. **检查仓库 URL：** 检查 `/etc/apt/sources.list` 或其他仓库文件中的 URL 是否正确，确认这些 URL 是否仍然有效。你可能需要检查仓库服务器是否还在提供这个包。
3. **更换镜像源：** 有时候，你访问的镜像可能暂时无法使用。你可以切换到另一个镜像：
   - 打开“软件和更新”工具（或在终端运行 `software-properties-gtk`）。
   - 在“Ubuntu 软件”标签下，从“下载自”选项中选择一个不同的服务器。
4. **手动安装包：** 如果上述方法无效，你可以尝试从可靠来源（例如 Ubuntu Packages）手动下载 `.deb` 包并使用 `dpkg` 安装：

   bash 复制 编辑

   ```
   sudo dpkg -i /path/to/package.deb
   ```

解决方法：此时我打开了 VPN，并在终端执行命令 sudo apt update，再次执行“sudo apt-get install build-essential g++-multilib gcc-multilib”，成功执行完毕

问题 2

操作位置：2.2-2.2.1-3-(3)

不知道如何运行 make clean，学习到在包含 Makefile 的文件夹下运行

所以我在 code 的每一个包含 Makefile 的子文件夹中执行 make clean 和 make

在 bin 底下执行 make clean 和 make 成功，那么就可以继续跟着指南走了

发现在 lab3 和 test 中执行 make 会报错，没关系，暂时用不到，lab3 会在实验 3 解决，而 test 一会指南会解决

问题 4

操作位置：2.3-2.3.1

(a)su 命令执行失败：Authentication failure

找了半天密码，发现我压根没有设置 root 用户，解决办法：

执行 sudo passwd root

设置密码

然后就可以用 su 命令了

(b) cd /usr/local

由于权限问题，想要不用命令行，直接复制压缩包到该文件夹下是没有权限的，/usr/local 可以通过命令行寻找

(c) copy file “gcc-2.8.1-mips.tar.gz” to file fold “/usr/local” with cp command.

这句话不是命令，

先把 gcc-2.8.1-mips.tar.gz 复制到 OS 文件夹下，在终端进入 OS 文件夹

执行命令 cp gcc-2.8.1-mips.tar.gz /usr/local

(d) tar xzvf gcc-2.8.1-mips.tar.gz

执行命令 cd /usr/local 进入 local 目录下再执行 tar xzvf gcc-2.8.1-mips.tar.gz

思考：为什么 nachos-3.4.tar.gz 一定要安装在 /usr/local 目录中？

答：打开 code/Makefile.dep，在大约 38 行左右，查看变量 GCCDIR 的值，即 GCCDIR = /usr/local/mips/bin/decstation-ultrix-

交叉编译器用于对../test 目录下的 Nachos 应用程序（如 sort.c）进行编译，经转换后会生成 Nachos 可执行的文件 sort.noff：

.noff 可执行文件中的指令基于 MIPS 架构,Nachos 模拟的 CPU 执行 MIPS 架构

指南第 1 页(不计目录)写道：

Nachos 模拟的 CPU 基于 MIPS 架构，执行 MIPS 指令集，用来执行 Nachos 的应用程序；因此 Nachos 的编译程序应该将 Nachos 的应用程序（类似

于 C 程序）编译

成包含 MIPS 指令集的可执行程序，以便在 Nachos 系统上运行。

Nachos 本身没有提供编译器，其应用程序只能在 Linux 环境下编程，并在 Linux 环境下将其编译成基于 MIPS 指令集的应用程序，然后利用 Nachos 提供的工具

coff2noff 将其转换成 Nachos 的应用程序（noff 格式）。

Nachos 系统提供的交叉编译器 gcc-2.8.1-mips.tar.gz 提供的 gcc、g++、as、ld 等工具负责实现该功能。

个人理解：

/usr/local/ 目录是 Linux 系统中用于存放本地安装的软件、程序、库文件等内容的标准位置。

这个压缩包包含与 MIPS 架构相关的工具链、编译器、库或模拟器等，将它们放在 /usr/local/ 下，以便系统的其他程序或开发人员可以访问和使用。

问题 5

操作位置：2.5

在 c++example 目录下**执行 make 命令**，得到报错：

```
zhang@zhang:~/OS/nachos-3.4/c++example$ make
g++ -g -o inheritstack inheritstack.cc list.cc
inheritstack.cc:19:10: fatal error: iostream.h: No such file or directory
   19 | #include <iostream.h>
      |          ^~~~~~~~~~~~
compilation terminated.
In file included from list.cc:20:
copyright.h:25:26: warning: ISO C++ forbids converting a string constant to ‘char*’ [-Wwrite-strings]
   25 | static char *copyright = "Copyright (c) 1992,1993,1995 The Regents of the University of California.  All
 rights reserved.";
      |                          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~
make: *** [Makefile:7: inheritstack] Error 1
```

解决方案：将 inheritstack.cc、templatestack.cc 和 stack.cc 中的 <iostream.h>改为\<iostream>

再次执行 make，得到报错：

```
inheritstack.cc: In member function ‘void Stack::SelfTest(int)’:
inheritstack.cc:201:2: error: ‘cout’ was not declared in this scope; did you mean ‘std::cout’?
  201 |  cout << "pushing " << count << "\n";
      |  ^~~~
      |  std::cout
In file included from inheritstack.cc:19:
/usr/include/c++/9/iostream:61:18: note: ‘std::cout’ declared here
   61 |   extern ostream cout;  /// Linked to standard output
      |                  ^~~~
inheritstack.cc:207:2: error: ‘cout’ was not declared in this scope; did you mean ‘std::cout’?
  207 |  cout << "popping " << Pop() << "\n";
      |  ^~~~
      |  std::cout
In file included from inheritstack.cc:19:
/usr/include/c++/9/iostream:61:18: note: ‘std::cout’ declared here
   61 |   extern ostream cout;  /// Linked to standard output
      |                  ^~~~
inheritstack.cc: In function ‘int main()’:
inheritstack.cc:221:5: error: ‘cout’ was not declared in this scope; did you mean ‘std::cout’?
  221 |     cout << "Testing ArrayStack\n";
      |     ^~~~
      |     std::cout
```

解决方案，在 inheritstack.cc、templatestack.cc 和 stack.cc 中加入 using namespace std;

建议：有一个 warning，可以将 copyright.h 中 25 行的 char* 前加上 const 来解决

执行 gdb stack，得到报错：

```
zhang@zhang:~/OS/nachos-3.4/c++example$ gdb stack

Command 'gdb' not found, but can be installed with:

sudo apt install gdb
```

解决方案：执行 sudo apt install gdb

gdb 记录

l/list ——展示 10 行代码

b/break 129 ——在 129 行打一个断点

b/break func ——在名为 func 的函数上打一个断点(进入函数即遇到断点)

d ——删除所有断点

r/run ——持续运行直到遇到断点或结束

n/next ——逐语句跟踪程序

s/step ——逐条汇编指令进行

p/print/d/display ——输出程序中变量的值

info reg ——输出当前所有寄存器内容

其余 gdb 命令随用随查

p41 页回答问题

虽然这些问题是在 2.6 中给出，但是 2.7 给出了这些答案的解决方法，

我通过跟着指南操作得到了以下地址，并没有遇到什么需要补充的问题(不过看懂指南确实花了我不少力气)

经过我反复实践，发现这些问题得到的地址不会改变

不过我得到的地址与指南不同，推测可能是所用虚拟机不同等原因

(1)三种方法：

①info address func 可以找到函数 func()的地址

②disassemble func 可以通过反汇编查看代码并看到 func()地址

③b func 可以在函数打断点，系统会提示断点地址(即函数地址)

l．InterruptEnable()的地址为 0x3021

Ⅱ. SimpleThread()的地址为 0x326f

Ⅲ.ThreadFinish()的地址为 0x2ff6

Ⅳ.ThreadRoot()的地址为 0x4e72

(2)

简述指南：分别找到主线程和其他线程被创建的地方，打断点从相关变量的值中获取

Ⅰ.the main thread of the Nachos：地址为 0x56563ca0

Ⅱ.the forked thread created by the main thread：的地址为 0x56563d00

(3)

Ⅰ.当主线程第一次运行 SWITCHO)函数,CPU 返回的地址是 0x56559e72

Ⅱ.该地址对应程序的 ThreadRoot ()

第一次调用 SWITCH()，是从主线程(main)切换到新建的子线程（forked thread),

因此第一次调用 SWITCH，其返回值是新建子线程（forked thread）的入口 ThreadRoot(),

即切换到 ThreadRoot()开始执行新建的子线程。

(4)

Ⅰ.当调用 ForkO 新建的线程首次运行 SWITCHO 函数时，CPU 返回的地址是 0x56556a20

Ⅱ.该地址对应程序的 Scheduler::Run()

子线程开始执行后，后续子线程与主线程 main 发生的上下文切换都是

从上次被

中断的地方开始执行，即 Scheduler::Run()中语句 SWITCH(oldThread, nextThread)之后，

因此断点地址都是相同的，即后续 SWITCH 的返回值都是相同的。

问题 6

我在完成实验一后，对 OS 文件夹复制备份，发现有 16 个文件无法复制，明明是空文件夹，但是系统提示我这 16 个文件重名

当我点击覆盖，会卡死，经过检查发现，这 16 个文件均是执行 make 后得到的,于是跳过