报告四：调试与性能分析、元编程和 PyTorch 编程

23020007011 崔涛

2024 年 9 月 15 日

1 实验目的

本次课程主要讲授了调试及性能分析，元编程和 PyTorch 编程，通过学习这些工具可以帮助我们优化模型性能，提高代码质量和加速以后的开发过程。

2 介绍

2.1 调试及性能分析，元编程和 PyTorch 编程

- 调试及性能分析的工具提供了可视化界面，使得开发者能够直观地看到程序运行过程中的性能数据，从而更容易发现潜在问题，还能提供关于如何优化瓶颈的具体建议，帮助开发者改进模型性能。

- 元编程允许开发者在运行时动态地创建和修改代码，这为 PyTorch 等深度学习框架提供了极大的灵活性。通过元编程，可以自动化许多重复性的编程任务，减少人为错误并提高开发效率。

- PyTorch 是一个广泛使用的深度学习框架，它提供了丰富的 API 和工具来构建、训练和部署深度学习模型。PyTorch 以其易用性和灵活性而闻名，支持多种编程范式（如命令式和函数式编程），并且与 Python 生态系统紧密集成。

3 练习内容

3.1 学习样例 20 个

1. 使用 Linux 上的 `journalctl` 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。

输入 journalct1 来得到超级用户的执行和登录信息，输入 journalct1|grep sudo 来得到超级用户执行的指令。

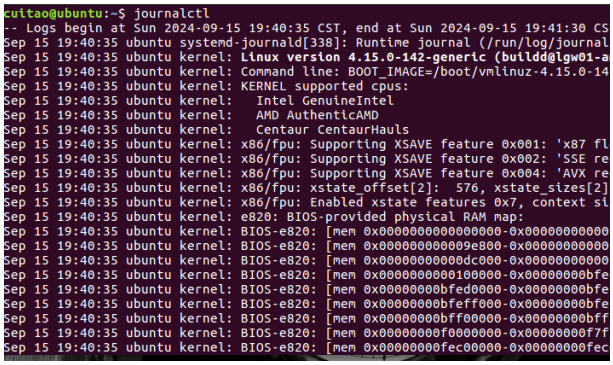

图 1: 获取超级用户执行信息

2. 安装 `shellcheck` 并尝试对脚本进行检查。这段代码有什么问题吗?
安装 shellcheck 后输入 shellcheck 加文件名字，即可检查

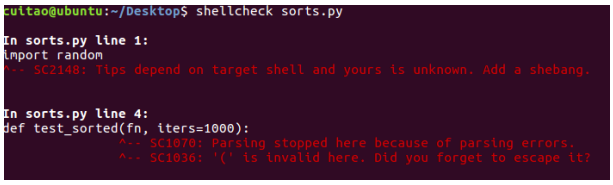

图 2: Shellcheck 检查代码

3. 使用 cProfile 来比较插入排序和快速排序的性能
输入如下代码使用 cprofile 比较测试，*python − m cP rof ile − s time sorts.py* 比较得出：在测试情况下，插入排序的性能比较好。

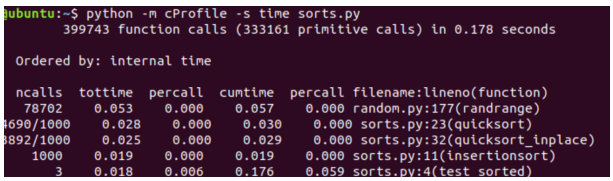

图 3: CProfile 比较性能

4. 限制进程资源也是一个非常有用的技术。执行 stress -c 3 并使用 htop 对 CPU 消耗进行可视化。先输入 sudo apt install htop 下载 htop, 再输入 stress -c 3 来创建负载，再输入 taskset –cpu-list 0,2 stress -c 3 限制负载，使用 htop 命令比较前后差异。

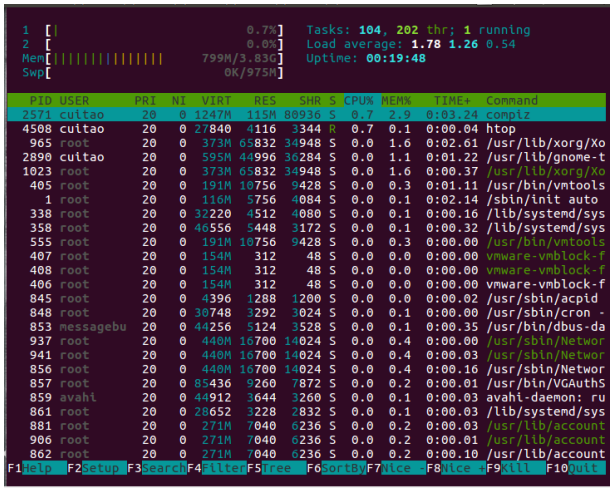

图 4: 限制前

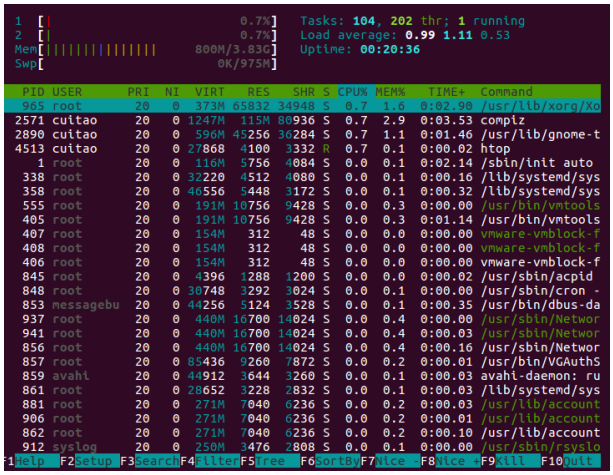

图 5: 限制后

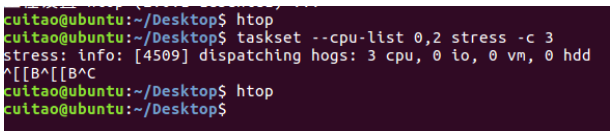

图 6: 限制指令

5. 使用 line_profiler 来比较插入排序和快速排序的性能。

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 @*prof ile*, 插入后再输入指令 *kernprof − l − v sorts.py* 即可得出代码块性能。

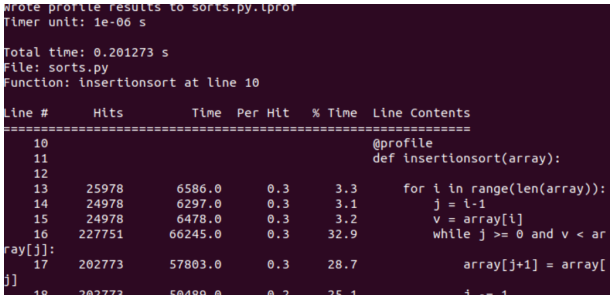

图 7: Line_profiler 分析算法性能

6.lsof 命令的使用命令格式为 **lsof [选项] [绝对路径的文件名]** 列出被各种进程打开的文件信息

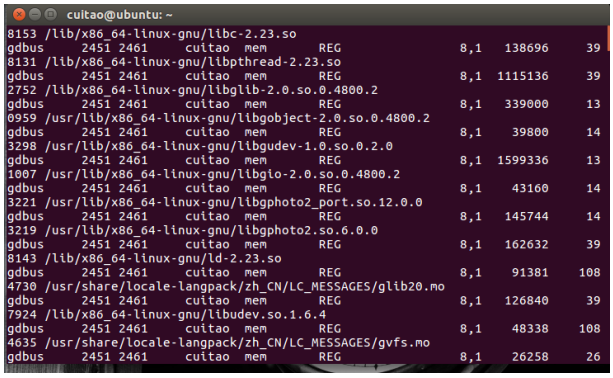

图 8: lsof 命令

7.strace 调试命令 `strace <命令>`：跟踪系统调用和信号。它可以显示程序执行过程中所调用的所有系统调用以及这些调用的参数、返回值等信息。例如：`strace ls -l`会显示执行`ls -l`命令时的系统调用信息。

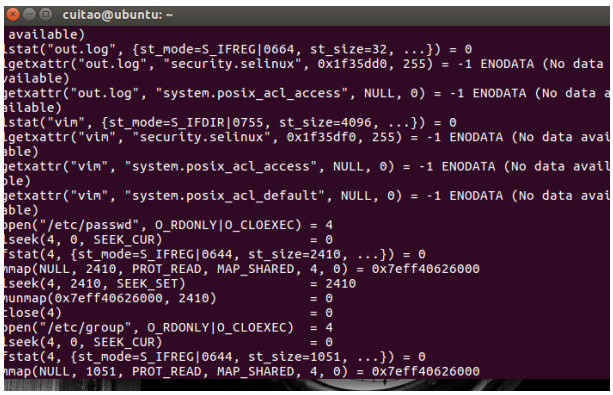

图 9: Strace 调试

8.ltrace 指令 `ltrace <命令>`：跟踪程序运行过程中库函数的调用情况。比如查看一个程序在运行时调用了哪些动态链接库中的函数以及这些函数的参数和返回值。例如：`ltrace my_program`可以跟踪`my_program`运行时的库函数调用

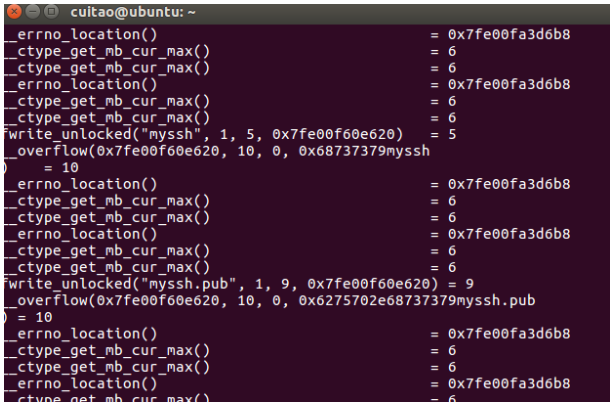

图 10: ltrace 指令

9.valgrind 相关

- `valgrind --tool=memcheck <可执行程序>`：用于检测程序中的内存错误，比如内存泄漏、越界访问、使用未初始化的内存等。它可以提供详细的内存错误信息以及错误发生的位置。
- `valgrind --tool=callgrind <可执行程序>`：分析程序中函数的调用关系和执行时间，用于性能调优，可帮助找出程序中哪些函数占用了过多的时间。

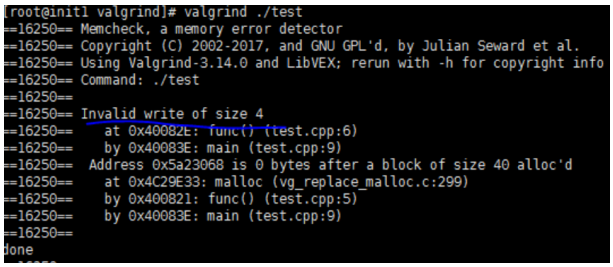

图 11: valgrind 指令

10.core dump 相关

`ulimit -c unlimited`：设置 core 文件大小无限制。当程序崩溃时，系统会生成一个 core 文件，里面包含了程序崩溃时的内存、寄存器等信息。

11.gdb 调试分析

`gdb <可执行程序> <core 文件>`：利用生成的 core 文件进行调试，可以找出程序崩溃的位置以及当时的变量状态等信息。例如，程序崩溃后生成了 core 文件，可以通过`gdb my_program core`来进行调试分析。

12nm

- 格式：`nm <可执行文件或目标文件>`。
- 作用：列出目标文件或可执行文件中的符号，比如全局函数、全局变量等，可用于检查符号是否正确链接等问题。

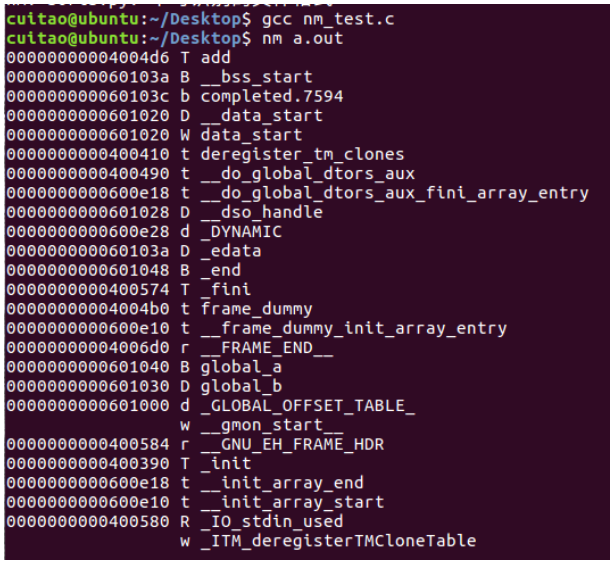

图 12: nm 指令

13.readelf

- 格式：`readelf -a <可执行文件或目标文件>`。
- 作用：显示 elf 文件（可执行文件和目标文件在 Linux 下多为 elf 格式）的详细信息，包括头信息、段信息、符号表等，对分析文件结构和链接相关问题有帮助。

图 13: readelf 指令

14. 使用 usname 查看计算机信息 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。

图 14: uname 指令

15.objdump

- 格式：`objdump -d <可执行文件或目标文件>`。

- 作用：用于反汇编可执行文件或者目标文件，可以查看机器码对应的汇编指令，帮助调试程序在底层的执行逻辑。

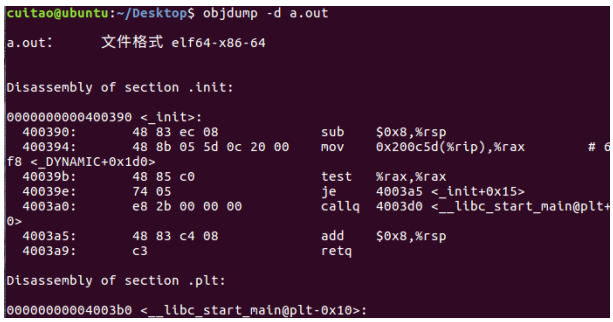

图 15: objdump 指令

16.dmesg

- 格式：直接运行`dmesg`。
- 作用：查看内核环形缓冲区的信息，其中可能包含硬件相关的错误信息、驱动加载相关信息等，对调试与内核相关或者硬件相关的问题提供参考。

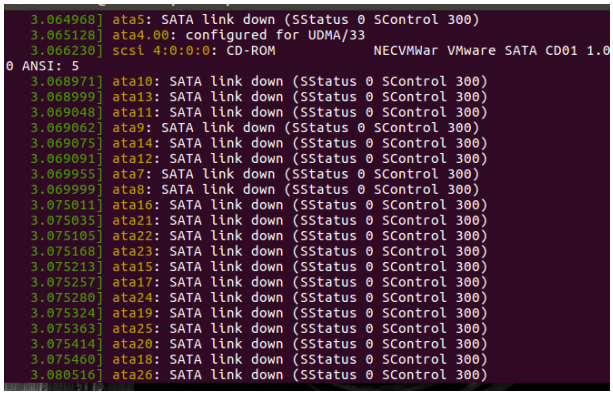

图 16: dmesg 指令

17.perf

- 格式：例如`perf record <命令>`进行数据采集，`perf report`查看报告。
- 作用：用于 Linux 性能分析，它可以监测诸如 CPU 时钟周期、指令执行次数、缓存命中率等多种性能相关的数据，能帮助定位性能瓶颈所在。

18.stap

- 格式：编写 SystemTap 脚本，然后使用`stap <脚本>`运行。
- 作用：可以动态地收集和分析内核及用户空间程序的运行时信息，可用于调试复杂的系统级问题，比如内核模块间的交互、系统调用的行为等。

19.gprof

- 格式：先使用`gcc -pg <源文件>`编译程序，运行程序后再执行`gprof <可执行程序>`。
- 作用：用于分析程序的性能瓶颈，它可以给出每个函数被调用的次数以及在每个函数上花费的时间等信息，帮助开发者进行性能优化相关的调试工作。

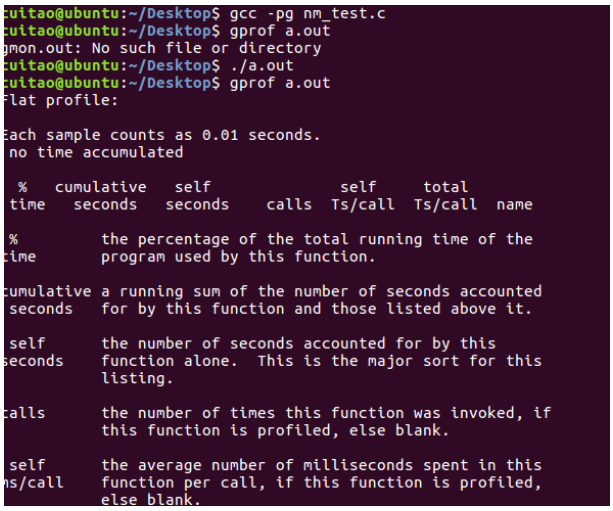

图 17: gprof 命令

20.systemctl status

- 格式：`systemctl status <服务名称>`。
- 作用：当调试与系统服务相关的问题时，这个指令可以查看服务的运行状态、最近的日志信息等，帮助判断服务是否正常工作以及出现问题的可能原因。

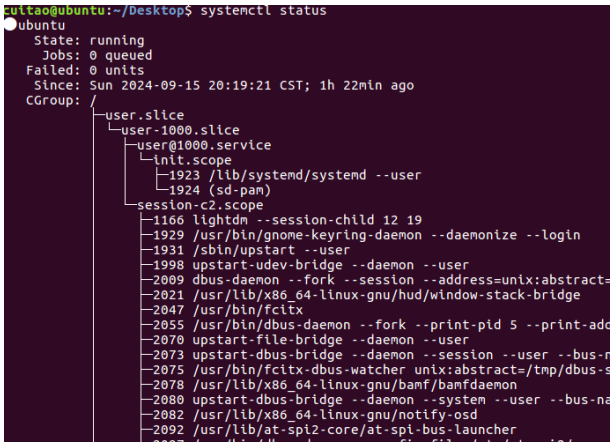

图 18: systemctl status 命令

4 收获感悟

在学习 Linux 下调试与分析的过程中，我收获颇丰。我了解了各种调试工具如 GDB 。通过 GDB，我能深入到程序运行的内部，逐行跟踪代码执行，精准定位到程序崩溃或者出现异常的位置。我学会了分析内存相关问题，理解了内存布局并且学会使用一些内存泄漏检测工具，这有助于我优化程序的内存使用，避免因内存问题导致的程序不稳定。同时，学习分析系统调用和内核日志也让我对 Linux 系统的运行机制有了更深入的认识。从系统调用的角度去理解程序与系统的交互，能发现隐藏在程序运行背后的问题根源。总的来说，本次学习极大丰富了我的技能。它不仅能提高我的程序的质量和稳定性，还能加深我对操作系统原理的理解，为进一步深入学习和开发工作奠定了坚实的基础。
提交记录如下：

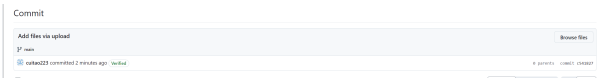

图 19: 提交记录

github 路径您可以在此查看项目的源代码:
`https://github.com/cuitao223/homework4`