

M1: 打印进程树 (pstree)

March 7, 2025

📅 2025

计算机世界有趣的地方在于，你可以动手构建任何你认为应该可以实现的东西。我们的热身实验是一个很简单的代码练习——但我们并不是实现类似于排序、最短路径这样的算法习题，而是一个真正有意义的“实用工具”：了解操作系统中运行的程序(进程)之间的层级关系，并打印进程树。

🕒 **Soft Deadline: 2025 年 3 月 18 日 23:59:59**

你需要首先阅读[实验须知](#)，其中包含了代码获取方法、提交方法、如何查看提交结果等。在命令行 `git pull origin M1` 下载框架代码。

⚠️ 学术诚信 (Academic Integrity)

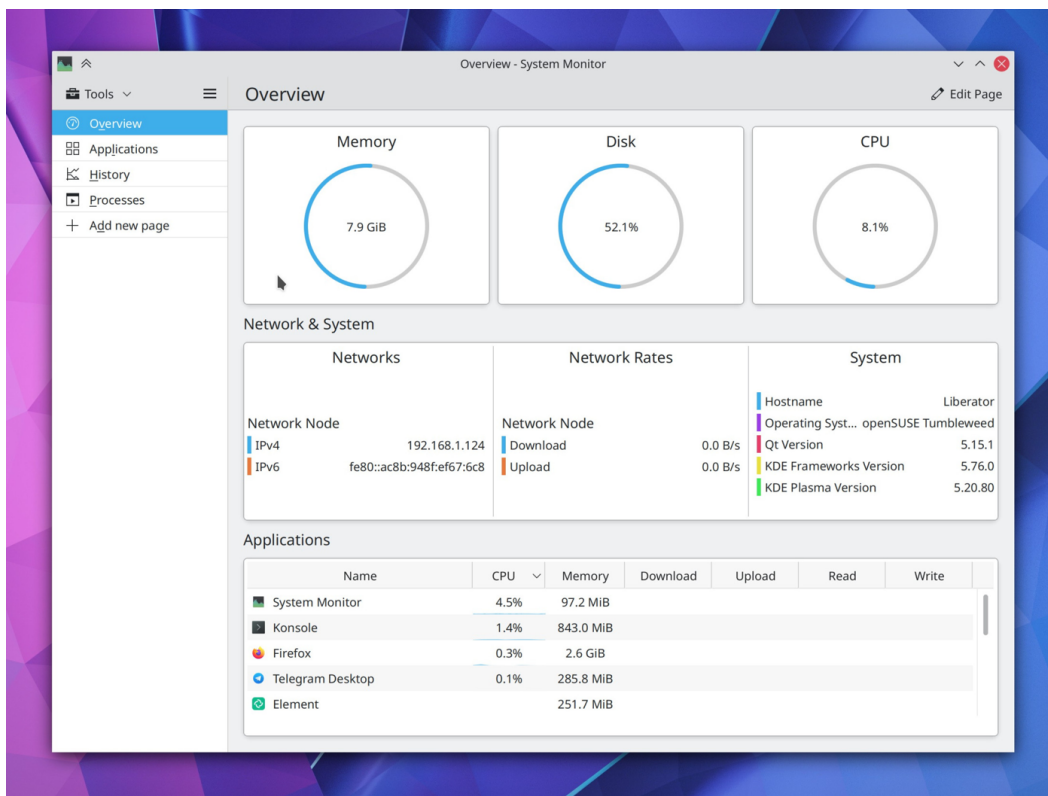
正如课堂上所说，主动“参考”他人的代码、使用他人测试用例都是不严格要求自己的行为。为了使你变得更强大，遵守学术诚信可以使你获得真正的训练。坚信**计算机世界里没有玄学**，无论是 C 代码、汇编代码还是处理器，都可以看作是严格的数学对象，可以使你在遇到问题时少一些焦躁，冷静下来分析下一步应该做什么。

具体到这个实验，你应当主动地避免阅读任何人(包括互联网)上打印进程树或进程列表的代码，但可以查看 API (如 `readdir` 等)的使用方法和例程。遇到问题时尽量自己调试解决，但可以向他人(包括人工智能)请教调试的技巧，例如“我遇到了 XX 现象”，但不知道下一步应该怎么做。

为了确保你对操作系统有真实的了解，**本课程禁止直接使用 AI 生成的代码和直接复制互联网上的代码**。你应当主动不要求 AI 写代码或从互联网上复制代码，但你可以询问 AI 或在互联网上搜索 API 的使用方法。

1. 背景

我们早已熟知操作系统拥有同时运行多个的应用程序的能力——我们可以同时玩游戏的时候聊 QQ、放音乐……那么，管理这些同时运行的程序就是一项“应该可以实现”的功能，于是我们就来实现它。今天的操作系统都提供[任务管理器](#)工具监测程序运行状态，例如显示在一段时间内，各个程序(进程、状态机)的活跃程度、占用的内存等等。下面的图片展示了 Plasma Desktop 的任务管理器，能够显示系统资源的使用情况和进程的信息。



2. 实验描述

实验要求：实现 pstree 打印进程之间的树状的父子关系

Linux 系统中可以同时运行多个程序。运行的程序称为**进程**。除了所有进程的根之外，每个进程都有它唯一的父进程，你的任务就是把这棵树在命令行中输出。你可以自由选择展示树的方式 (例如使用缩进表示父子关系)。

Linux 系统自带了 `ps` 命令，进程树会以非常漂亮的格式排版 (每个进程的第一个孩子都与它处在同一行，之后的孩子保持相同的缩进)：

```
systemd├─accounts-daemon├─{gdbus}└─{gmain}├─acpid├─agetty├─atd├─cron├─dbus-daemon├─dhclient├─2*[iscsid]├─lvmetad├─lxcfs─10*[{lxcfs}]├─mdadm├─polkitd├─{gdbus}└─{gmain}├─rsyslogd├─{in:imklog}├─{in:imuxsock}└─{rs:main Q:Reg}...
```

Linux 的 `psmisc` 中 `ps` 的实现大约有 1,300 行，支持多种命令行参数——但在这个实验中，我们希望大家避免不必要的麻烦，使用最简单的绘制方法。大家可以先玩一下 Linux 的 `ps`，使用 `man` 命令查看 `ps` 支持的功能，并试一试。在这个实验中，我们需要它的简化版：

2.1 总览

你需要在 `/pstree/pstree.c` 中完成你的代码。

2.2 描述

把系统中的进程按照父亲-孩子的树状结构打印到终端。

- `-p` 或 `--show-pids`: 打印每个进程的进程号。
- `-n` 或 `--numeric-sort`: 按照pid的数值从小到大顺序输出一个进程的直接孩子。
- `-V` 或 `--version`: 打印版本信息。

你可以在命令行中观察系统的 `ps` 的执行行为 (如执行 `ps -V`、`ps --show-pids` 等)。这些参数可能任意组合，但你不需要处理单字母参数合并的情况，例如 `-np`。

2.3 解释

☕ 对上面的描述感到困惑?

上述实验要求描述是参照 man page 的格式写出的，其中有很多 UNIX 命令行工具遵守的共同约定 (UNIX 的资深用户对此了如指掌；但对给初学者，尤其是从出生以来就生活在 GUI 环境中而不是遇事就读手册的大家造成了很大的困扰)，例如 [POSIX对命令行参数有一定的约定](#)。

觉得文档很长？让人工智能帮你做个总结吧，它能提取出其中最重要的部分，~500 字的总结就已经非常到位了！**读完以后，你立即发现你对手册的理解增加了** (手册的格式竟然也是 POSIX 标准的一部分！)。

以下对 `ps` 总览和描述的一些解释：

1. 中括号扩起的参数是可选参数，`[]` 后的 `...` 代表参数的 0 次或多次重复。因此 `-p`，`-n`，`-V` 都是可选的参数。
2. 同一个选项可以有别名。在 `ps` 中，`-p` 和 `--show-pids` 的含义是一样的。
3. 若不另行说明，整数范围在 32 位有符号整数范围内；但如果数值和文件大小有关，则其合法的范围是 0 到系统最大支持的文件大小。

⚠ 请遵守 UNIX 对返回值的约定

`main` 函数的返回值代表了命令执行的状态，其中 `EXIT_SUCCESS` 表示命令执行成功，`EXIT_FAILURE` 表示执行失败。对于 POSIX 来说，0 代表成功，非 0 代表失败：例如 `diff` 返回 1 表示比较的文件不同，返回 2 表示读取文件失败 (`cmp` 的行为也类似)。

按照 UNIX 惯例，`main` 函数返回非 0 将在 Online Judge 中被判定为 Runtime Error。

如果不知道这些约定，使用 Linux/Unix 的时候就会举步维艰。Unix 世界有一套自己定义的“游戏规则”。也难怪会有笑话：

Unix is user-friendly – it's just choosy about who its friends are.

当然，在渐渐熟悉游戏规则以后就会发现，这套设计在 1970 年代有限的计算机算力和使用者的负担之间达成了优雅的平衡——Shell 总是可以帮助你最短的时间把 quick and dirty 的事情搞定：UNIX 哲学强调 *Keep it simple, stupid* 和 *Everything is a file*。

3. 正确性标准

你可以任意选择树的形态，以下输出都是合法的：

```
$ ./pstree-64
systemd└─accounts-daemon└─
    ...
```

```
$ ./pstree-64
systemd
|
+--accounts-daemon-
|
...
```

```
$ ./pstree-64
systemd
  accounts-daemon
  ...
```

只要输出系统中的进程即可；此外，允许进程列表有轻微出入。细心的同学可能发现你第一个版本的 `pstree` 可能和系统输出不太一样。在线评测会容忍你输出的一些缺陷；此外，作为第一个实验，我们会手下留情，没有非常强劲测试数据。但你仍然需要确保：

1. 正确列出系统中的进程，并正确实现参数组合的行为；
2. 编写可移植的代码。我们会同时测试 32-bit 和 64-bit 的版本。

在 Hard Test 上 Wrong Answer 的提示

试一试 `pstree -V > /dev/null`，你会发现输出并没有到 `/dev/null`。我们希望你的行为和系统中的 `pstree -V` 基本一致：输出到正确的输出流、包含“pstree”字符串，其他描述可以不同。

4. 问题分析

根据《操作系统》课程重要的基本原理：合理的需求一定有合适的方法满足，操作系统一定会提供进程相关的对象和获取信息的 API。

暂停一下，想一想再继续！

如果你是操作系统的设计者，你会提供怎样的 API (syscall)?

一个可行的想法是操作系统可以提供类似迭代器的 API，可以在某个时刻对进程列表进行“快照”，然后程序可以通过 API 迭代快照里的进程。

```
Snapshot *CreateProcessSnapshot(); // 迭代开始
Process *FirstProcess(Snapshot *snapshot); // 取得第一个进程
Process *NextProcess(Process *process); // 获得下一个进程
int ReleaseProcessSnapshot(Snapshot *snapshot); // 迭代结束
```

高级语言可以进一步封装，例如借助 WMI (Windows Management Instrumentation) 库：

```
import wmi
for proc in wmi.WMI().Win32_Process():
    ...
```

UNIX 操作系统的设计者用另一种方法使应用程序能访问进程列表：操作系统会不断更新一个对象（文本文件）的内容，这样应用程序就能用文件 API (open, read, close) 来获取进程列表，例如大家可以用熟悉的 C 语言 `FILE *` 访问。

今天我们可以考虑在系统里创建一个名为 `/system/processes.json` 的文本文件，每当进程创建或退出，这个文件的内容就会更新（当然，操作系统保证更新在瞬间完成）：

```
[
  {
    "pid": 1,
    "parent": -1,
    "command": "/bin/init"
  },
  {
    "pid": 2,
    "parent": 1,
    "command": "/bin/bash"
  }
]
```

UNIX 采用了 Everything is a File 的设计。换句话说，我们可以把操作系统的状态变成文件系统的一部分，从而可以使用我们熟悉的 read, write 等 API 访问操作系统中的各类信息。在这个实验中，我们学习 UNIX/Linux 是如何把操作系统的状态放在文件系统之中的。虽然这个实验里你只需要读取进程列表和进程之间的父子关系，但用类似的办法，也可以从 Linux 系统中读取 CPU 占用率、内存使用等信息——于是你也可以实现自己的任务管理器了！

💡 思考题：两种机制的优点和缺点 🌶️

两种机制，到底谁好谁坏？

Everything is a file 和提供 type-safe 的 API 都不是完美的。前者对人类用户来说更好用，例如可以用 UNIX 命令行工具任意解析而不需要写代码，但带来系统中很多“隐藏在角落里的特性”；后者把规约写在 API 里，可以在你犯错时更好地保护你、运行也更高效，但“上手”也稍稍困难一些：Windows Kernel + GDI 的 Windows API 就在 1,000 个以上。但是，我们预见到“大语言模型和程序合成（低代码）”技术能够从根本上改变我们开发的方式，操作系统 API 会逐渐成为一种只有库函数使用的“汇编语言”。

5. 实验指南

🍵 解决问题的一般方法

因为人的脑容量有限，通常解决问题的办法就是把比较复杂的问题分解成小问题，再把小问题继续分解下去。而在学校里所做的训练就是建立问题分解的思路和培养解决问题的能力。如果想自己尝试，我们也鼓励大家忽略下面的教程，自己动手搞定，遇到不明白的地方可以求助 Google (Bing, Stackoverflow, ...); 但对于这样的简单任务，大语言模型的表现就有些太好了 (请不要用它们!)。完成之后可以看一下实验指南，看自己的理解是否有可以改进的空间。

如果你觉得打印进程树这个问题比较困难，我们也把问题分解一下：

1. 得到命令行的参数，根据要求设置标志变量的数值；
2. 得到系统中所有进程的编号 (每个进程都会有唯一的编号) 保存到列表里；
3. 对列表里的每个编号，得到它的父亲是谁；
4. 在内存中把树建好，按命令行参数要求排序；
5. 把树打印到终端上。

5.1. 命令行参数

main 函数的参数是进程“初始状态”的一部分，它是由进程的创建者决定的，操作系统负责把它们放在内存中适当的位置；作为 C 语言的程序员，我们只要直接访问 main 函数的参数即可：

```
#include <stdio.h>
#include <assert.h>

int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++) {
        assert(argv[i]); // C 标准保证
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    assert(!argv[argc]); // C 标准保证
    return 0;
}
```

你可以尝试编译运行这个程序，并且这个程序传入不同，并且观察程序的输出。你会确认 `argv[0]`, ..., `argv[argc-1]` 就是所有命令行的参数，这是操作系统与 C 程序之间的约定。 `getopt` (man 3 `getopt`) 库可以处理命令行参数；当然，你也可以直接自己动手解析。

5.2. 开始做实验

回想一下大家做 OJ 题的过程。在编程的过程中，难免会经历修改代码 → 编译 → 运行 → 修改代码.....这样的循环。你会选择怎么做呢？新手每次都键入命令 (或者他发现 Ctrl-p 可以重复命令)。

- 之后，有同学人发现，可以把命令写在一行里，比如 `gcc a.c && ./a.out`，一键就能编译运行了。
- 再之后会发现可以写个 Makefile (就像这个实验一样)，用 `make test` 跑完所有测试。
- 再之后会发现可以每次在文件改动以后自动运行所有测试.....有个神奇的命令叫 `inotifywait` (我们并不推荐)。

即便现在有 IDE 和丰富的插件，UNIX 哲学依然是无处不在的 (甚至是这些 IDE 的组成基础)，说得更具体一点，“只要你敢想，就一定能做到”。是的，因为之后会反复编译运行这个程序，所以**编译和测试自动化**非常重要。常见的 C 项目组织是编写 Makefile，在命令行中使用 `make` 实现编译，`make test` 完成测试。我们已经为大家提供了 Makefile，欢迎大家仔细阅读 (但不是强制的)。

⚠ 请用给定的 Makefile 编译程序

只有用 `make` 命令编译，才会留下你的开发历史——实验的编译迟早会复杂到无法“手工键入命令”完成。我们非常鼓励你在 IDE (例如 `vscode`) 中配置好 build script，这样可以一键编译/调试/运行。

以下两点有助于调试时放平心态：(1) **机器永远是对的**；(2) **未测代码永远是错的**。祝大家编程愉快！

5.3. 得到系统中进程的编号

进程是操作系统中的对象，因此操作系统一定提供了 API 访问它们。已经剧透过，系统里的**每个进程都有唯一的编号**，它在 C 语言中的类型是 `pid_t`。不知道这是什么？当然是想办法啦！除了找人工智能“拿来”一个答案，在互联网上能找到更权威的定义：[glibc 对它的官方文档解释](#)。以后遇到问题要自己找答案哦！

操作系统以什么样的方式让你获取系统里的进程呢？之前也提示过：

Everything is a file.

一切皆文件，进程信息当然也可以是“一切”的一部分。Linux 提供了 `procfs`，目录是 `/proc`。如果你进去看一眼，就会发现除了一些比如 `cpuinfo` 这样的文件，还有很多以数字命名的目录——聪明的你也许已经猜到了，每个目录的名字就是进程号，目录里存储了进程相关的运行时数据。

我们鼓励大家先玩一玩 `procfs`，里面可有很多有趣的东西！你可以用 `cat` 可以打印文件的内容，对照文档 (或者 ChatGPT)，你会发现原来我们以为离我们很遥远的“观测进程执行”，简单得只要解析文本文件就可以了！例如，每个进程的父进程也隐藏在 `/proc/[pid]/` 中的某个文件里。试试 `man 5 proc`，里面有详细的文档。很多 Online Judge 都使用 `procfs` 读取进程的运行时间/内存数据。

5.4. 遍历所有进程

了解了 `procfs` 之后，我们的问题就变得简单一些了：只要能得到 `/proc` 目录下的所有以数字为开头的目录，我们就遍历了系统中的进程。因此你会去互联网上搜索如何用 C 语言遍历目录。之后，你可以用你熟悉的方式打开 `procfs` 里的文件：

```
FILE *fp = fopen(filename, "r");
if (!fp) goto release;

// 用fscanf, fgets等函数读取

release:
if (fp) fclose(fp);
```

procfs 里的信息足够让你写一个非常不错的任务管理器。那么，“真正”的任务管理器，例如 ps 命令，是否也是基于 procfs 实现的呢？这就是一个典型的“好问题”：他帮助你建立你的实验作业和真实系统之间的联系。操作系统课程也给大家足够的工具，使得同学们可以把任务管理器打开，查看它调用的操作系统 API。我们在课堂上已经演示过 gcc 和 xedit 的例子，就用 strace 工具就能查看进程运行时的系统调用序列：

```
$ strace ps
...
openat(AT_FDCWD, "/proc/1/stat", O_RDONLY) = 6
read(6, "1 (systemd) S 0 1 1 0 -1 4194560"... , 1024) = 190
close(6) = 0
...
```

5.5. 建树和打印

终于，在得到进程编号 (名字) 和父子关系的基础上，我们就有了一个“算法题”，它输入由有向边构成的树，要求输出图形化表示的树结构：

```
1 2
2 3
2 4
3 5
4 6
3 7
```

我们不妨把有根树理解成一个“括号序列”(前序遍历)：

```
1(2(3(5,7),4(6)),8)
```

按照从左到右的顺序，输出每一个节点的编号——需要打印多少空格呢？答案是括号嵌套的深度，我们不妨试试看：

```
1
 2
   3
    5
    7
  4
   6
 8
```

看起来挺正确的！你就得到了一个“基本款”的 pstree。如果你想好看一些，你只要考虑在空间里填上什么样的字符就行了——没错，我们只要“向左走”，找到它的父亲所在的列 (边向左走边画“-”)，最后画上垂直的线，我们就得到了树状的结构！

```
1
+-2
| +-3
| | +-5
| | +-7
| +-4
|   +-6
+-8
```


看起来不错呢！我们找到问题正确的切入点，就可以简化问题。

5.6. 写出正确的代码

☕ 完成了？

是时候问问自己：我的程序对吗？

虽然在这个实验里，我们的测试用例相对简单；但在未来的实验中，Online Judge 可能会在各种奇葩的条件下运行你的程序哦！除了你们做的 OJ 题中会有复杂的逻辑 (参数的组合) 导致 bug 之外，和系统打交道的编程可有更多的麻烦之处：

1. 你的程序遵守 POSIX 的返回值规定吗？如果你的 main 函数返回了非 0 的数值，我们将认为程序报告了错误——在非法的输入上返回 0，以及在合法的输入上返回非 0 都将导致 Wrong Answer。
2. 程序够 robust 吗？它会不会在一些非法的输入上 crash？如果系统里的进程很多呢？如果内存不够了呢？如果 `open` 或者 `malloc` 失败了呢？要知道，crash 一般是因为 `undefined behavior (UB)` 导致的——UB 没把所有的文件都删掉真是谢天谢地了。
3. 万一我得到进程号以后，进去发现文件没了 (进程终止了)，怎么办？会不会有这种情况？万一有我的程序会不会 crash.....？
4. 进程的信息一直在变，文件的内容也一直在变 (两次 `cat` 的结果不同)。那我会不会读到不一致的信息 (前半是旧信息、后半是新信息)？这两个问题都是 race condition 导致的；我们将会在并发部分回到这个话题。
5. 如果我不确信这些事会不会发生，我有没有写一个程序，至少在压力环境下测试一下它们有没有可能发生？嗯，如果我同时运行很多程序，每个程序都不断扫描目录、读取文件，也观察不到这个问题，至少应该可以放点心。

随着课程的深入，这些问题都会得到解答。

当你的程序越来越复杂，这些问题也许将会成为你挥之不去的阴影。这就对了——从 Intel 的 CPU 到 Linux Kernel 都有数不清的 bug。你也许听说过“形式化验证”，但事实也证明，经过验证正确的编译器 (CertComp) 和操作系统 (seL4, FSCQ, ...) 都依然存在 bug，尽管它们的可靠性依然比程序员手写的高得多。

写出正确的代码远比想象中困难——目前地球上还没人能保证复杂的系统没有 bug 和漏洞。我们切盼望着没有 bug 的那一天的到来——我们离这个终极的答案似乎越来越近，但依然虚无缥缈。不过也不用太绝望，这门课里会教给大家一些有关“写代码”的知识，更重要的是正确的解决问题的思维方式 (“世界观”)。