M1: 打印进程树(pstree)

截止日期

Soft Deadline: 3月24日23:59:59。

收到的作业

引子：用什么样的API获取系统中正在运行的进程?

从自顶向下的角度看，操作系统就是“一组API规约”。我们学习编程的过程，其实已经不由自主地使用了很多API，例如打开文件会涉及到 open 系统调用；printf 最终会调用 write 系统调用，向编号为1的文件描述符写入数据。这些操作系统API对我们来说都是自然而然的——但操作系统上并不是所有程序都那么“简单”，操作系统API可以用来实现**任何应用软件**，包括所谓的“系统软件”。

因此，“获取操作系统中运行的程序”似乎是操作系统应该为应用程序提供的API，不然就没法实现任务管理器了。很自然会想操作系统可以提供一个类似迭代器的API:

```
Snapshot *CreateProcessSnapshot();
Process *FirstProcess(Snapshot *snapshot);
Process *NextProcess(Process *process);
int ReleaseProcessSnapshot(Snapshot *snapshot);
```

Windows API就是这么设计的。不过这么做也会使操作系统API的数量**暴涨**，因为所有事情都要通过特定的API完成——如果考虑Windows API动态链接库导出的符号，那么仅Kernel和GDI的Windows API就在1,000个以上。

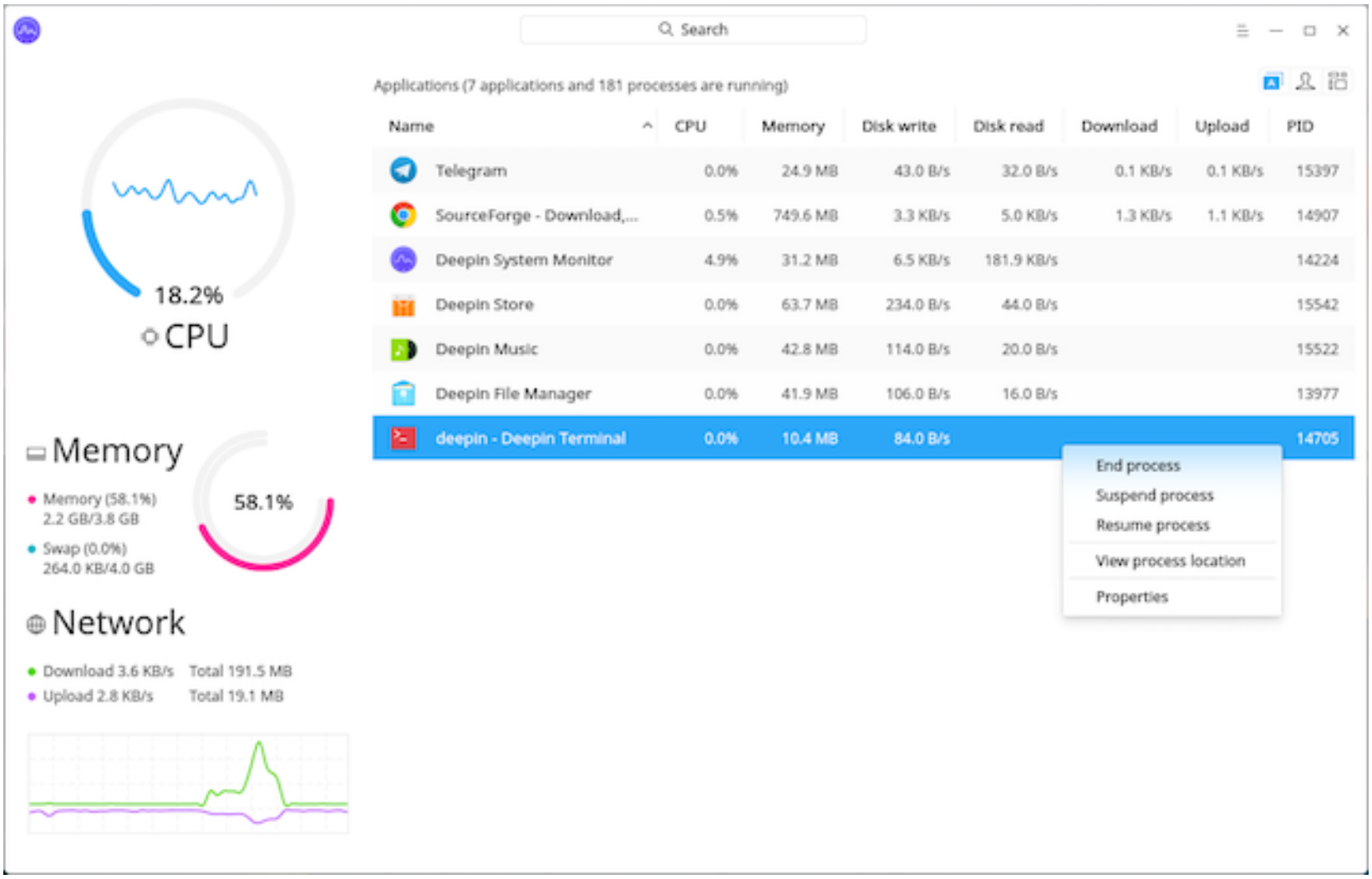
更简单的办法：我们可以在操作系统中创建一个虚拟的文本文件，这个文件的内容恰好是操作系统中的进程，例如 /system/process_list :

```
- pid: 1
  parent: -1
  command: /bin/init
- pid: 2
  parent: 1
  command: /bin/bash
...
```

这样我们就可以用C的文件操作API来获取进程信息了——**我们把操作系统的状态变成了文件系统的一部分**。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中的。

背景

操作系统能同时运行多个程序。大家都用过任务管理器 (https://www.cyberciti.biz/tips/top-linux-monitoring-tools.html)——能够显示当前系统运行的状态、进程、处理器占用等等。



上图是Deepin的任务管理器，除了系统的状态，还可以显示进程的信息。任务管理器当然也是操作系统支持的应用程序，本学期的第一个实验就是实现任务管理器中进程列表的部分。用类似的办法，可以从Linux系统中读取出CPU占用率、内存使用等信息，你其实就已经获得了实现任务管理器的足够数据。

实验描述

获取实验代码与提交

本学期的所有代码(minilab, OSlab)都在同一个目录中完成。请参考代码获取与提交 (OS2019_Code)。

`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，这是最容易的)。在Linux中，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 `pstree` 的实现大约有1300行，支持多种命令行参数。这个实验里实现最简单的就行。大家可以先玩一下Linux的 `pstree` ，它真的很简单。

总览

pstree [OPTION]...

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，--show-pids`：打印每个进程的进程号。

`-n  --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  --version`：打印版本信息。

在命令行用这些参数执行 `pstree` (如 `pstree -V`、 `pstree --show-pids`)参考它们的行为。

解释

上述实验要求描述是参照man page的格式写出的，但其中有很多UNIX命令行工具遵守的共同约定(大家都默认，所以给初学者学习造成了很大的困扰)，例如POSIX (http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html)就对命令行参数有一定的约定，从中摘抄一些约定如下：

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. 同一个选项可以有不同的名字。在`pstree`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，则其合法的范围是是0到系统最大支持的文件大小(以后会遇到有数值的参数)。

此外，`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`用1表示比较的文件不同，而用2表示读取文件失败(`cmp`的行为也应当如此)。UNIX Shell对返回值有额外的处理 (http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html)。这解释了为什么一些OJ会明确要求main函数返回值为0，当返回非0时将被认为是Runtime Error。

如果不知道这些约定，就会对使用Linux/Unix感到十分困难——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，这套设计简洁好用(但也有很多人批评它不够优雅)。如果你的目标是用最短的时间把事情搞定，用Shell和各种命令行工具的组合一定是你的第一选择，记住：*Keep it simple, stupid*和*Everything is a file*.

评分标准

Minilab的唯一评分标准为正确性：

- 需要打印出正确的进程树(格式不限，可以简陋)，其中包含操作系统中的进

程。可以和 pstree 命令进行对比，允许任何格式；允许进程列表有轻微出入。

- 在32bit和64bit平台上同时运行通过。为了通过编译，你需要在你的环境上安装额外的运行库。

把大象装进冰箱

想自己尝试?

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。当然还是希望在完成实验以后把讲义的后半部分读完，毕竟写东西很费时间的，没准还有惊喜呢。

> “把大象放进冰箱总共分几步？”
>
> “三步，第一步把冰箱门打开；第二步把大象放进去，第三步把冰箱门带上。”
>
> — 赵本山、宋丹丹《钟点工》

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。

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**。

1. 命令行参数

获取命令行参数的一小段代码：

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

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

可以在终端里试试给这个程序传入不同的参数会输出什么，获取命令行参数的。这个问题就算是很好地解决啦： `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。在ICS PA中我们已经知道getopt (`man 3 getopt`)可以处理命令行参数，不过如果你想实际体验一下编程，你也可以自己动手实现getopt的功能。

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。比较常见的项目组织是编写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。

提示

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？不靠谱的人每次都键入命令(或者他发现Ctrl-p可以重复命令)。

- 聪明一些的人发现，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。
- 更聪明的人发现可以写个Makefile (就像这个实验一样)，用 `make test` 跑完所有测试。
- 更聪明的人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` 。

我不否认在编写大项目的时候IDE更好用，不过在写小项目的时候能得到一点即时的反馈也是很不错的，尤其是能弄清背后的道理。就帮你们到这里啦！UNIX哲学是无处不在的，说得更具体一点，“只要你敢想，就一定能做到”。祝大家编程愉快。

最后，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**。

2. 得到系统中进程的编号

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？Google一把就知道啦。你能找到glibc对它的官方文档解释 (https://www.gnu.org/software/libc/manual/html_node/Process-Identification.html)。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？之前已经提示过了：

> Everything is a file.

一切皆文件，进程信息当然也是“一切”的一部分。Linux提供了procfs (https://en.wikipedia.org/wiki/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，这些就是操作系统里的进程啦！

可以先玩一会儿，用 `cat` 可以打印文件的值，可以把文件的内容打出来看看，再对照文档。

现在的问题就变成：怎样得到 `/proc` 目录下的所有以数字为开头的目录。如果你找对关键字，会发现有很多种方法，一定要自己试一试哦。

3. 得到进程之间的关系

procfs里还有很多有趣的东西，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。还有，你们的OJ就是读取 `/proc/[pid]/` 中的信息得到时间/内存等统计信息的。

就像一个普通的文件一样，你可以用你熟悉的方式打开它，把你要的信息读出来就行了：

```
FILE *fp = fopen(filename, "r");
if (fp) {
  // 用fscanf, fgets等函数读取
  fclose(fp);
} else {
  // 错误处理
}
```

procfs里的信息足够让你写一个非常不错的任务管理器。

4. 建树和打印

这是数据结构方面的内容，这门课不会涉及啦。把它当一个OJ题就好了。如果你没有头绪，试着定义一个递归过程：

1. 定义$f(T) \to [s_1, s_2, \dots, s_n]$把$T$打印成若干行(第$i$行是字符串$s_i$)。
2. 求$f(T)$时，先把它所有子树的字符串序列求出来。
3. 把这些字符串拼到适当的位置，加上一些连接线。

然后你会发现你并不需要真的实现$f(T)$，而是一遍递归一边打印就行。

写出正确的代码

完成了?

是时候问问自己：我的程序对吗？

提示：助教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？程序是否有出错的可能？它会不会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) (https://en.wikipedia.org/wiki/Undefined_behavior)导致的——UB没把所有的文件都删掉真是谢天谢地了。
2. 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……？
3. 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(前一半是旧信息、新一半是新信息)？
4. 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，到已经被形式化验证的编译器，都有数不清的bug。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。我个人热切盼望着没有bug的那一天的到来，不过似乎遥不可及。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，更重要的是正确的思维方式(“世界观”)：操作系统会提供什么、该提供什么、不该提供什么、应该怎么提供。