Lecture 07: Debugging and Profiling

Debugging

Printf debugging and Logging

A first approach to debug a program is to add print statements around where you have detected the problem, and keep iterating.

A second approach is to use logging in your program, instead of ad hoc print statements. Logging is better than regular print statements.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/6535ac51-ca3c-4637-b8f0-e1dcc57ab6c5/logger.py

```
$ python logger.py
# Raw output as with just prints
$ python logger.py log
# Log formatted output
$ python logger.py log ERROR
# Print only ERROR levels and above
$ python logger.py color
# Color formatted output
```

Programs like `ls` or `grep` are using ANSI escape codes, which are special sequences of characters to indicate your shell to change the color of the output. For example, executing `echo -e "\e[38;2;255;0;0mThis is red\e[0m"` prints the message This is red in red on your terminal.

```
#!/usr/bin/env bash
for R in $(seq 0 20 255); do
    for G in $(seq 0 20 255); do
        for B in $(seq 0 20 255); do
            printf "\e[38;2;${R};${G};${B}m\e[0m";
        done
    done
done
```

Third party logs

In UNIX systems, it is commonplace for programs to write their logs under `/var/log`. For instance, the NGINX webserver places its logs under `/var/log/nginx`.

Most (but not all) Linux systems use `systemd`, a system daemon that controls many things in your system such as which services are enabled and running. `systemd` places the logs under `/var/log/journal` in a specialized format and you can use the `journalctl` command to display the messages.

On macOS there is still `/var/log/system.log` but an increasing number of tools use the system log, that can be displayed with `log show`. On most UNIX systems you can also use the `dmesg` command to access the kernel log.

For logging under the system logs you can use the `logger` shell program.

Logs can be quite verbose and they require some level of processing and filtering to get the information you want. There are also some tools like `lnav`, that provide an improved presentation and navigation for log files.

Debuggers

Many programming languages come with some form of debugger. In Python this is the Python Debugger `pdb`.

- **l**(ist) - Displays 11 lines around the current line or continue the previous listing.
- **s**(tep) - Execute the current line, stop at the first possible occasion.
- **n**(ext) - Continue execution until the next line in the current function is reached or it returns.
- **b**(reak) - Set a breakpoint (depending on the argument provided).
- **p**(rint) - Evaluate the expression in the current context and print its value. There's also **pp** to display using `pprint` instead.
- **r**(eturn) - Continue execution until the current function returns.
- **q**(uit) - Quit the debugger.

Note that since Python is an interpreted language we can use the `pdb` shell to execute commands and to execute instructions. `ipdb` is an improved `pdb` that uses the `IPython` REPL enabling tab completion, syntax highlighting, better tracebacks, and better introspection while retaining the same interface as the `pdb` module.

For more low level programming you will probably want to look into `gdb` (and its quality of @life modification `pwndbg`) and `lldb`.

Specialized Tools

Whenever programs need to perform actions that only the kernel can, they use System Calls. There are commands that let you trace the syscalls your program makes. In Linux there's `strace` and macOS and BSD have `dtrace`.

Below are some examples of using `strace` or `dtruss` to show `stat` syscall traces for an execution of `ls`. For a deeper dive into `strace`, this is a good read.

```
# On Linux
sudo strace -e lstat ls -l > /dev/null
4
# On macOS
sudo dtruss -t lstat64_extended ls -l > /dev/null
```

Tools like `tcpdump` and Wireshark are network packet analyzers that let you read the contents of network packets and filter them based on different criteria.

Static Analysis

Static analysis programs take source code as input and analyze it using coding rules to reason about its correctness.

Static analysis tools can identify this kind of issues. When we run `pyflakes` on the code we get the errors related to both bugs. `mypy` is another tool that can detect type checking issues. In the shell tools lecture we covered `shellcheck`, which is a similar tool for shell scripts.

```
$ pyflakes foobar.py
foobar.py:6: redefinition of unused 'foo' from line 3
foobar.py:11: undefined name 'baz'

$ mypy foobar.py
foobar.py:6: error: Incompatible types in assignment (expression has type "int", variable has type "Callable[[], Any]")
foobar.py:9: error: Incompatible types in assignment (expression has type "float", variable has type "int")
foobar.py:11: error: Name 'baz' is not defined
Found 3 errors in 1 file (checked 1 source file)
```

In vim, the plugins `ale` or `syntastic` will let you do that. For Python, `pylint` and `pep8` are examples of stylistic linters and `bandit` is a tool designed to find common security issues.

Profiling

Premature optimization is the root of all evil

Timing

Wall clock time can be misleading since your computer might be running other processes at the same time or waiting for events to happen. It is common for tools to make a distinction between *Real*, *User* and *Sys* time. In general, *User* + *Sys* tells you how much time your process actually spent in the CPU (more detailed explanation here).

- *Real* - Wall clock elapsed time from start to finish of the program, including the time taken by other processes and time taken while blocked (e.g. waiting for I/O or network)
- *User* - Amount of time spent in the CPU running user code
- *Sys* - Amount of time spent in the CPU running kernel code

Profilers

CPU

There are two main types of CPU profilers: *tracing* and *sampling* profilers.

Tracing profilers keep a record of every function call your program makes whereas sampling profilers probe your program periodically (commonly every millisecond) and record the program's stack. Here is a good intro article if you want more detail on this topic.

In Python we can use the `cProfile` module to profile time per function call.

```
$ python -m cProfile -s tottime grep.py 1000 '^(import|\s*def)[^,]*$' *.py

[omitted program output]

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     8000    0.266    0.000    0.292    0.000 {built-in method io.open}
     8000    0.153    0.000    0.894    0.000 grep.py:5(grep)
    17000    0.101    0.000    0.101    0.000 {built-in method builtins.print}
     8000    0.100    0.000    0.129    0.000 {method 'readlines' of '_io._IOBase' objects}
    93000    0.097    0.000    0.111    0.000 re.py:286(_compile)
    93000    0.069    0.000    0.069    0.000 {method 'search' of '_sre.SRE_Pattern' objects}
    93000    0.030    0.000    0.141    0.000 re.py:231(compile)
    17000    0.019    0.000    0.029    0.000 codecs.py:318(decode)
        1    0.017    0.017    0.911    0.911 grep.py:3(<module>)

[omitted lines]
```

A caveat of Python's `cProfile` profiler (and many profilers for that matter) is that they display time per function call. A more intuitive way of displaying profiling information is to include the time taken per line of code, which is what line *profilers* do.

A quick run with `line_profiler` shows the time taken per line:

```
$ kernprof -l -v a.py
Wrote profile results to urls.py.lprof
Timer unit: 1e-06 s

Total time: 0.636188 s
File: a.py
Function: get_urls at line 5

Line #  Hits         Time  Per Hit   % Time  Line Contents
==============================================================
 5                                           @profile
 6                                           def get_urls():
 7         1     613909.0 613909.0     96.5      response = requests.get('https://missing.csail.mit.edu')
 8         1      21559.0  21559.0      3.4      s = BeautifulSoup(response.content, 'lxml')
 9         1          2.0      2.0      0.0      urls = []
10        25        685.0     27.4      0.1      for url in s.find_all('a'):
11        24         33.0      1.4      0.0          urls.append(url['href'])
```

Memory

To help in the process of memory debugging you can use tools like Valgrind that will help you identify memory leaks in languages like C or C++.

In garbage collected languages like Python it is still useful to use a memory profiler because as long as you have pointers to objects in memory they won't be garbage collected.

Here's an example program and its associated output when running it with memory-profiler:

```
$ python -m memory_profiler example.py
Line #    Mem usage  Increment   Line Contents
================================================
     3                           @profile
     4      5.97 MB    0.00 MB   def my_func():
     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)
     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)
     7     13.61 MB -152.59 MB       del b
     8     13.61 MB    0.00 MB       return a
```

Event Profiling

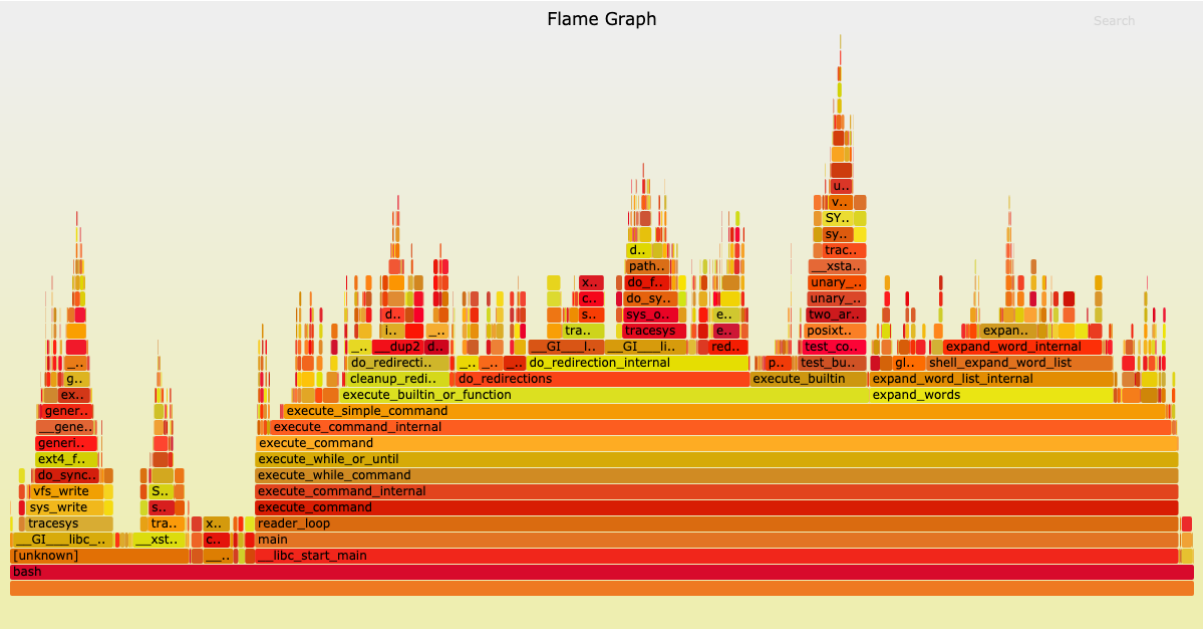
The `perf` command abstracts CPU differences away and does not report time or memory, but instead it reports system events related to your programs. For example, `perf` can easily report poor cache locality, high amounts of page faults or livelocks. Here is an overview of the command:

- `perf list` - List the events that can be traced with perf

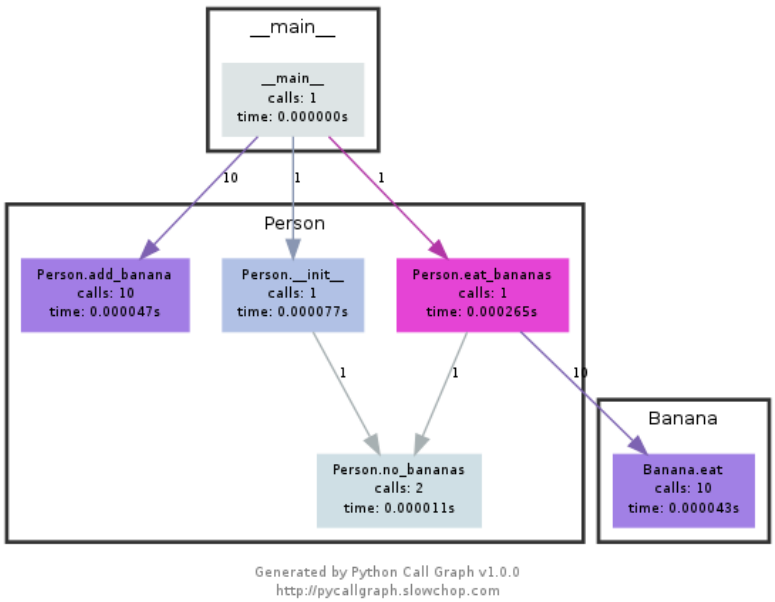
- `perf stat COMMAND ARG1 ARG2` - Gets counts of different events related a process or command
- `perf record COMMAND ARG1 ARG2` - Records the run of a command and saves the statistical data into a file called `perf.data`
- `perf report` - Formats and prints the data collected in `perf.data`

Visualization

One common way to display CPU profiling information for sampling profilers is to use a Flame Graph, which will display a hierarchy of function calls across the Y axis and time taken proportional to the X axis.



Call graphs or control flow graphs display the relationships between subroutines within a program by including functions as nodes and functions calls between them as directed edges. In Python you can use the `pycallgraph` library to generate them.



Resource Monitoring

There are a myriad of command line tools for probing and displaying different system resources like CPU usage, memory usage, network, disk usage and so on.

- **General Monitoring** - Probably the most popular is `htop`, which is an improved version of `top`. `htop` presents various statistics for the currently running processes on the system. `htop` has a myriad of options and keybinds, some useful ones are: `<F6>` to sort processes, `t` to show tree hierarchy and `h` to toggle threads. See also `glances` for similar implementation with a great UI. For getting aggregate measures across all processes, `dstat` is another nifty tool that computes real-time resource metrics for lots of different subsystems like I/O, networking, CPU utilization, context switches, &c.
- **I/O operations** - `iotop` displays live I/O usage information and is handy to check if a process is doing heavy I/O disk operations
- **Disk Usage** - `df` displays metrics per partitions and `du` displays **d**isk **u**sage per file for the current directory. In these tools the `-h` flag tells the program to print with **h**uman readable format. A more interactive version of `du` is `ncdu` which lets you navigate folders and delete files and folders as you navigate.
- **Memory Usage** - `free` displays the total amount of free and used memory in the system. Memory is also displayed in tools like `htop`.
- **Open Files** - `lsof` lists file information about files opened by processes. It can be quite useful for checking which process has opened a specific file.
- **Network Connections and Config** - `ss` lets you monitor incoming and outgoing network packets statistics as well as interface statistics. A common use case of `ss` is figuring out what process is using a given port in a machine. For displaying routing, network devices and interfaces you can use `ip`. Note that `netstat` and `ifconfig` have been deprecated in favor of the former tools respectively.
- **Network Usage** - `nethogs` and `iftop` are good interactive CLI tools for monitoring network usage.

If you want to test these tools you can also artificially impose loads on the machine using the `stress` command.

Specialized tools

Tools like `hyperfine` let you quickly benchmark command line programs. We can use hyperfine to compare command `fd` and `find`:

```
$ hyperfine --warmup 3 'fd -e jpg' 'find . -iname "*.jpg"'
Benchmark #1: fd -e jpg
  Time (mean ± σ):      51.4 ms ±   2.9 ms    [User: 121.0 ms, System: 160.5 ms]
  Range (min … max):    44.2 ms …  60.1 ms    56 runs

Benchmark #2: find . -iname "*.jpg"
  Time (mean ± σ):      1.126 s ±  0.101 s    [User: 141.1 ms, System: 956.1 ms]
  Range (min … max):    0.975 s …  1.287 s    10 runs

Summary
  'fd -e jpg' ran
   21.89 ± 2.33 times faster than 'find . -iname "*.jpg"'
```

Others

spiside/pdb-tutorial

The purpose of this tutorial is to teach you the basics of pdb, the Python De Bugger for Python2 and Python3. It will also include some helpful tricks to make your debugging sessions a lot less stressful.

https://github.com/spiside/pdb-tutorial

Python Debugging With Pdb - Real Python

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: Python Debugging With pdb Debugging applications can sometimes be an unwelcome activity. You're busy working under a

https://realpython.com/python-debugging-pdb/