Debugging and Profiling

A golden rule in programming is that code does not do what you expect it to do, but what you tell it to do. Bridging that gap can sometimes be a quite difficult feat. In this lecture we are going to cover useful techniques for dealing with buggy and resource hungry code: debugging and profiling.

Debugging

Printf debugging and Logging

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements" — Brian Kernighan, *Unix for Beginners*.

A first approach to debug a program is to add print statements around where you have detected the problem, and keep iterating until you have extracted enough information to understand what is responsible for the issue.

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

- You can log to files, sockets or even remote servers instead of standard output.
- Logging supports severity levels (such as INFO, DEBUG, WARN, ERROR, &c), that allow you to filter the output accordingly.
- For new issues, there's a fair chance that your logs will contain enough information to detect what is going wrong.

Here is an example code that logs messages:

```
$ 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
```

One of my favorite tips for making logs more readable is to color code them. By now you probably have realized that your terminal uses colors to make things more readable. But how does it do it? 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, as long as it supports true color. If your

terminal doesn't support this (e.g. macOS's Terminal.app), you can use the more universally supported escape codes for 16 color choices, for example `echo -e "\e[31;1mThis is red\e[0m"`.

The following script shows how to print many RGB colors into your terminal (again, as long as it supports true color).

```
#!/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

As you start building larger software systems you will most probably run into dependencies that run as separate programs. Web servers, databases or message brokers are common examples of this kind of dependencies. When interacting with these systems it is often necessary to read their logs, since client side error messages might not suffice.

Luckily, most programs write their own logs somewhere in your system. 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`. More recently, systems have started using a **system log**, which is increasingly where all of your log messages go. 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. Similarly, 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. Here's an example of using `logger` and how to check that the entry made it to the system logs. Moreover, most programming languages have bindings logging to the system log.

```
logger "Hello Logs"
# On macOS
log show --last 1m | grep Hello
# On Linux
journalctl --since "1m ago" | grep Hello
```

As we saw in the data wrangling lecture, logs can be quite verbose and they require some level of processing and filtering to get the information you want. If you find yourself heavily filtering through `journalctl` and `log show` you can consider using their flags, which can perform a first pass of filtering of their output. There are also some tools like `lnav`, that provide an improved presentation and navigation for log files.

Debuggers

When printf debugging is not enough you should use a debugger. Debuggers are programs that let you interact with the execution of a program, allowing the following:

- Halt execution of the program when it reaches a certain line.
- Step through the program one instruction at a time.
- Inspect values of variables after the program crashed.
- Conditionally halt the execution when a given condition is met.
- And many more advanced features

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

Here is a brief description of some of the commands `pdb` supports:

- **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. p locals(): return all values
- **r**(eturn) – Continue execution until the current function returns.
- **q**(uit) – Quit the debugger.

Let's go through an example of using `pdb` to fix the following buggy python code. (See the lecture video).

python -m ipdb bubble.py

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n):
            if arr[j] > arr[j+1]:
                arr[j] = arr[j+1]
                arr[j+1] = arr[j]
    return arr

print(bubble_sort([4, 2, 1, 8, 7, 6]))
```

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`. They are optimized for C–like language debugging but will let you probe pretty much any process and get its current machine state: registers, stack, program counter, &c.

gdb --args sleep 20
run

Specialized Tools

Even if what you are trying to debug is a black box binary there are tools that can help you with that. 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`. `dtrace` can be tricky to use because it uses its own `D` language, but there is a wrapper called `dtruss` that provides an interface more similar to `strace` (more details here).

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 article and this zine are good reads.

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

Under some circumstances, you may need to look at the network packets to figure out the issue in your program. 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.

For web development, the Chrome/Firefox developer tools are quite handy. They feature a large number of tools, including:

- Source code – Inspect the HTML/CSS/JS source code of any website.
- Live HTML, CSS, JS modification – Change the website content, styles and behavior to test (you can see for yourself that website screenshots are not valid proofs).
- Javascript shell – Execute commands in the JS REPL.
- Network – Analyze the requests timeline.
- Storage – Look into the Cookies and local application storage.

Static Analysis

For some issues you do not need to run any code. For example, just by carefully looking at a piece of code you could realize that your loop variable is shadowing an already existing variable or function name; or that a program reads a variable before defining it. Here is where static analysis tools come into play. Static analysis programs take source code as input and analyze it using coding rules to reason about its correctness.

In the following Python snippet there are several mistakes. First, our loop variable `foo` shadows the previous definition of the function `foo`. We also wrote `baz` instead of `bar` in the last line, so the program will crash after completing the `sleep` call (which will take one minute).

```
import time

def foo():
    return 42

for foo in range(5):
    print(foo)
bar = 1
bar *= 0.2
time.sleep(60)
print(baz)
```

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. Here, `mypy` will warn us that `bar` is initially an `int` and is then casted to a `float`. Again, note that all these issues were detected without having to run the code.

```
$ 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
foobar.py:9: error: Incompatible types in assignment (expression has type
foobar.py:11: error: Name 'baz' is not defined
Found 3 errors in 1 file (checked 1 source file)
```

In the shell tools lecture we covered `shellcheck`, which is a similar tool for shell scripts.

Most editors and IDEs support displaying the output of these tools within the editor itself, highlighting the locations of warnings and errors. This is often called **code linting** and it can also be used to display other types of issues such as stylistic violations or insecure constructs.

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. For other languages people have compiled comprehensive lists of useful static analysis tools, such as Awesome Static Analysis (you may want to take a look at the *Writing* section) and for linters there is Awesome Linters.

A complementary tool to stylistic linting are code formatters such as `black` for Python, `gofmt` for Go, `rustfmt` for Rust or `prettier` for JavaScript, HTML and CSS. These tools autoformat your code so that it's consistent with common stylistic patterns for the given programming language. Although you might be unwilling to give stylistic control about your code, standardizing code format will help other people read your code and will make you better at reading other people's (stylistically standardized) code.

Profiling

Even if your code functionally behaves as you would expect, that might not be good enough if it takes all your CPU or memory in the process. Algorithms classes often teach big *O* notation but not how to find hot spots in your programs. Since premature optimization is the root of all evil, you should learn about profilers and monitoring tools. They will help you understand which parts of your program are taking most of the time and/or resources so you can focus on optimizing those parts.

Timing

Similarly to the debugging case, in many scenarios it can be enough to just print the time it took your code between two points. Here is an example in Python using the `time` module.

```
import time, random
n = random.randint(1, 10) * 100

# Get current time
start = time.time()

# Do some work
print("Sleeping for {} ms".format(n))
time.sleep(n/1000)

# Compute time between start and now
print(time.time() - start)

# Output
# Sleeping for 500 ms
# 0.5713930130004883
```

However, 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

For example, try running a command that performs an HTTP request and prefixing it with `time`. Under a slow connection you might get an output like the one below. Here it took over 2 seconds for the request to complete but the process only took 15ms of CPU user time and 12ms of kernel CPU time.

```
$ time curl https://missing.csail.mit.edu &> /dev/null
real    0m2.561s
user    0m0.015s
sys     0m0.012s
```

Profilers

CPU

Most of the time when people refer to *profilers* they actually mean *CPU profilers*, which are the most common. 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. They use these records to present aggregate statistics of what your program spent the most time doing. Here is a good intro article if you want more detail on this topic.

Most programming languages have some sort of command line profiler that you can use to analyze your code. They often integrate with full fledged IDEs but for this lecture we are going to focus on the command line tools themselves.

In Python we can use the `cProfile` module to profile time per function call. Here is a simple example that implements a rudimentary grep in Python:

```python
#!/usr/bin/env python

import sys, re

def grep(pattern, file):
    with open(file, 'r') as f:
        print(file)
        for i, line in enumerate(f.readlines()):
            pattern = re.compile(pattern)
            match = pattern.search(line)
            if match is not None:
                print("{}: {}".format(i, line), end="")

if __name__ == '__main__':
    times = int(sys.argv[1])
    pattern = sys.argv[2]
    for i in range(times):
        for file in sys.argv[3:]:
            grep(pattern, file)
```

We can profile this code using the following command. Analyzing the output we can see that IO is taking most of the time and that compiling the regex takes a fair amount of time as well. Since the regex only needs to be compiled once, we can factor it out of the for.

```
$ 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.p
     8000    0.100    0.000    0.129    0.000 {method 'readlines' of '_io
    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.S
    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. That can become unintuitive really fast, especially if you are using third party libraries in your code since internal function calls are also accounted for. A more intuitive way of displaying profiling information is to include the time taken per line of code, which is what *line profilers* do.

For instance, the following piece of Python code performs a request to the class website and parses the response to get all URLs in the page:

```
#!/usr/bin/env python
import requests
from bs4 import BeautifulSoup

# This is a decorator that tells line_profiler
# that we want to analyze this function
@profile
def get_urls():
    response = requests.get('https://missing.csail.mit.edu')
    s = BeautifulSoup(response.content, 'lxml')
    urls = []
    for url in s.find_all('a'):
        urls.append(url['href'])

if __name__ == '__main__':
    get_urls()
```

If we used Python's `cProfile` profiler we'd get over 2500 lines of output, and even with sorting it'd be hard to understand where the time is being spent. 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(
 8         1      21559.0  21559.0      3.4      s = BeautifulSoup(respon
 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['hre
```

Memory

In languages like C or C++ memory leaks can cause your program to never release memory that it doesn't need anymore. To help in the process of memory debugging you can use

tools like Valgrind that will help you identify memory leaks.

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 (note the decorator like in `line-profiler`).

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

```
$ 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

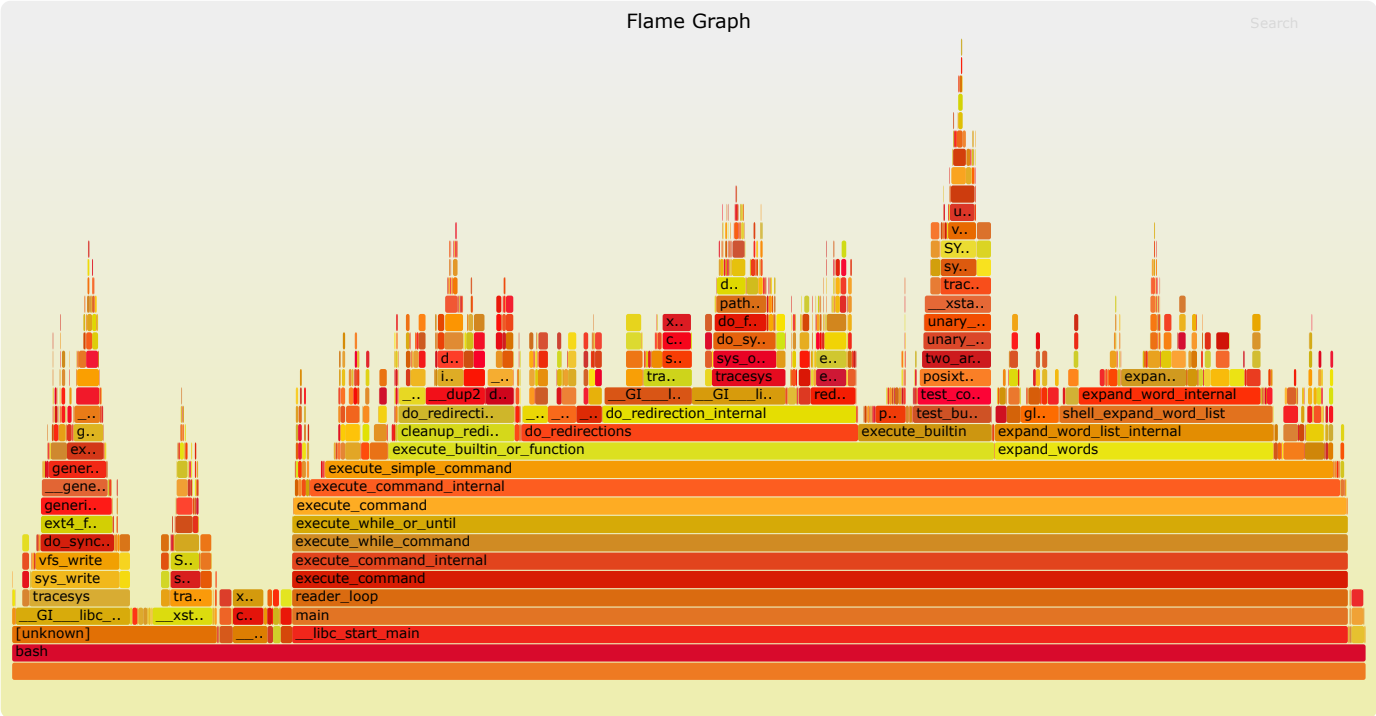
As it was the case for `strace` for debugging, you might want to ignore the specifics of the code that you are running and treat it like a black box when profiling. The `perf` command abstracts CPU differences away and does not report time or memory, but instead it reports system events related to your 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`

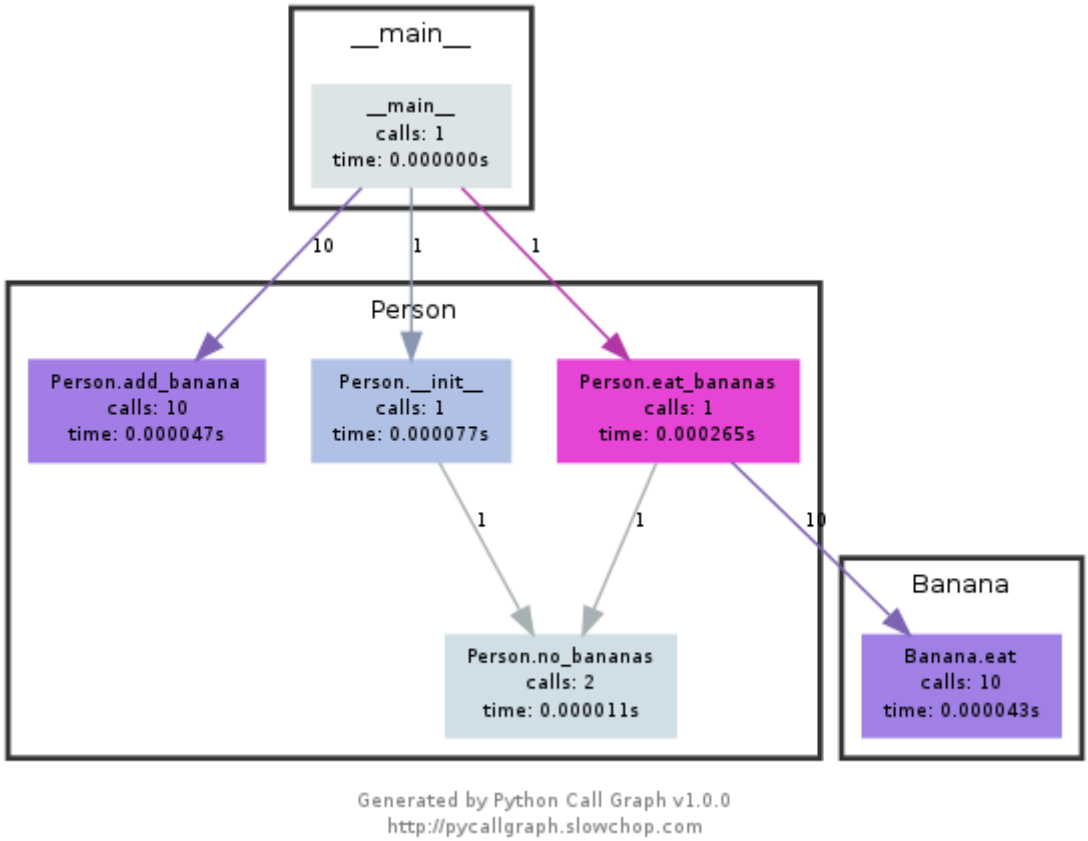
Visualization

Profiler output for real world programs will contain large amounts of information because of the inherent complexity of software projects. Humans are visual creatures and are quite terrible at reading large amounts of numbers and making sense of them. Thus there are many tools for displaying profiler's output in an easier to parse way.

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. They are also interactive, letting you zoom into specific parts of the program and get their stack traces (try clicking in the image below).



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. When coupled with profiling information such as the number of calls and time taken, call graphs can be quite useful for interpreting the flow of a program. In Python you can use the `pycallgraph` library to generate them.



Resource Monitoring

Sometimes, the first step towards analyzing the performance of your program is to understand what its actual resource consumption is. Programs often run slowly when they are resource constrained, e.g. without enough memory or on a slow network connection. 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.

sudo lsof | grep ":8080 . LISTEN"

- **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

Sometimes, black box benchmarking is all you need to determine what software to use. Tools like `hyperfine` let you quickly benchmark command line programs. For instance, in the shell tools and scripting lecture we recommended `fd` over `find`. We can use `hyperfine` to compare them in tasks we run often. E.g. in the example below `fd` was 20x faster than `find` in my machine.

```
$ 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: 16
  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: 95
  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"'
```

As it was the case for debugging, browsers also come with a fantastic set of tools for profiling webpage loading, letting you figure out where time is being spent (loading, rendering, scripting, &c). More info for Firefox and Chrome.

Exercises

Debugging

1. Use `journalctl` on Linux or `log show` on macOS to get the super user accesses and commands in the last day. If there aren't any you can execute some harmless commands such as `sudo ls` and check again.

2. Do this hands on `pdb` tutorial to familiarize yourself with the commands. For a more in depth tutorial read this.

3. Install shellcheck and try checking the following script. What is wrong with the code? Fix it. Install a linter plugin in your editor so you can get your warnings automatically.

```
#!/bin/sh
## Example: a typical script with several problems
for f in $(ls *.m3u)
do
  grep -qi hq.*mp3 $f \
    && echo -e 'Playlist $f contains a HQ file in mp3 format'
done
```

4. (Advanced) Read about reversible debugging and get a simple example working using rr or RevPDB.

Profiling

5. Here are some sorting algorithm implementations. Use cProfile and line_profiler to compare the runtime of insertion sort and quicksort. What is the bottleneck of each algorithm? Use then `memory_profiler` to check the memory consumption, why is insertion sort better? Check now the inplace version of quicksort. Challenge: Use `perf` to look at the cycle counts and cache hits and misses of each algorithm.

6. Here's some (arguably convoluted) Python code for computing Fibonacci numbers using a function for each number.

```
#!/usr/bin/env python
def fib0(): return 0

def fib1(): return 1

s = """def fib{}(): return fib{}() + fib{}()"""

if __name__ == '__main__':

    for n in range(2, 10):
        exec(s.format(n, n-1, n-2))
    # from functools import lru_cache
    # for n in range(10):
    #     exec("fib{} = lru_cache(1)(fib{})".format(n, n))
    print(eval("fib9()"))
```

Put the code into a file and make it executable. Install prerequisites: pycallgraph and graphviz. (If you can run `dot`, you already have GraphViz.) Run the code as is with `pycallgraph graphviz -- ./fib.py` and check the `pycallgraph.png` file. How many times is `fib0` called?. We can do better than that by memoizing the functions.

Uncomment the commented lines and regenerate the images. How many times are we calling each `fibN` function now?

7. A common issue is that a port you want to listen on is already taken by another process. Let's learn how to discover that process pid. First execute `python -m http.server 4444` to start a minimal web server listening on port `4444`. On a separate terminal run `lsof | grep LISTEN` to print all listening processes and ports. Find that process pid and terminate it by running `kill <PID>`.

8. Limiting processes resources can be another handy tool in your toolbox. Try running `stress -c 3` and visualize the CPU consumption with `htop`. Now, execute `taskset --cpu-list 0,2 stress -c 3` and visualize it. Is `stress` taking three CPUs? Why not? Read `man taskset`. Challenge: achieve the same using `cgroups`. Try limiting the memory consumption of `stress -m`.

9. (Advanced) The command `curl ipinfo.io` performs a HTTP request and fetches information about your public IP. Open Wireshark and try to sniff the request and reply packets that `curl` sent and received. (Hint: Use the `http` filter to just watch HTTP packets).