LECTURE 17

MORE CACHES AND PIPELINES

- recall that memory is virtualized
- a virtual address $\longrightarrow$ hardware address translation is necessary for every memory access
- the translation uses a page table
- the page table is stored in memory
- hence two effective memory accesses per memory fetch instruction?

Solution: part of the page table is **cached** in the CPU

the “Translation Lookaside Buffer” (TLB)

Caches are used at various levels to hide access latency

	typical latency		
instruction	~0.25 ns	0.25	ns
RAM	~100 ns	100	ns
solid state drive (SSD)	~0.2 ms	200,000	ns
hard disk drive (HDD)	~2 ms	2,000,000	ns
wired ethernet (round-trip)	~1 ms	1,000,000	ns
wifi latency (round-trip)	~10 ms	10,000,000	ns
same-city internet (round-trip)	~5 ms	5,000,000	ns
same-continent internet (round-trip)	~25 ms	25,000,000	ns
transatlantic internet (round-trip)	~100 ms	100,000,000	ns

Examples of caches

- SSDs have internal RAM caches (typically 0-4 GB)
- the operating system caches files in memory
- large content provides (Google, Amazon, Netflix, Cloudflare) have caches all over the world

```
ping canada.ca
```

```
PING canada.ca (205.193.117.159) 56(84) bytes of data.
64 bytes from 205.193.117.159 (205.193.117.159): icmp_seq=1 ttl=228 time=200 ms
64 bytes from 205.193.117.159 (205.193.117.159): icmp_seq=2 ttl=228 time=172 ms
64 bytes from 205.193.117.159 (205.193.117.159): icmp_seq=3 ttl=228 time=148 ms
64 bytes from 205.193.117.159 (205.193.117.159): icmp_seq=4 ttl=228 time=181 ms
```

```
ping google.com.au
```

```
PING google.com.au (142.251.209.3) 56(84) bytes of data.
64 bytes from mil04s50-in-f3.1e100.net (142.251.209.3): icmp_seq=1 ttl=115 time=12.2 ms
64 bytes from mil04s50-in-f3.1e100.net (142.251.209.3): icmp_seq=2 ttl=115 time=14.6 ms
64 bytes from mil04s50-in-f3.1e100.net (142.251.209.3): icmp_seq=3 ttl=115 time=12.9 ms
64 bytes from mil04s50-in-f3.1e100.net (142.251.209.3): icmp_seq=4 ttl=115 time=11.8 ms
```

Examples of pipelines

- Storage devices:
  - SSDs typically access data in “pages” of **4096 bytes**
  - **0.2ms** SSD latency would imply a max speed of 20 MB / s
  - instead SSDs routinely read and write 500 MB / s

- Networks:
  - Network packets are typically **1500 bytes**
  - **10ms** WiFi latency would imply 150 KB / s
  - instead most WiFi networks do at least 10 MB / s

- Browsers:
  - Google Chrome maintains up to **6** connections per domain

BENCHMARKING

```
time ./application
```

```
real    0m2.501s
user    0m2.498s
sys     0m0.001s
```

- `real`: elapsed "real" (wall-clock) time
- `user`: time spent in user mode (running `./application` code)
- `sys`: time spent in system mode (running OS kernel code)
- `user` + `sys` $\lesssim$ `real` (there may be other applications running)

```
time head -n 1000000 /dev/random > /dev/null
```

```
real    0m0.777s
user    0m0.109s
sys     0m0.668s
```

Variance

```
time head -n 1000000 /dev/random > /dev/null
```

```
real    0m2.502s
user    0m2.497s
sys     0m0.003s
```

```
time head -n 1000000 /dev/random > /dev/null
```

```
real    0m2.505s
user    0m2.500s
sys     0m0.002s
```

```
time head -n 1000000 /dev/random > /dev/null
```

```
real    0m2.501s
user    0m2.496s
sys     0m0.001s
```

Reasons for variance

- power and temperature throttling

  (CPU adapts speed to avoid overheating)

- interactions with devices

  (OS has in-memory caches for files, storage devices have internal memory caches)

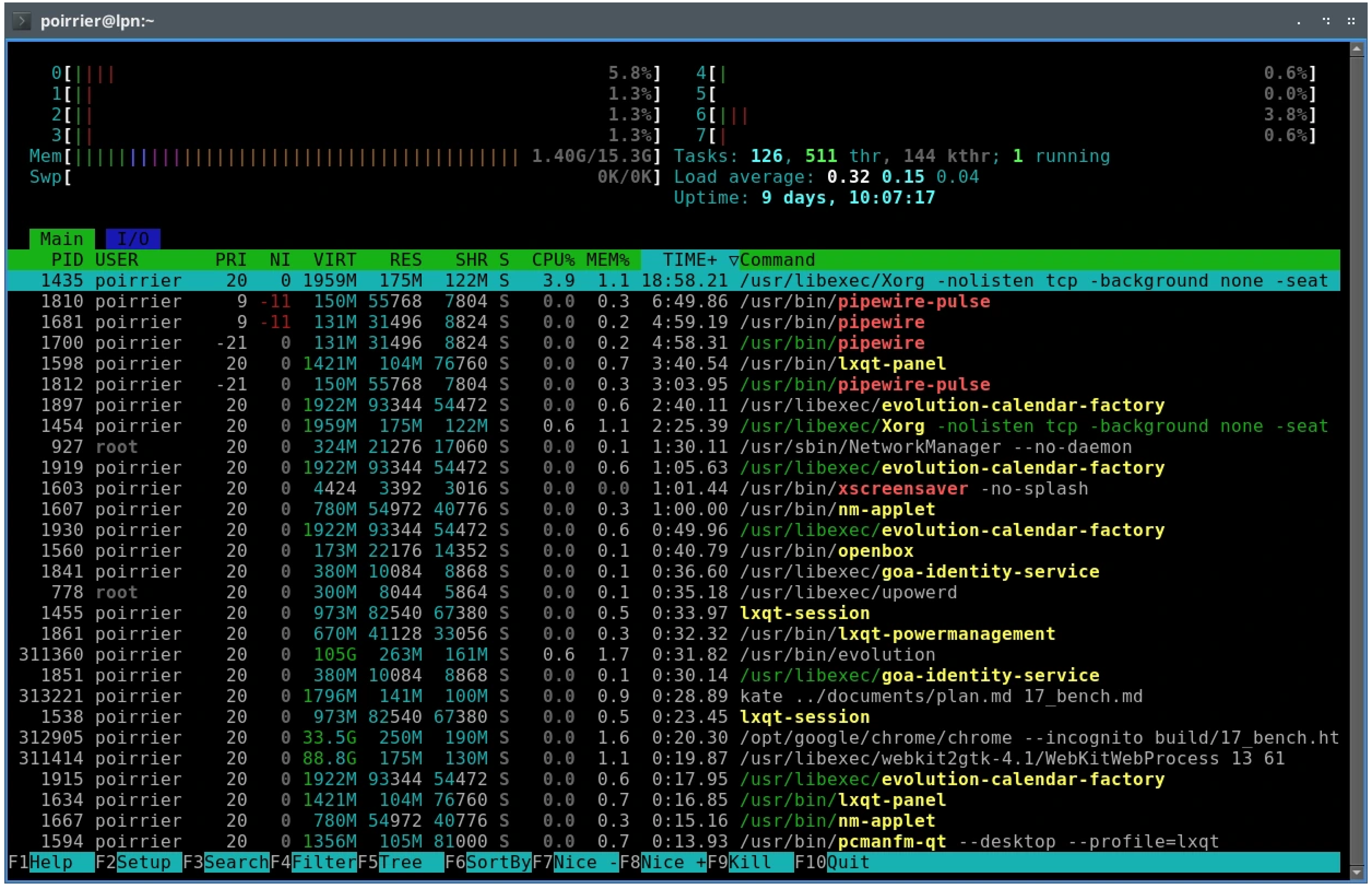
- other processes

  (must share resources)

`top`

`htop`

`ps aux`



(271 processes)

Effect of file caches

```
time md5sum 2GB_file
```

```
860a0023a913fd3fa4b6ad8bfbdd2c62  2GB_file

real	0m5.904s
user	0m4.062s
sys	0m0.560s
```

```
time md5sum 2GB_file
```

```
860a0023a913fd3fa4b6ad8bfbdd2c62  2GB_file

real	0m4.029s
user	0m3.674s
sys	0m0.331s
```

Inaccuracies

- executable startup is slow
- initialization adds overhead
- input and output are slow

Executable startup is slow

```c
int main() { return 0; }
```

```
clang -O3 -o main main.c
time ./main
```

```
real    0m0.003s
user    0m0.000s
sys     0m0.002s
```

```
time python -c 'exit(0)'
```

```
real    0m0.030s
user    0m0.023s
sys     0m0.008s
```

→ we cannot accurately benchmark application that only take a few milliseconds.

Initialization adds overhead

```
time glpsol LP_576x18380.mps
```

```
real    0m0.256s
user    0m0.246s
sys     0m0.010s
```

```
time glpsol --check LP_576x18380.mps
```

```
real    0m0.168s
user    0m0.161s
sys     0m0.008s
```

What are we really measuring?

The speed of the MPS file parser, not the simplex algorithm.

Input and output are slow

```python
def riemann_zeta(s):
    r = 0.0

    for i in range(1, 1000000):
        r = r + 1 / (i ** s)

    return r

# ζ(2) = (pi ** 2) / 6
print('pi ≈ ', (riemann_zeta(2) * 6) ** 0.5)
```

```
time python zeta.py
```

```
pi ≈  3.141591698659554

real	0m0.124s
user	0m0.118s
sys	0m0.006s
```

```python
def riemann_zeta(s):
    r = 0.0

    for i in range(1, 1000000):
        r = r + 1 / (i ** s)
        print('r = ', r)

    return r

# ζ(2) = (pi ** 2) / 6
print('pi ≈ ', (riemann_zeta(2) * 6) ** 0.5)
```

```
time python zeta.py
```

```
r =  1.0
r =  1.25
r =  1.3611111111111112
r =  1.4236111111111112
r =  1.4636111111111112
r =  1.4913888888888889
r =  1.511797052154195

       [...]

r =  1.6449330668467699
r =  1.64493306684777
pi ≈  3.141591698659554

real    0m3.768s
user    0m2.516s
sys     0m0.999s
```

Aggregate measures

- if we benchmark our code on different inputs, we may want to use
  - total time / average time
  - geometric mean
  - or other aggregate measures
  - or some visualization (bar graphs, performance profiles, etc.)

- but beware: all aggregate measures are biased

	Input 1		Input 2		Input 3		Average
Version A	2530s		2300s		12s		1614s
Version B	2535s	1.002x	2304s	1.002x	6s	0.5x	1615s

STATIC INSTRUMENTATION

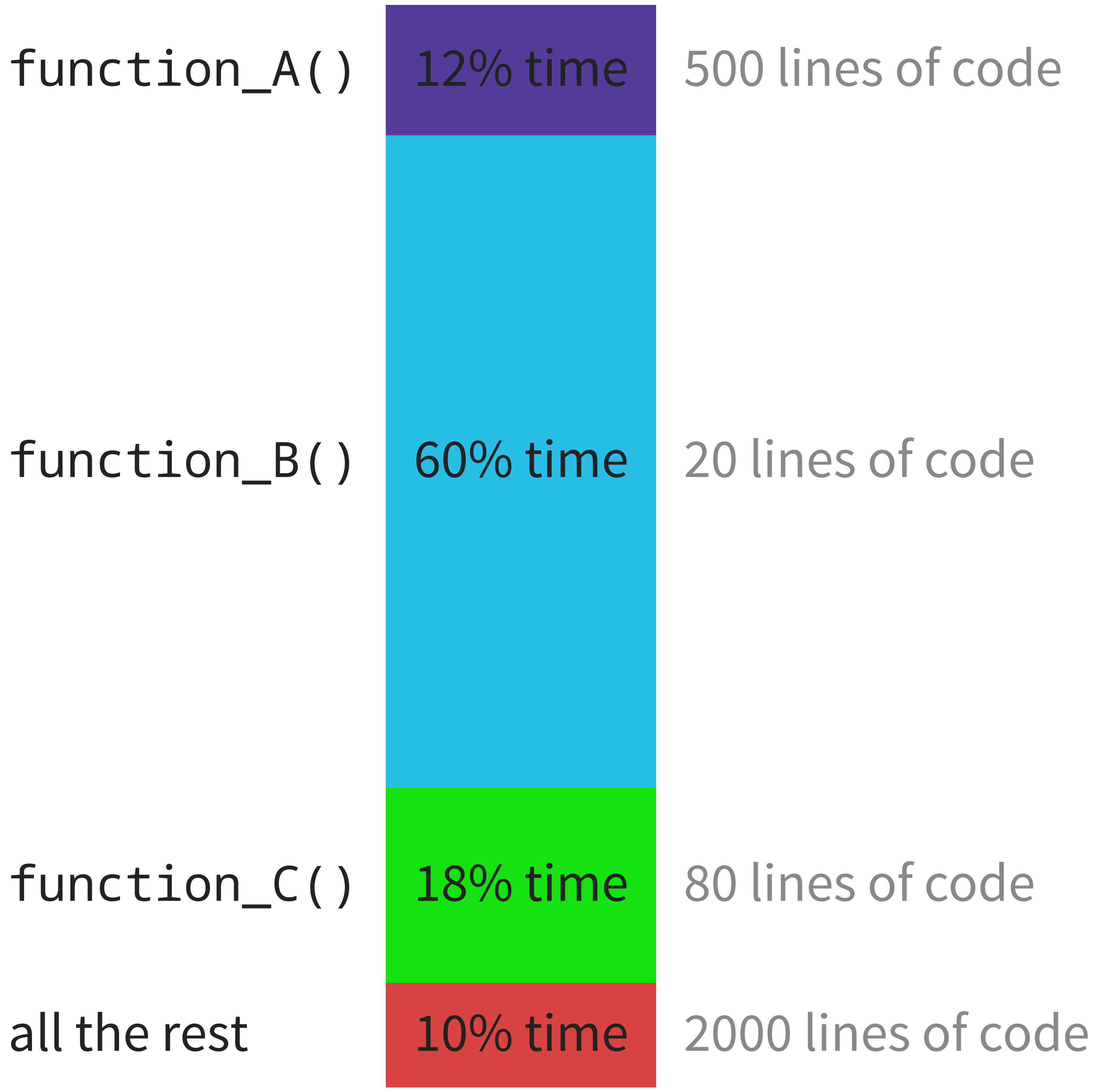
- we may want to benchmark specific parts of our code
  - to circumvent executable startup, initialization, and input/output
  - to benchmark parts of the code that run quickly
  - to find bottlenecks

- for that, we need to add timing instrumentation to our code

About bottlenecks

“Premature optimization is the root of all evil”

– Donald Knuth
“Structured Programming With GoTo Statements”
1974


function_A()
12% time
500 lines of code
function_B()
60% time
20 lines of code
function_C()
18% time
80 lines of code
all the rest
10% time
2000 lines of code

time.time()

```
initialize()
function_A()
function_B()
function_C()
cleanup()
```

```python
import time

t0 = time.time()
initialize()
t1 = time.time()
function_A()
t2 = time.time()
function_B()
t3 = time.time()
function_C()
t4 = time.time()
cleanup()
t5 = time.time()

print(f'total time: {t5 - t0:16.6f}')

print(f'function_A: {t2 - t1:16.6f}')
print(f'function_B: {t3 - t2:16.6f}')
print(f'function_C: {t4 - t3:16.6f}')

print(f'      rest: {(t5 - t0) - (t4 - t1):16.6f}')
```

clock_gettime()

```c
int main()
{
    initialize();
    function_A();
    function_B();
    function_C();
    cleanup();
    return 0;
}
```

```c
int main()
{
    struct timespec t0, t1, t2, t3, t4, t5;

    clock_gettime(CLOCK_MONOTONIC, &t0);
    initialize();
    clock_gettime(CLOCK_MONOTONIC, &t1);
    function_A();
    clock_gettime(CLOCK_MONOTONIC, &t2);
    function_B();
    clock_gettime(CLOCK_MONOTONIC, &t3);
    function_C();
    clock_gettime(CLOCK_MONOTONIC, &t4);
    cleanup();
    clock_gettime(CLOCK_MONOTONIC, &t5);

    print_all_clocks(&t0, &t1, &t2, &t3, &t4, &t5);
    return 0;
}
```

Cumulative time

```
initialize()

for i in range(1000000):
    function_A()
    function_B()
    function_C()

cleanup()
```

```
import time.time

initialize()
tA, tB, tC = 0

for i in range(1000000):
    t0 = time.time()
    function_A()
    t1 = time.time()
    function_B()
    t2 = time.time()
    function_C()
    t3 = time.time()

    tA = tA + (t1 - t0)
    tB = tB + (t2 - t1)
    tC = tC + (t3 - t2)

cleanup()
```

Caveat: measuring time takes time!

`time.time()`: ~40 ns (and the actual time fluctuates)

Microbenchmarks

What do we do if `function_A()` takes much less time than `time.time()`?

```python
import time.time

initialize()
tA, tB, tC = 0

for i in range(1000000):
    t0 = time.time()
    function_A()
    t1 = time.time()
    function_B()
    t2 = time.time()
    function_C()
    t3 = time.time()

    tA = tA + (t1 - t0)
    tB = tB + (t2 - t1)
    tC = tC + (t3 - t2)

cleanup()
```

Microbenchmark for `function_A()`:

```python
import time.time

initialize()

t0 = time.time()

for i in range(50000000):
    function_A()

t1 = time.time()

cleanup()
```

Microbenchmarks limitations

- It may not make sense to call `function_A()` in isolation
  - Take `sin(x)` for example: which value of x do we choose?
  - Always the same?
    - Are we sure `sin(0)` takes as much time as `sin(0.1)`?
  - A random value for x?
    - What if generating pseudo-random values takes more time than `sin()`?
- What about caches?
  - Caches will be “hot” (already filled with relevant data)
  - Microbenchmarking presents an over-optimistic picture of memory access times

AUTOMATED INSTRUMENTATION: PROFILERS

gprof

Add "-pg" to gcc/clang parameters

```
gcc -O3 -o app app.c -pg
```

Run the application

```
./app
```

Generate report

```
gprof app
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 63.77      3.82     3.82        1     3.82     4.24  tree_dfs
 28.88      5.55     1.73        1     1.73     1.73  lut_build
  4.17      5.80     0.25  1523737     0.00     0.00  aux_h_merge
  2.84      5.97     0.17    10331     0.00     0.00  aux_d_sort_swapper
  0.33      5.99     0.02                             tree_prune
  0.00      5.99     0.00     6715     0.00     0.00  aux_h_sort
  0.00      5.99     0.00      706     0.00     0.00  tree_gc
  0.00      5.99     0.00        6     0.00     0.00  dict_append_file
  0.00      5.99     0.00        1     0.00     0.00  dict_filter_dupes
  0.00      5.99     0.00        1     0.00     0.00  lut_hash_word
  0.00      5.99     0.00        1     0.00     0.00  solver_connected
  0.00      5.99     0.00        1     0.00     5.97  tree_build
```

```
index % time    self  children    called     name
                0.00    5.97       1/1           main [2]
[1]     99.7    0.00    5.97       1         tree_build [1]
                3.82    0.42       1/1           tree_dfs [3]
                1.73    0.00       1/1           lut_build [4]
                0.00    0.00       1/10331       aux_d_sort_swapper [5]
                0.00    0.00       1/1           lut_hash_word [12]
                0.00    0.00       1/6           dict_append_file [10]
-----------------------------------------------
                                                 <spontaneous>
[2]     99.7    0.00    5.97                 main [2]
                0.00    5.97       1/1           tree_build [1]
                0.00    0.00       1/1           solver_connected [13]
-----------------------------------------------
                                4174             tree_dfs [3]
                3.82    0.42       1/1           tree_build [1]
[3]     70.8    3.82    0.42       1+4174    tree_dfs [3]
                0.17    0.25   10330/10331       aux_d_sort_swapper [5]
                0.00    0.00    6715/6715        aux_h_sort [8]
                0.00    0.00     706/706         tree_gc [9]
                                4174             tree_dfs [3]
```

- Pros
  - Easy to use
  - Exhaustive profile information
  - Generally low overhead
- Cons
  - Overhead increases when bottlenecks are in small, short functions (up to 2x runtime)
  - Limited accuracy

STOCHASTIC INSTRUMENTATION