Profiling

Topics

- How can we find what code takes the most time?
- How can we inspect a compiled executable?

Profiling:
time & gprof

Time

- Use time for how long a program takes to run:

```
(bbg)$ time ./myapp
real  0m7.546s
user  0m0.006s
sys   0m0.016s
```

Waiting

- Options to slow down a program:
  - Calling kernel sleep functions:
    ..
  - Busy waits, like:
    ```
    for (int i = 0; i < 20000000; i++) {
        // Do nothing
    }
    ```
- Busy wait is bad:
  - Consumes CPU time: not given to other threads
  - Consumes power: CPU runs at max speed
  - Time of delay..
  - Non-portable: changes with different CPU / compiler

Profiling with gprof

- Profiling:..
  - What parts take the most time?
- gprof Usage:
  - Enable with GCC flag: -pg
  - Log written to current directory when program exits
    - Log named gmon.out
  - Analyze log with one of:

```
(bbg)$ gprof myApp gmon.out
(host)$ arm-linux-gnueabihf-gprof myApp gmon.out
```

GCC bug:
Getting empty gprof?
compile with `-no-pie`

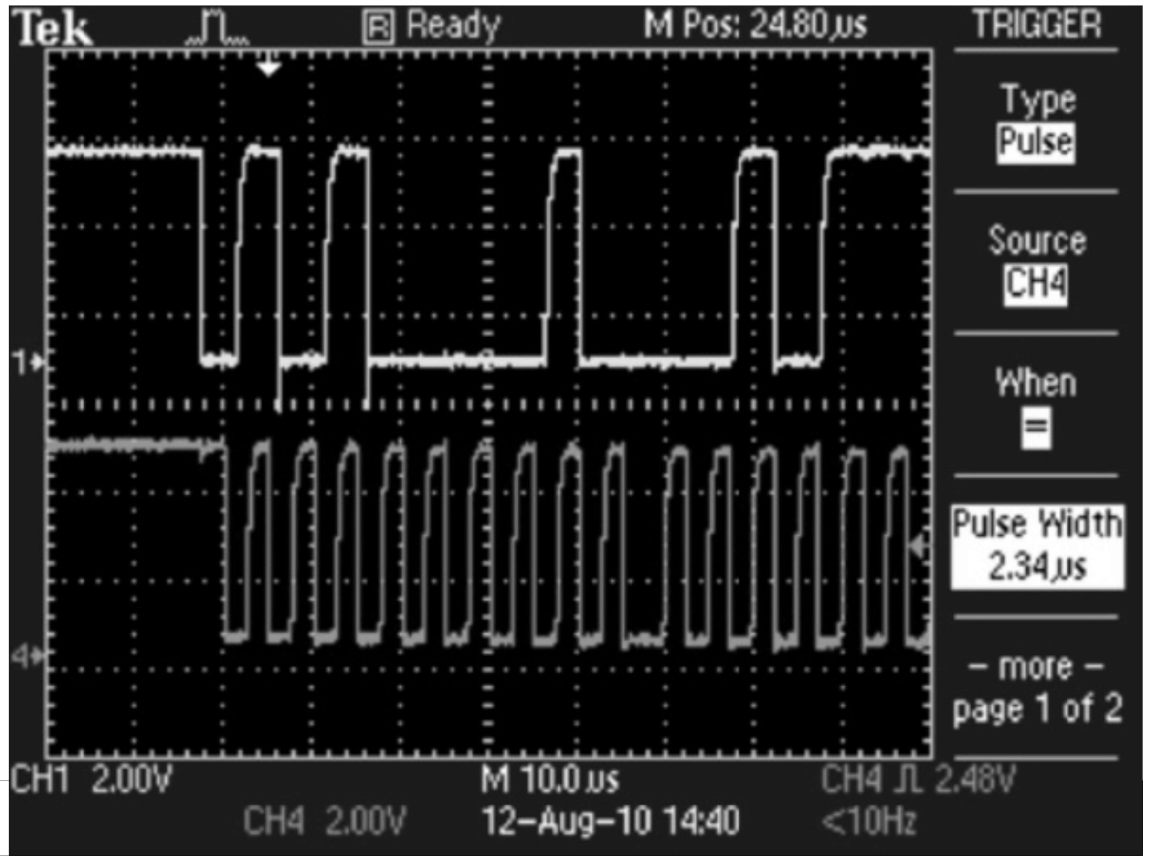
gprof example

```
(bbg)$ ./primer
... program runs and exits gracefully, writing gmon.out ...

(bbg)$ gprof primer gmon.out
<... omitted ...>
index % time    self  children    called     name
                                                 <spontaneous>
[1]     54.0   10.87    0.00                   __aeabi_uidiv [1]
-----------------------------------------------
                                                 <spontaneous>
[2]     14.8    2.98    0.00                   __udivdi3 [2]
-----------------------------------------------
                                                 <spontaneous>
[3]     10.3    0.00    2.08                   findPrimesThread [3]
                2.07    0.00   16588/16588         isPrime [4]
                0.01    0.00     754/754           storeNewPrime [9]
                0.00    0.00     754/4406          sleep_usec [25]
<...>
```

Profile with GPIO

- ..
  - Set bit (pin) when entering region of interest
  - Clear bit (pin) when leaving region.
- Use oscilloscope or logic analyzer to view actual pin changes.
- May be most useful within kernel or bare-metal due to sys-call overheads changing timing. Better with memory-mapped GPIO.



Information from Executables
LDD, readelf

LDD

- LDD:..
  - Helps find needed (missing?) libraries on system.
  - Linux libraries are .so files: shared object

```
(bbg)$ ldd ./primer
linux-vdso.so.1 (0xbea79000)
libpthread.so.0 => /lib/arm-linux-gnueabihf/libpthread.so.0 (0xb6f68000)
libm.so.6 => /lib/arm-linux-gnueabihf/libm.so.6 (0xb6ef3000)
libc.so.6 => /lib/arm-linux-gnueabihf/libc.so.6 (0xb6e03000)
/lib/ld-linux-armhf.so.3 (0x7f5be000)
```

  - Note the folder of the .so file:
    /lib/arm-linux-gnueabi/ Emulated floating point
    /lib/arm-linux-gnueabihf/ Hardware floating point

readelf

- Displays information on ELF executable files
  - ELF: Executable and Linkable Format

```
(bbg)$ readelf -h ./primer
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x10d89
  Start of program headers:          52 (bytes into file)
  Start of section headers:          42464 (bytes into file)
  Flags:                             0x5000400, Version5 EABI, hard-float ABI
...
```

Summary

- Profiling:
  - time to see how much time is used
  - gprof to see where time is used
- Info on Executables:
  - ldd to see what libraries are loaded
  - readelf to see executable's architecture etc.