# Announcements

- Program #5 on the web
- Reading
  - Notes

# Time

- On a computer there are many ways measure time
  - Wall time
    - always running
  - Process time
    - time your process was running
    - doesn't count:
      - time when your process was stopped for others
      - time when your program stopped to wait for I/O
      - time spent running in OS kernel (system calls)
- What time to use depends on what you are measuring

# Timing a Program

- To Time an entire program
  - time <program name> <program arguments>
  - prints out time in the form:
    - 2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w
    - first two numbers are user and system time
    - third number is wall time
    - fourth is percentage of wall time (user+system)/wall
    - remainder are paging and I/O statistics
- Provides Some idea about timing
  - hard to know what to do about it
    - what functions are taking most of the time?

# Representing Time Of Day

- How to deal with
  - timezones
  - daylight savings times
  - computers moving timezones
  - leap seconds?
- Answer:
  - keep time internally a seconds + fractions of a second
    - 2 32 bits values work nicely for 1ns accuracy for > 100 years
  - use a reference starting point
    - called epoch - midnight 1/1/1970
  - keep all time in UTC form until printed
    - no time zones or daylight savings time to deal with

# Adding Timing Calls to your program

- Wall Time
  - int gettimeofday(struct timeval *tv, struct timezone *tz);
  - tv is a struct of time tv_sec and tv_usec ($10^{-6}$ seconds)
  - tz is no longer used (pass null)
- Process Time
  - int getrusage(int who, struct rusage *usage);
  - who is RUSAGE_SELF or RUSAGE_CHILDREN
    - children is all terminated children
  - rusage contains fields for
    - struct timeval ru_utime; /* user time used */
    - struct timeval ru_stime; /* system time used */
    - various other OS stats

# Example Measuring Time

```
main()  {
    struct rusage startRU, endRU;
    struct timeval startWall, endWall;

    getimeofday(&startWall, NULL);
    getrusage(RUSAGE_SELF, &startRU);
    /* code to time */

    getimeofday(&endWall, NULL);
    getrusage(RUSAGE_SELF, &endRU);
    /* compute difference */
}
```

# Tools to help Measure Programs

- Gprof
  - compile program with special flags
    - gcc -pg foo.c
  - run program
    - produces timing output (gmon.out)
  - run "gprof a.out gmon.out"
    - produces output of time per function
    - includes number of times a function is called
    - uses statistical sampling rather than direct timer calls

# Sample Gprof Output

Each sample counts as 0.01 seconds.

| % time | cumulative seconds | self seconds | calls | self ns/call | total ns/call | name |
|---|---|---|---|---|---|---|
| 58.68 | 1.96 | 1.96 | 4999999 | 392.00 | 412.00 | add |
| 25.60 | 2.81 | 0.85 | 4999999 | 171.00 | 171.00 | lookup |
| 5.69 | 3.00 | 0.19 | 5000000 | 38.00 | 621.00 | insert |
| 3.29 | 3.12 | 0.11 | | | | main |
| 2.84 | 3.21 | 0.10 | | | | printCount |
| 2.40 | 3.29 | 0.08 | 4999977 | 16.00 | 16.00 | singleRotateLeft |
| 0.90 | 3.32 | 0.03 | | | | delete |
| 0.60 | 3.34 | 0.02 | 4999977 | 4.00 | 20.00 | balance |

# How Processors Spend their time

- Not all instructions take the same amount of time
- Some instructions are more expensive
- Caches
  - processors have caches to store values
    - each cache item stores several values (called line size)
  - The same instruction make take different amounts of time
    - misses from the cache can be 10-100x slower
  - Goal is to re-use same cache items multiple times
  - Consider:

```
for (i=0; i < limit; i++) {
    for (j=0; j < limit j++) {
        for (k=0; k < limit; k++)
            a[i,j] += b[i,k] * c[k,j];
    }
}
```

# Issues in Conducting Measurements

- Number of Runs
  - A single run is not sufficient
    - many things go on in a computer
      - operating system functions
      - other programs running
  - Multiple runs provide increased accuracy
    - take mean of K fastest runs
- Workload
  - what data is the program given for measurement runs?
    - Does it look like a "typical" use of the program
  - Many algorithms might look good if the measured workload
    - is too small - $O(n^2)$ algorithms similar to $O(n)$
    - is too large - not typical of usage pattern

# Sources of Performance Problems

- Too Small of I/O Ops
  - calling read one or a few characters at a time
- Left debugging printf's in program
- Poor Basic Algorithms
  - used $O(n^2)$ for large n and frequent operations
- Algorithms compose poorly
  - AVL tree - implementing deleteTable via deleteItem
    - forces frequent rebalancing
- Cache memory performance
  - Use each cache item once