# Measuring Time

**Alan L. Cox**
**alc@cs.rice.edu**

Some slides adapted from CMU 15.213 slides

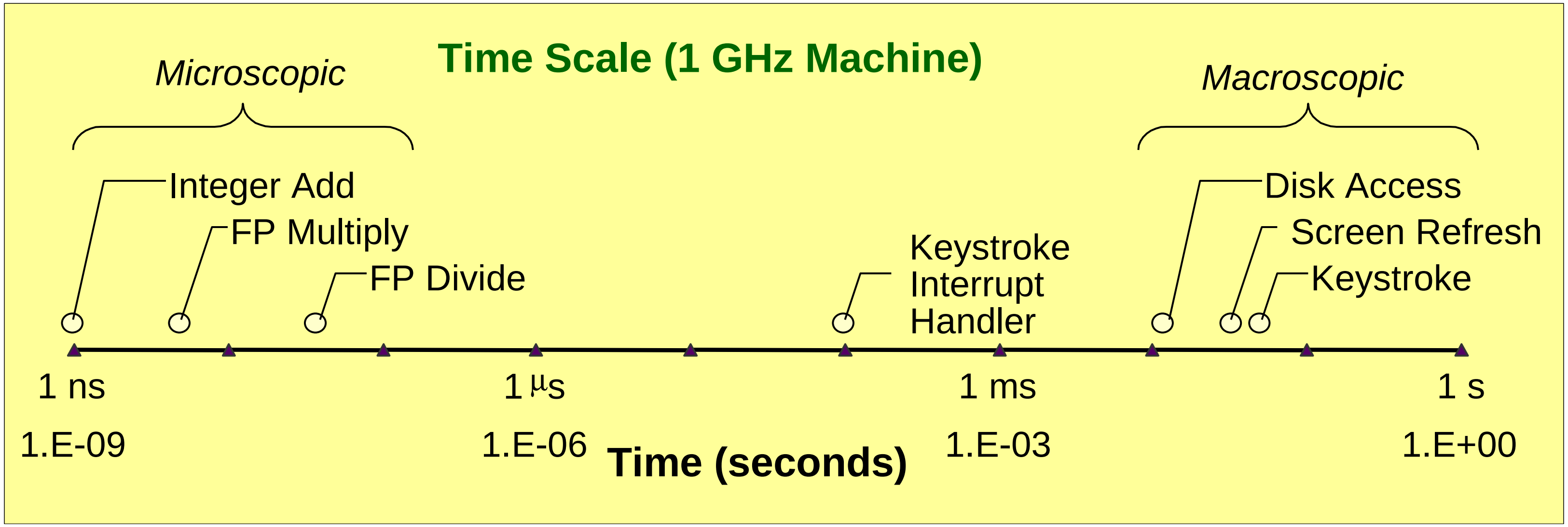
# Computer Time Scales

## Two Fundamental Time Scales

- **Processor: ~$10^{-9}$ s**
- **External events: ~$10^{-2}$ s**
  - **Keyboard input**
  - **Disk seek**
  - **Screen refresh**

## Implication

- **Can execute many instructions while waiting for external event to occur**
- **Can alternate among processes without anyone noticing**



# Measurement Challenge

## How Much Time Does Program *X* Require?

- **CPU time**
  - How many total seconds are used when executing *X*?
  - Measure used for most applications
  - Small dependence on other system activities
- **Actual ("Wall Clock") Time**
  - How many seconds elapse between the start and the completion of *X*?
  - Depends on system load, I/O times, etc.

## Confounding Factors

- **How does time get measured?**
- **Many processes share computing resources**
  - Transient effects when switching from one process to another
  - Suddenly, the effects of alternating among processes become noticeable

# "Time" on a Computer System

**real (wall clock) time**

 **= user time** *(time executing instructions in the user process)*

 **= system time** *(time executing instructions in kernel on behalf of user process)*

 **= some other user's time** *(time executing instructions for a different user process)*

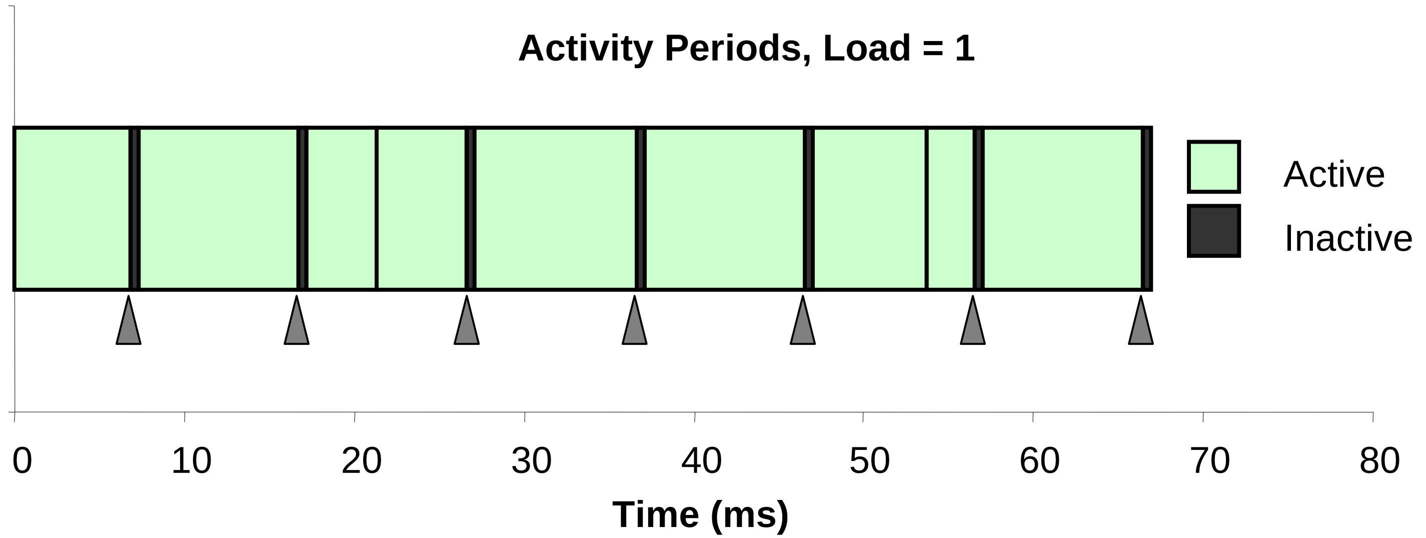

**= real (wall clock) time**

***We will use the word "time" to refer to user time***

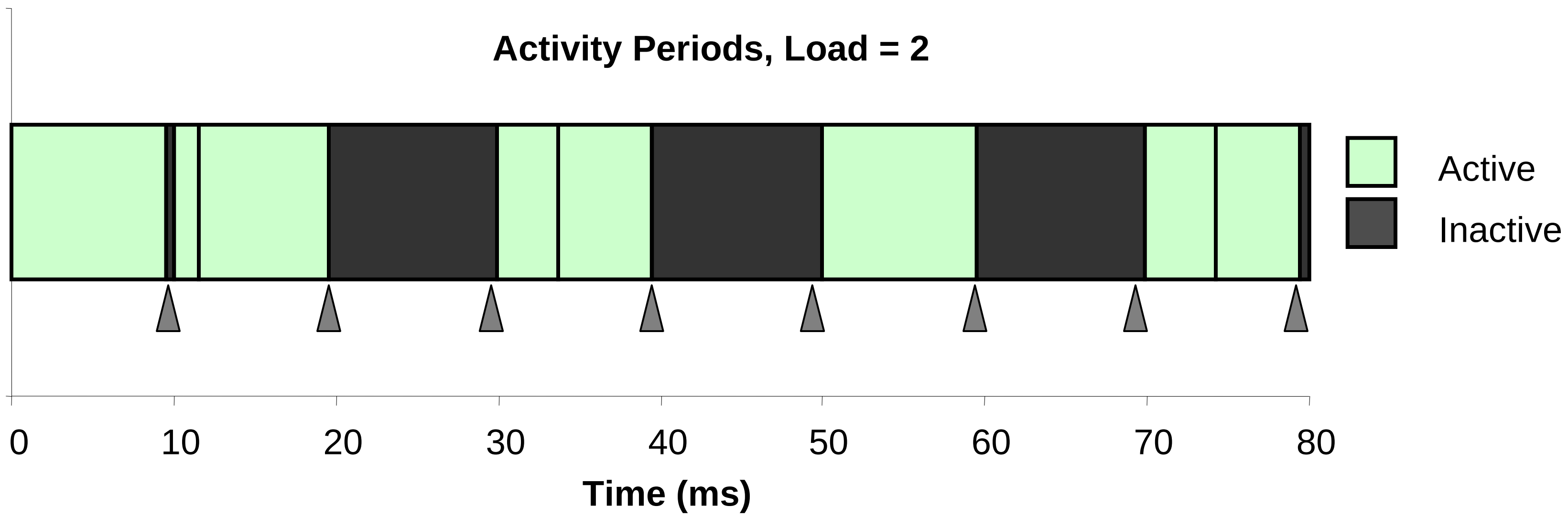
 **cumulative user time**

# Activity Periods: Light Load

- **Most of the time spent executing one process**
- **Periodic interrupts every 10ms**
  - **Interval timer**
  - **Keep system from executing one process to exclusion of others**
- **Other interrupts**
  - **Due to I/O activity**
- **Inactivity periods**
  - **System time spent processing interrupts**



# Activity Periods: Heavy Load



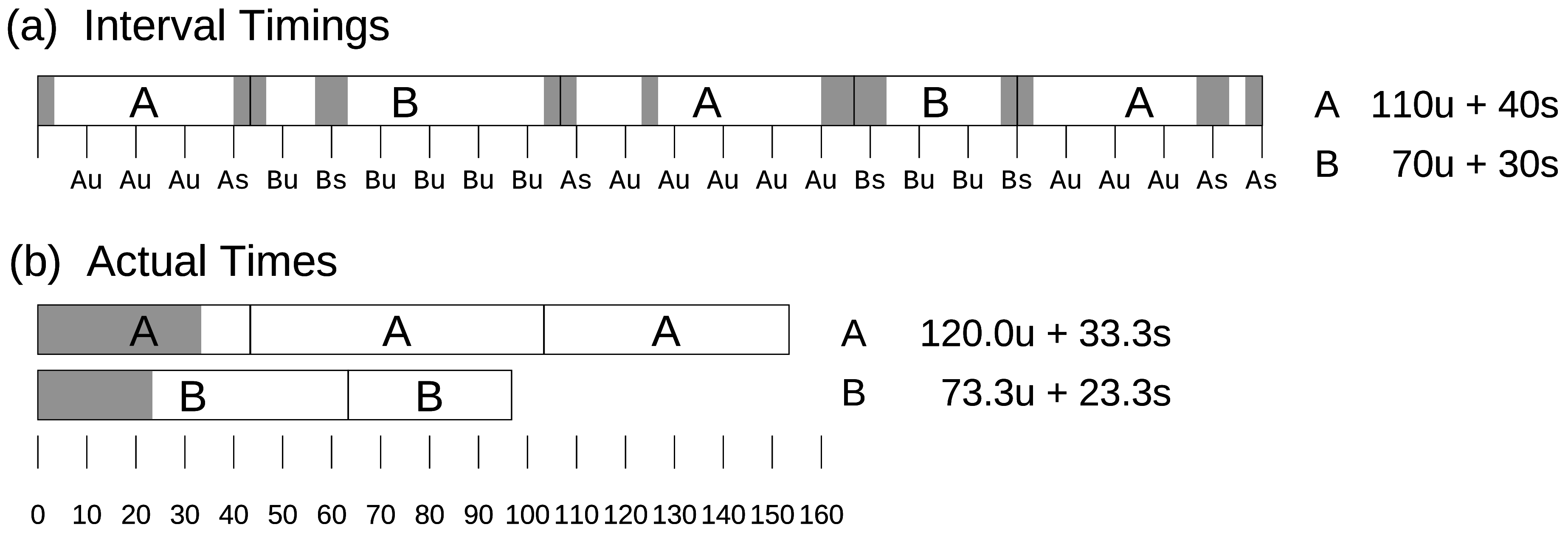

- **Sharing processor with one other active process**
- **From perspective of this process, system appears to be “inactive” for ~50% of the time**
  - **Other process is executing**

# Interval Counting

## OS Measures Runtimes Using Interval Timer

- **Maintain 2 counts per process**
  - User time
  - System time
- **Each timer interrupt, increment counter for executing process**
  - User time if running in user mode
  - System time if running in kernel mode

# Interval Counting Example



# Unix `time` Command

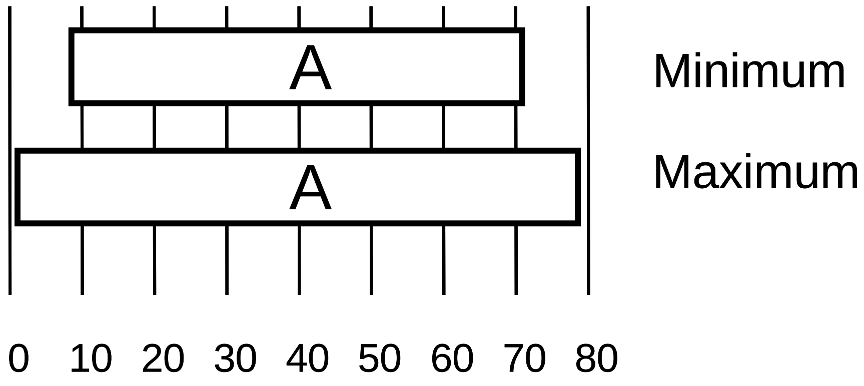
```
unix% time make osevent
gcc -O2 -Wall –Wextra -g -c clock.c
gcc -O2 -Wall –Wextra -g -c options.c
gcc -O2 -Wall –Wextra -g -c load.c
gcc -O2 -Wall –Wextra -g -o osevent . . .
0.820u 0.300s 0:01.32 84.8%
```

- **0.82 seconds user time**
  - **82 timer intervals**
- **0.30 seconds system time**
  - **30 timer intervals**
- **1.32 seconds wall clock time**
- **84.8% of total was used running these processes**
  - **(.82+0.3)/1.32 = .848**

# Accuracy of Interval Counting

## Worst Case Analysis

- **Timer Interval = δ**
- **Single process segment measurement can be off by ± δ**
- **No bound on error for multiple segments**
  - **Could consistently underestimate, or consistently overestimate**

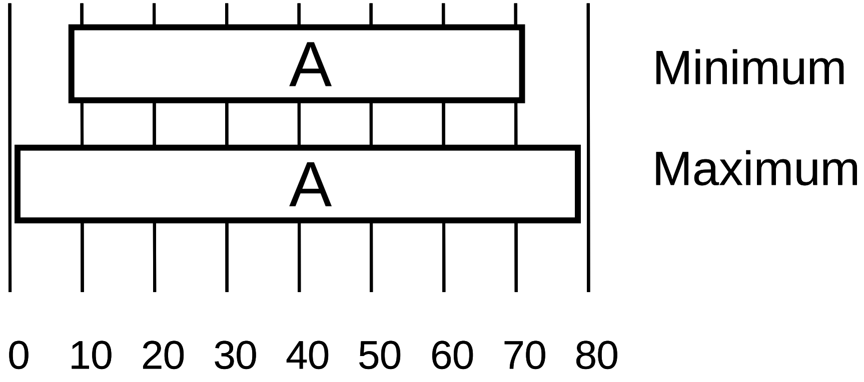



- **Computed time = 70ms**
- **Min Actual = 60 + ε**
- **Max Actual = 80 – ε**

# Accuracy of Interval Counting

## Average Case Analysis

- Over/underestimates tend to balance out
- As long as total run time is sufficiently large
  - Min run time ~1 second
  - 100 timer intervals
- Consistently miss ~4% overhead due to timer interrupts




- Computed time = 70ms
- Min Actual = 60 + ε
- Max Actual = 80 – ε

# Cycle Counters

## Most modern systems have built in registers that are incremented every clock cycle

- Very fine grained
- Often counts elapsed global time

## On x86 and x86-64 machines:

- 64 bit counter
  - Cycle counter period:
    - A 3 GHz machine wraps around every 195 years
- Special instruction to access

# x86 Cycle Counter

## RDTSC

- Assembly instruction to access 64-bit cycle counter
- Places low/high 32 bits in two different registers
- Expressed as machine cycles, not nanoseconds

```
uint64_t
rdtsc(void)
{
  uint32_t low, high;

  /* Get cycle counter */
  asm("rdtsc“ : "=a" (low), "=d" (high)); /* %eax, %edx */
  return (low | ((uint64_t)high << 32));
}
```

# Measuring Cycles with `rdtsc()`

## Idea

- **Get current cycle counter**
- **Compute something**
- **Get new cycle counter**
- **Perform 64-bit subtraction to get elapsed cycles**

```
uint64_t start, end;
int i;
int iters = 100;

start = rdtsc();
for (i = 0; i < iters; i++)
  getpid();
end = rdtsc();

printf("getpid(): Average cycles = %ld\n", (end – start) / iters);
```

# Converting Cycles to Seconds

## Idea

- **Compute elapsed cycles**
- **Get processor's clock frequency (cycles/second)**
- **Divide elapsed cycles by the clock frequency**

## How do you get the clock frequency?

```
UNIX% cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Xeon(R) CPU           X5460  @ 3.16GHz
stepping        : 6
cpu MHz         : 3158.758
cache size      : 6144 KB
…
```

# Measurement Pitfalls

## Overhead/resolution

- Calling `rdtsc()` incurs some overhead
- Resolution of `rdtsc()` may not allow very short code sequences to be timed
- Want to measure long enough code sequence to compensate

## Unexpected Cache Effects

- artificial hits or misses
- e.g., these measurements were taken with the Alpha cycle counter:
- `foo1(array1, array2, array3); /* 68,829 cycles */`
- `foo2(array1, array2, array3); /* 23,337 cycles */`

  vs.
- `foo2(array1, array2, array3); /* 70,513 cycles */`
- `foo1(array1, array2, array3); /* 23,203 cycles */`

# Dealing with Overhead & Cache Effects

- **Always execute function once to “warm up” cache**
- **Keep doubling number of times execute P() until reach some threshold (i.e., CMIN = 50000)**

```
int cnt = 1;
int i;
uint64_t start, end, tm;

do  {
  P();                              /* Warm up cache */
  start = rdtsc();
  for (i = 0; i < cnt; i++)
    P();
  end = rdtsc();
  tm = (end – start) / cnt;
  cnt += cnt;
} while ((end – start) < CMIN);  /* Make sure long enough */

return (tm);
```

# Multitasking Effects

## Cycle Counter Measures Elapsed Time

- **Keeps accumulating during periods of inactivity**
  - System activity
  - Running other processes

## Key Observation

- **Cycle counter never underestimates program run time**
- **Possibly overestimates by large amount**

# High Resolution CPU Time

```
#include <time.h>

struct timespec {
    time_t tv_sec;  /* seconds */
    long   tv_nsec; /* nanoseconds */
};

int clock_gettime(clockid_t id, struct timespec *tp);

struct timespec ts;

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts);
```

- **High resolution per-process timer based on the cycle counter**
  - **However, higher overhead than `rdtsc()`**
- `CLOCK_PROCESS_CPUTIME_ID` **is not portable (Linux only)**

# Time of Day Clock

- **Unix `gettimeofday()` function**
  - **Elapsed time since reference time (1/1/1970)**
- **Implementation**
  - **Uses interval counting on some machines**
    - **Coarse grained**
  - **Uses cycle counter on others**
    - **Fine grained, but significant overhead and only 1 microsecond resolution**

```
#include <sys/time.h>
#include <unistd.h>

  struct timeval tstart, tfinish;
  double tsecs;
  gettimeofday(&tstart, NULL);
  P();
  gettimeofday(&tfinish, NULL);
  tsecs = (tfinish.tv_sec - tstart.tv_sec) +
      1e-6 * (tfinish.tv_usec - tstart.tv_usec);
```

# Measurement Summary

## Timing is highly case and system dependent

- **What is overall duration being measured?**
  - **> 1 second: interval counting is OK**
  - **<< 1 second: must use cycle counters**
- **On what hardware / OS / OS version?**
  - **Accessing counters (`clock_gettime`? `rdtsc`?)**
  - **Timer interrupt overhead**
  - **Scheduling policy**

## Devising a Measurement Method

- **Long durations: use Unix `time` command**
- **Short durations**
  - **Use `clock_gettime` or `gettimeofday`**
  - **Work directly with cycle counters**

# Important Tools When Optimizing

## Observation

- **Generating assembly code**
  - Lets you see what optimizations compiler can make
  - Understand capabilities/limitations of particular compiler

## Measurement

- **Accurately compute time taken by code**
  - Most modern machines have built in cycle counters
  - Using them to get reliable measurements is tricky
- **Profile procedure calling frequencies**
  - Unix: `gprof`

# Profiling Example

## Task

1. **Count word frequencies in text document**
2. **Produce sorted list of words in descending frequency**

## Steps

1. **Convert strings to lowercase**
2. **Apply hash function**
3. **Read words and insert into hash table**
   - **Mostly list operations**
   - **Maintain counter for each unique word**
4. **Sort results**

## Data Set

- **Collected works of Shakespeare**
- **946,596 total words, 26,596 unique**
- **Initial implementation: 9.2 seconds**

| Shakespeare's most frequent words | |
|---|---|
| 29,801 | the |
| 27,529 | and |
| 21,029 | I |
| 20,957 | to |
| 18,514 | of |
| 15,370 | a |
| 14,010 | you |
| 12,936 | my |
| 11,722 | in |
| 11,519 | that |

# Profiling Example

## Augment Executable Program with Timing Functions

## Computes (approximate) amount of time spent in each function

- **Periodically (~ every 10ms) interrupt program**
- **Determine what function is currently executing**
- **Increment its timer by interval (e.g., 10ms)**

## Counts how many times each function is called

```
gcc –O2 –pg prog.c –o prog
prog
gprof prog
```

**Executes normally, plus generates file `gmon.out`**

**Generates profile info from `gmon.out`**

# Profiling Example: Results

| % time | cumulative seconds | self seconds | calls | self ms/call | total ms/call | name |
|---|---|---|---|---|---|---|
| 86.60 | 8.21 | 8.21 | 1 | 8210.00 | 8210.00 | sort_words |
| 5.80 | 8.76 | 0.55 | 946596 | 0.00 | 0.00 | lower1 |
| 4.75 | 9.21 | 0.45 | 946596 | 0.00 | 0.00 | find_ele_rec |
| 1.27 | 9.33 | 0.12 | 946596 | 0.00 | 0.00 | h_add |

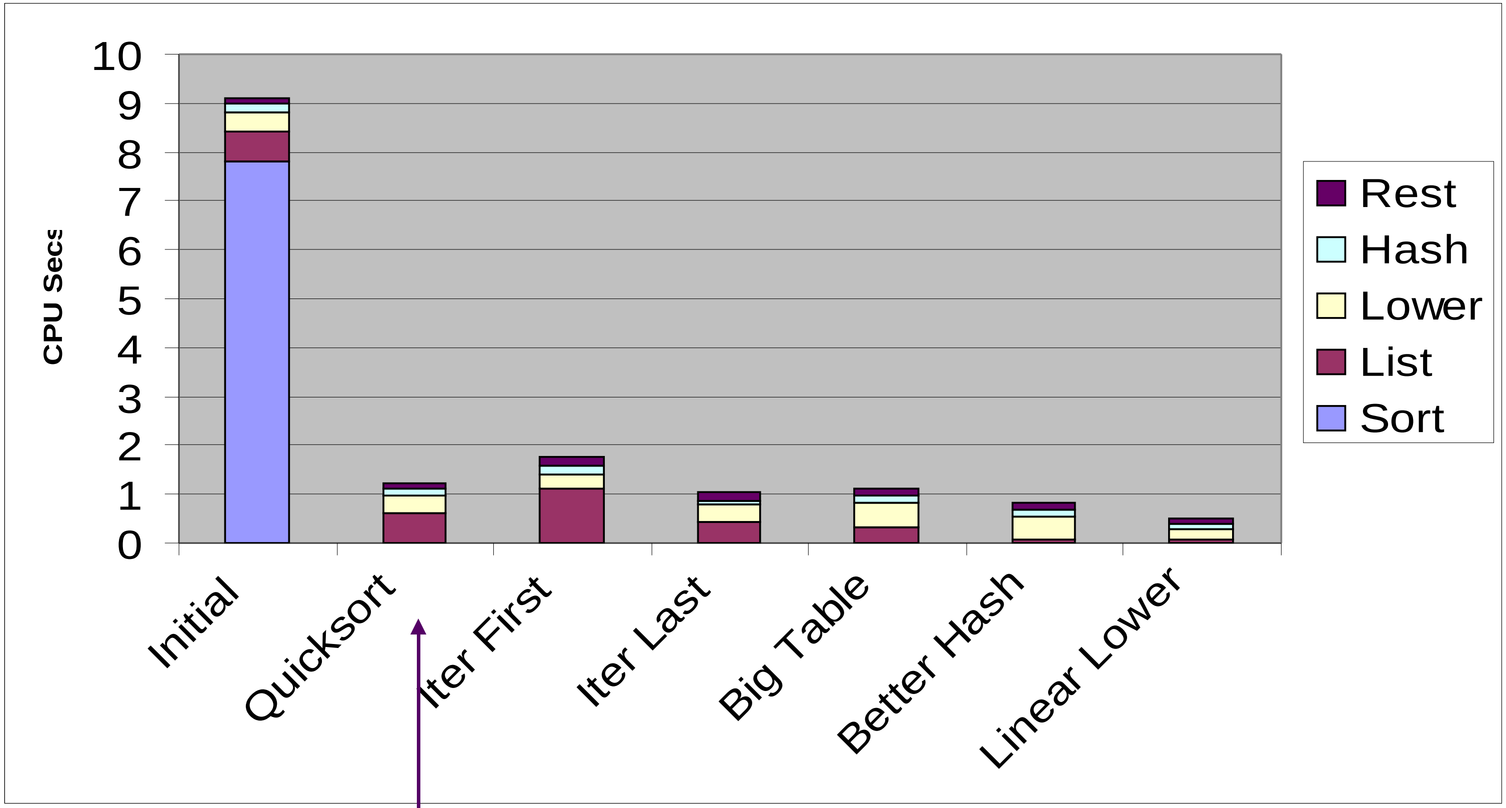
time and #calls per function

average time per function call (self = function, total = function + children)

Bottleneck:

Inefficient sort:  1 call = 87% of CPU time

# Profiling Example Optimized: 1




Use library **`qsort`** instead

# Profiling Example Optimized: 2


Use iterative function to insert elements into linked list first or last

Last $\rightarrow$ tends to place most common words at front of list

More hash buckets

Better hash function

Move **`strlen`** out of **`lower`** loop

# Profiling Observations

## Benefits

- Helps identify performance bottlenecks
- Especially useful with large, complex systems

## Limitations

- Only shows performance for data tested
  - E.g., mostly short test words → linear lower didn't gain big
    - Quadratic inefficiency could remain lurking in code

```
for (i=0; i<strlen(str); i++) {
  str[i] = tolower(str[i]);
}
```

- Timing mechanism fairly crude
  - Only accurate for programs that run long enough (> 3 sec)

# Better profilers

## Linux OProfile

- **No special compilation options**
  - **Profile preexisting, stock applications**
- **Uses cycle counters**
- **Simultaneously profile application and operating system**

# Next Time

## Virtual Memory