Testing, Timing, Profiling, & Instrumentation

CS 217

• How do you know if your program is correct?
  ◦ Will it ever core dump?
  ◦ Does it ever produce the wrong answer?
    – Testing

• How do you know what your program is doing?
  ◦ How fast is your program?
  ◦ Why is it slow for one input but not for another?
  ◦ Does it have a memory leak?
    – Timing
    – Profiling
    – Instrumentation

See Kernighan & Pike book:
“The Practice of Programming”
Program Verification

• How do you know if your program is correct?
  ○ Can you prove that it is correct?
  ○ Can you prove properties of the code?
    – e.g., it terminates

Program Checker

Right/Wrong

setarray.c

Program Testing

• Convince yourself that your program probably works

Test Program

Probably Right/Wrong

setarray.c

Specification

How do you write a test program?
Test Programs

- Properties of a good test program
  - Tests boundary conditions
  - Exercise as much code as possible
  - Produce output that is known to be right/wrong

  How do you achieve all three properties?

Program Testing

- Testing boundary conditions
  - Almost all bugs occur at boundary conditions
  - If program works for boundary cases, it probably works for others

- Exercising as much code as possible
  - For simple programs, can enumerate all paths through code
  - Otherwise, sample paths through code with random input
  - Measure test coverage

- Checking whether output is right/wrong?
  - Match output expected by test programmer (for simple cases)
  - Match output of another implementation
  - Verify conservation properties

  - Note: real programs often have fuzzy specifications
Example Test Program

```c
int main(int argc, char *argv[]) {
    Set_T oSet;
    SetIter_T oSetIter;
    const char *pcKey;
    char *pcValue;
    int iLength;

    /* Test Set_new, Set_put, Set_getKey, Set_getValue. */
    oSet = Set_new(2, myStringCompare);
    Set_put(oSet, "Ruth", "RightField");
    Set_put(oSet, "Gehrig", "FirstBase");
    Set_put(oSet, "Mantle", "CenterField");
    Set_put(oSet, "Jeter", "Shortstop");
    printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
            ");
    printf("This output should list 4 players and their positions\n");
    printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
            ");
    pcKey = (const char *)Set_getKey(oSet, "Ruth");
    pcValue = (char *)Set_getValue(oSet, "Ruth");
    printf("%s\t%s\n", pcKey, pcValue);
    pcKey = (const char *)Set_getKey(oSet, "Gehrig");
    pcValue = (char *)Set_getValue(oSet, "Gehrig");
    printf("%s\t%s\n", pcKey, pcValue);
    pcKey = (const char *)Set_getKey(oSet, "Mantle");
    pcValue = (char *)Set_getValue(oSet, "Mantle");
    printf("%s\t%s\n", pcKey, pcValue);
    pcKey = (const char *)Set_getKey(oSet, "Jeter");
    pcValue = (char *)Set_getValue(oSet, "Jeter");
    printf("%s\t%s\n", pcKey, pcValue);
}
```

Systematic Testing

- Incremental testing
  - Test as write code
  - Test simple cases first
  - Test code bottom-up

- Stress testing
  - Generate test inputs procedurally
  - Intentionally create error situations for testing
  - Run tests as batch processes ... often

```c
void *testmalloc(size_t n) {
    static int count = 0;
    if (++count > 10) return 0;
    else return malloc(n);
}
```
Timing, Profiling, & Instrumentation

- How do you know what your code is doing?
  - How slow is it?
  - How long does it take for certain types of inputs?
  - Where is it slow?
    - Which code is being executed most?
  - Why am I running out of memory?
    - Where is the memory going?
    - Are there leaks?
  - Why is it slow?
    - How imbalanced is my binary tree?

```
Input → Program → Output
```

Timing

- Most shells provide tool to time program execution
  - e.g., bash "time" command

```
bash> tail -1000 /usr/lib/dict/words > input.txt
bash> time sort5.pixie < input.txt > output.txt
real   0m12.977s
user   0m12.860s
sys    0m0.010s
```
Timing

• Most operating systems provide a way to get the time
  ◦ e.g., UNIX “gettimeofday” command

```c
#include <sys/time.h>
struct timeval start_time, end_time;
gettimeofday(&start_time, NULL);
<execute some code here>
gettimeofday(&end_time, NULL);

float seconds = end_time.tv_sec - start_time.tv_sec +
  1.0E-6F * (end_time.tv_usec - start_time.tv_usec);
```

Profiling

• Gather statistics about your program’s execution
  ◦ e.g., how much time did execution of a function take?
  ◦ e.g., how many times was a particular function called?
  ◦ e.g., how many times was a particular line of code executed?
  ◦ e.g., which lines of code used the most time?

• Most compilers come with profilers
  ◦ e.g., pixie and prof
```
#include <stdio.h>
#include <string.h>
#include "stringarray.h"

int CompareStrings(void *s1, void *s2)
{
    return strcmp(s1, s2);
}

int main()
{
    StringArray_T stringarray = StringArray_new();
    StringArray_read(stringarray, stdin);
    StringArray_sort(stringarray, CompareStrings);
    StringArray_write(stringarray, stdout);
    StringArray_free(stringarray);
    return 0;
}
```

### Summary of ideal time data (pixie-counts)

<table>
<thead>
<tr>
<th>Function</th>
<th>excl. secs</th>
<th>excl.%</th>
<th>cum.%</th>
<th>cycles</th>
<th>instructions</th>
<th>calls</th>
<th>function</th>
<th>(dsso: file, line)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array_sort</td>
<td>1.935</td>
<td>54.3%</td>
<td>54.3%</td>
<td>1742355689</td>
<td>1778628217</td>
<td>1</td>
<td>Array_sort</td>
<td>(sort5: array.c, 110)</td>
</tr>
<tr>
<td>strcmp</td>
<td>5.879</td>
<td>36.3%</td>
<td>91.2%</td>
<td>1149885000</td>
<td>1239970000</td>
<td>499950000</td>
<td>strcmp (lib.so.1: strcmp.a, 34)</td>
<td></td>
</tr>
<tr>
<td>fstrcmp</td>
<td>1.936</td>
<td>54.3%</td>
<td>105.5%</td>
<td>270229536</td>
<td>575736340</td>
<td>499950000</td>
<td>fstrcmp (lib.so.1: fstrcmp.c, 28)</td>
<td></td>
</tr>
<tr>
<td>_doprint</td>
<td>0.010</td>
<td>0.1%</td>
<td>99.8%</td>
<td>1879873</td>
<td>2279949</td>
<td>10000</td>
<td>_doprint (lib.so.1: doprint.c, 227)</td>
<td></td>
</tr>
<tr>
<td>strlen</td>
<td>0.004</td>
<td>0.0%</td>
<td>99.8%</td>
<td>746528</td>
<td>584896</td>
<td>20000</td>
<td>strlen (lib.so.1: strlen.c, 58)</td>
<td></td>
</tr>
<tr>
<td>fgets</td>
<td>0.004</td>
<td>0.0%</td>
<td>99.8%</td>
<td>700059</td>
<td>880218</td>
<td>10001</td>
<td>fgets (lib.so.1: fgets.c, 28)</td>
<td></td>
</tr>
<tr>
<td>_memcpy</td>
<td>0.003</td>
<td>0.0%</td>
<td>99.9%</td>
<td>494950</td>
<td>666600</td>
<td>10018</td>
<td>_memcpy (lib.so.1: memcpy.c, 29)</td>
<td></td>
</tr>
<tr>
<td>_getopt</td>
<td>0.002</td>
<td>0.0%</td>
<td>99.9%</td>
<td>420000</td>
<td>510000</td>
<td>10000</td>
<td>_getopt (sort5: array.c, 72)</td>
<td></td>
</tr>
<tr>
<td>argv_add</td>
<td>0.002</td>
<td>0.0%</td>
<td>99.9%</td>
<td>417401</td>
<td>411003</td>
<td>10000</td>
<td>argv_add (sort5: array.c, 103)</td>
<td></td>
</tr>
<tr>
<td>printf</td>
<td>0.002</td>
<td>0.0%</td>
<td>99.9%</td>
<td>340000</td>
<td>450000</td>
<td>10000</td>
<td>printf (lib.so.1: printf.c, 23)</td>
<td></td>
</tr>
<tr>
<td>array_add</td>
<td>0.002</td>
<td>0.0%</td>
<td>99.9%</td>
<td>310028</td>
<td>250028</td>
<td>1</td>
<td>array_add (sort5: array.c, 22)</td>
<td></td>
</tr>
<tr>
<td>StringArray_write</td>
<td>0.001</td>
<td>0.0%</td>
<td>99.9%</td>
<td>267789</td>
<td>296579</td>
<td>2680</td>
<td>StringArray_write (sort5: str...c, 22)</td>
<td></td>
</tr>
<tr>
<td>resolve_relocations</td>
<td>0.001</td>
<td>0.0%</td>
<td>99.9%</td>
<td>262426</td>
<td>345576</td>
<td>10164</td>
<td>resolve_relocations (rld: rld.c, 2636)</td>
<td></td>
</tr>
<tr>
<td>malloc</td>
<td>0.001</td>
<td>0.0%</td>
<td>99.9%</td>
<td>263196</td>
<td>329639</td>
<td>10038</td>
<td>malloc (lib.so.1: malloc.c, 933)</td>
<td></td>
</tr>
<tr>
<td>_free</td>
<td>0.001</td>
<td>0.0%</td>
<td>99.9%</td>
<td>262829</td>
<td>413379</td>
<td>10000</td>
<td>_free (lib.so.1: malloc.c, 127)</td>
<td></td>
</tr>
</tbody>
</table>

bash> cc -o sort5.o etc.
bash> pixie sort5
bash> sort5.pixie < input.txt > output.txt
bash> prof sort5.Counts
```
Instrumentation

- Gather statistics about your data structures
  - e.g., how many nodes are at each level of my binary tree?
  - e.g., how many elements are in each bucket of my hash table?
  - e.g., how much memory is allocated from the heap?

```
2, 1, 4, 3, 6, 5, 8, 7, 10, 9, 11
```

Instrumentation Example

Hash table implemented as array of sets

```
typedef struct Hash *Hash_T;

struct Hash {
    Set_T *buckets;
    int nbuckets;
};

void Hash_PrintBucketCounts(Hash_T oHash, FILE *fp) {
    int i;
    /* Print number of elements in each bucket */
    for (i = 0; i < oHash->nbuckets; i++)
        fprintf(fp, "%d
", Set_getLength(oHash->buckets[i]), fp);
    fprintf(fp, "\n");
}
```
Summary & Guidelines

• Test your code as you write it
  ○ It is very hard to debug a lot of code all at once
  ○ Isolate modules and test them independently
  ○ Design your tests to cover boundary conditions
  ○ Test modules bottom-up

• Instrument your code as you write it
  ○ Include asserts and verify data structure sanity often
  ○ Include debugging statements (e.g., #ifdef DEBUG and #endif)
  ○ You’ll be surprised what your program is really doing!!!

• Time and profile your code **only** when you are done
  ○ Don’t optimize code unless you have to (you almost never will)
  ○ Fixing your algorithm is almost always the solution
  ○ Otherwise, running optimizing compiler is usually enough