Testing, Profiling, and Instrumentation

Testing, Profiling, and Instrumentation

• 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

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 Testing

• Convince yourself that your program probably works

Specification ➔ Test Program ➔ Probably Right/Wrong

Specification ➔ Program Checker ➔ Right/Wrong

symtable.c ➔ ?

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?

Test Boundary Conditions

- Most bugs occur at boundary conditions
  - What is the most common boundary condition bug?

- What are the boundary conditions of this code?

```c
int i;
char s[MAXLINE];

for ( i=0; (s[i] = getchar()) != '\n' && i < MAXLINE-1; i++ )
    s[i] = '\0';
```

- Boundary conditions
  - Input starts with \n
Test Boundary Condition, cont’d

- Rewrite the code

```c
for ( i=0; i<MAXLINE-1; i++ )
    if ((s[i] = getchar()) == '\n')
        break;
S[i] = '\0';
```

- Another boundary condition: EOF

```c
for ( i=0; i<MAXLINE-1; i++ )
    if ((s[i] = getchar()) == '\n' || s[i] == EOF)
        break;
S[i] = '\0';
```

- What are other boundary conditions?
  - Nearly full
  - Exactly full
  - Over full

Test As Your Write Code

- Recall using “assert” in previous lecture

- Check pre- and post-conditions for each function
  - Boundary conditions

- Check invariants

- Check error returns

- What is the typical percentage of code doing error-checking?
**Systematic Testing**

- Test plan
  - Unit tests (for each module)
  - System tests

- Design test cases
  - Know what input gives what output, according to the spec
  - Check properties of output
  - Compare independent implementations
  - What legal inputs to test
    - Boundary conditions and “inductions”
    - Multi-dimensional inputs and combinations
      - Assembler: instructions, comments, directives
      - Numerical program: operations, legal combinations
  - What illegal inputs to test
    - Common illegal inputs
    - Possible security holes
    - Multi-dimensional illegal inputs

**A Test Case Example**

- “De-comment” test
  - Empty comments
  - Test single line comment
  - Test very long line
  - Multiple line comment
  - Test many lines
  - Nested comment
  - String literal in comment
  - Character literal in comment
  - Comment in string literal
  - Comment in character literal
  - Unterminated comment
  - . . .

**Test Automation**

- Automation can provide better test coverage

- Test program
  - Client code to test modules
  - Scripts to test inputs and compare outputs

- QA test is an iterative process
  - Initial automated test program or scripts
  - Test simple parts first
  - Unit tests before system tests
  - Add tests as new cases created

- Regression test
  - Test all cases to compare the new version with the previous one
  - A bug fix often create new bugs in a large software system

- What tests cannot be done automatically?

**Stress Tests**

- Motivations
  - Use computer to generate inputs to test
  - High-volume tests often find bugs

- What to generate
  - Very long inputs
  - Random inputs (binary vs. ASCII)
  - Fault injection

- How much test
  - Exercise all data paths
  - Test all error conditions
Who Test What

- Implementers
  - White-box testing
  - Pros: An implementer knows all data paths
  - Cons: influenced by how code is designed/written

- Quality Assurance (QA) engineers
  - Black-box testing
  - Pros: No knowledge about the implementation
  - Cons: Unlikely to test all data paths

- Customers
  - Field test
  - Pros: Unexpected ways of using the software, “debug” specs
  - Cons: No enough cases

Timing, Profiling, and 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?

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

- Most operating systems provide a way to get the time
  - e.g., UNIX “grep” command

  #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);
Timing

• Some CPU provides access to CPU “ticks”

```c
unsigned long long int getCPUTicks(void)
{
    unsigned long long int x;
    asm volatile (".byte 0x0f, 0x31":"=A" (x));
    return x;
}
```

Profiling

• Gather statistics about your program’s execution
  ○ How much time did execution of a function take?
  ○ How many times was a particular function called?
  ○ How many times was a particular line of code executed?
  ○ Which lines of code used the most time?

• Most compilers come with profilers
  ○ e.g. pixie and prof

Profiling with gcc+gprof

• Apparently, prof doesn’t work with gcc, must use gprof

```bash
PROFFLAGS = -Wall -ansi -pedantic -O4 -DNDEBUG -pg
CFLAGS= $(PROFFLAGS)
profile: player testinput
    -player MIN <testinput
gprof player >profile
```

```bash
% make profile
gcc -Wall -O4 -DNDEBUG -pg -o player . .
player MIN <testinput
5
```

```bash
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

MIN player reports invalid move by MAX player

```bash
make: *** [profile] Error 1 (ignored)
gprof player >profile
```
First part of gprof profile looks like this; it's for sophisticated users (i.e., more sophisticated than your humble professor) and I will ignore it

Don't even think of optimizing these

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?
Instrumentation Example

```c
static void Tree_FillHistogram(Tree_T oTree, TreeNode_T cNode, 
    int *ipHistogram, int iLevel, int iMaxLevels) 
{
    int i;
    /* Increment histogram entry */
    ipHistogram[iLevel]++;
    /* Recurse to children */
    if (iLevel < iMaxLevels)
        for (i = 0; i < cNode->nchildren; i++)
            Tree_FillHistogram(oTree, cNode->child[i] iLevel+1, iMaxLevels);
}

void Tree_PrintHistogram(Tree_T oTree, FILE *fp) 
{
    /* Define histogram */
    int ipHistogram[MAX_LEVELS];
    int i;
    /* Load histogram recursively */
    Tree_FillHistogram(oTree, oTree->root, ipHistogram, 0, MAX_LEVELS);
    /* Print histogram */
    for (i = 0; i < MAX_LEVELS; i++)
        fprintf(fp, "d ", ipHistogram[i]);
    fprintf(fp, "n");
}
```

Interate...

1. Develop program
2. Test; modify program
3. Test again; if bugs, back to step 2
4. Is it fast enough? If not,
5. Profile; modify program; back to step 3
   • Typically, reprofile several times until no more performance improvement is justified

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

Hash table implemented as array of sets

```c
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");
}
```