Program Verification
Agenda

Famous bugs
Common bugs
Testing (from lecture 6)
Reasoning about programs
Techniques for program verification
Famous Bugs

The first bug: A moth in a relay (1945)
At the Smithsonian (currently not on display)
(in)Famous Bugs

- Safety-critical systems

Therac-25 medical radiation device (1985)
At least 5 deaths attributed to a race condition in software
(in)Famous Bugs

- Mission-critical systems

SW interface issue, backup failed
Cost: $400M payload

Race condition in power control software
Cost: $4B
(in)Famous Bugs

- Commodity hardware / software

Pentium bug (1994)
Float computation errors
Cost: $475M

Code Red worm on MS IIS server (2001)
Buffer overflow exploited by worm
Infected 359k servers
Cost: >$2B
Common Bugs

- **Runtime bugs**
  - Null pointer dereference (access via a pointer that is Null)
  - Array buffer overflow (out of bound index)
    - Can lead to security vulnerabilities
  - Uninitialized variable
  - Division by 0

- **Concurrency bugs**
  - Race condition (flaw in accessing a shared resource)
  - Deadlock (no process can make progress)

- **Functional correctness bugs**
  - Input-output relationships
  - Interface properties
  - Data structure invariants
  - ...
Program Verification

**Ideally**: Prove that any given program is correct

Specification → General Program Checker → Right or Wrong

In general: Undecidable

This lecture: For some (kinds of) properties, a Program Verifier can provide a proof (if right) or a counterexample (if wrong)
Pragmatically: Convince yourself that a specific program probably works

“Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.”
– Edsger Dijkstra
Path Testing Example (Lecture 6)

Example pseudocode:

```c
if (condition1)
    statement1;
else
    statement2;
...
if (condition2)
    statement3;
else
    statement4;
...
```

Path testing:

Should make sure all logical paths are executed

Four paths for four combinations of (condition1, condition 2): TT, TF, FT, FF

- Simple programs => maybe reasonable
- Complex program => combinatorial explosion!!!
  - Path test code fragments
Agenda

Famous bugs
Common bugs
Testing (from lecture 6)
Reasoning about programs
Techniques for program verification
Reasoning about Programs

Try out the program, say for x=3

- At line 4, before executing the loop: x=3, y=1, z=0
- Since z != x, we will execute the while loop
- At line 4, after 1\textsuperscript{st} iteration of loop: x=3, z=1, y=1
- At line 4, after 2\textsuperscript{nd} iteration of loop: x=3, z=2, y=2
- At line 4, after 3\textsuperscript{rd} iteration of loop: x=3, z=3, y=6
- Since z == x, exit loop, return 6: It works!

Example:

factorial program

Check:
If x >= 0, then y = fac(x)
(fac is the mathematical function)
Reasoning about Programs

Example:
factorial program

Check:
If $x \geq 0$, then $y = \text{fac}(x)$

Try out the program, say for $x=4$
- At line 4, before executing the loop: $x=4$, $y=1$, $z=0$
- Since $z \neq x$, we will execute the while loop
- At line 4, after 1\textsuperscript{st} iteration of loop: $x=4$, $z=1$, $y=1$
- At line 4, after 2\textsuperscript{nd} iteration of loop: $x=4$, $z=2$, $y=2$
- At line 4, after 3\textsuperscript{rd} iteration of loop: $x=4$, $z=3$, $y=6$
- At line 4, after 4\textsuperscript{th} iteration of loop: $x=4$, $z=4$, $y=24$
- Since $z == x$, exit loop, return 24: It works!
Reasoning about Programs

Try out the program, say for \( x=1000 \)
- At line 4, before executing the loop: \( x=1000, y=1, z=0 \)
- Since \( z \neq x \), we will execute the while loop
- At line 4, after 1\(^{st}\) iteration of loop: \( x=1000, z=1, y=1 \)
- At line 4, after 2\(^{nd}\) iteration of loop: \( x=1000, z=2, y=2 \)
- At line 4, after 3\(^{rd}\) iteration of loop: \( x=1000, z=3, y=6 \)
- At line 4, after 4\(^{th}\) iteration of loop: \( x=1000, z=4, y=24 \)

Example:
factorial program

Check:
If \( x \geq 0 \), then \( y = \text{fac}(x) \)

Example:

```c
int factorial(int x) {
    int y = 1;
    int z = 0;
    while (z != x) {
        z = z + 1;
        y = y * z;
    }
    return y;
}
```

Want to keep going on???
Let's try some mathematics …

- Annotate the program with “assertions” [Floyd 67]
  - Assertions (at program lines) are expressed as (logic) formulas
    - Here, we will use standard arithmetic
    - Meaning: Assertions hold before that line is executed

- For loops, we will use an assertion called a “loop invariant”
  - Invariant means that the assertion holds in each iteration of loop
  - We can prove this by using induction

Example:

```
1  int factorial(int x) {
2      int y = 1;
3      int z = 0;
4      while (z != x) {
5          z = z + 1;
6          y = y * z;
7      }
8      return y;
9  }
```

Check:
If \( x \geq 0 \), then \( y = \text{fac}(x) \)
Loop Invariant

Example:
factorial program

Check:
If \( x \geq 0 \), then \( y = \text{fac}(x) \)

• Loop invariant (assertion at line 4): \( y = \text{fac}(z) \)

• Try to prove by induction that the loop invariant holds
  • Base case: First time at line 4, \( z=0 \), \( y=1 \), \( \text{fac}(0)=1 \), \( y=\text{fac}(z) \) holds √
  • Induction hypothesis: Suppose \( y = \text{fac}(z) \) at line 4
  • Induction step: In next iteration of the loop (when \( z \neq x \))
    • \( z' = z+1 \) and \( y' = \text{fac}(z) \cdot z+1 = \text{fac}(z') \) (\( z'/y' \) denote updated values)
    • Therefore, at line 4, \( y' = \text{fac}(z') \), i.e., loop invariant holds again √
Proof of Correctness

Example:
factorial program

Check:
If x >= 0, then y = fac(x)

```
1 int factorial(int x) {
2     int y = 1;
3     int z = 0;
4     while (z != x) {
5         z = z + 1;
6         y = y * z;
7     }
8     return y;
9 }
```

• Loop invariant (assertion at line 4): \( y = \text{fac}(z) \) \(\checkmark\)

• What should we do now?
  • If loop condition is true, i.e., if \( z \neq x \), execute loop again, \( y = \text{fac}(z) \)
  • If the loop condition is false, i.e., if \( z = x \), exit the loop
    • At line 8, we now know that \( y = \text{fac}(z) \) AND \( z = x \), i.e., \( y = \text{fac}(x) \)
    • Thus, at return, \( y = \text{fac}(x) \)

• Proof of correctness of the factorial program is now done \(\checkmark\)
Program Verification

- Rich history in computer science
  - *Assigning Meaning to Programs* [Floyd, 1967]
    - Program is annotated with assertions (formulas in logic)
    - Program is proved correct by reasoning about assertions
  - *An Axiomatic Basis for Computer Programming* [Hoare, 1969]
    - Hoare Triple: \{P\} S \{Q\}
      - S: statement (or fragments) in program
      - P: precondition (formula in logic)
      - Q: postcondition (formula in logic)
    - Meaning: If S executes from a state where P is true, and if S terminates, then Q is true in the resulting state
    - This is called “partial correctness”
    - does not guarantee termination of S
    - For our example: \{x >= 0\} y = factorial(x) \{y = fac(x)\}
Program Verification

- **Proof Systems**
  - Perform reasoning using logic formulas and rules of inference
    - Soundness: If assertion A is proved, then A is true
    - Completeness: If assertion A is true, then A can be proved

- **Hoare Logic** [Hoare 69]
  - Inference rules for assignments, conditionals, loops, sequence
  - Given a program annotated with preconditions, postconditions, and loop invariants
    - Verification Condition (VC) can be generated automatically
    - If VC is “valid”, then program is correct
    - Validity of VC can be checked by a theorem-prover

- **Question**: Can these preconditions/postconditions/loop invariants be generated automatically?
Automatic Program Verification

• Question: Can these preconditions/postconditions/loop invariants be generated automatically?

• Answer: Yes! (in many cases)

• Techniques for deriving the assertions automatically
  • Model checking: based on exploring “states” of programs
  • Abstract interpretation: based on static analysis using “abstractions” of programs
  • … many other techniques

• Still an active area of research (after more than 45 years)!
Model Checking

• Temporal logic
  • Used for specifying correctness properties
  • [Pnueli, 1977]

• Model checking
  • Verifying temporal logic properties by state space exploration
  • [Clarke & Emerson, 1981] and [Queille & Sifakis, 1981]
Model Checker

- Model checker performs automatic state space exploration
  - If all reachable states are visited and error state is not reached, then property is proved correct
  - Otherwise, it provides a counterexample (trace to error state)

Model:

```c
int factorial(int x) {
    int y = 1;
    int z = 0;
    while (z != x) {
        z = z + 1;
        y = y * z;
    }
    return y;
}
```

Property: formula
- Is error state reachable?
  - (Example: error state is where $y \neq \text{fac}(x)$ at return)
Model Checking (simplified)

- Consider viewing a program *like* a DFA (not quite, …)
  - “State” in a program
    - Line number
    - Value of each program variable (not shown below)
  - “Transition” in a program
    - Statement in program (updates state)

- Example: factorial program

```c
int factorial(int x) {
    int y = 1;
    int z = 0;
    while (z != x) {
        z = z + 1;
        y = y * z;
    }
    return y;
}
```
Model Checking (simplified)

- Example: factorial program

```c
1 int factorial(int x) {
2    int y = 1;
3    int z = 0;
4    while (z != x) {
5        z = z + 1;
6        y = y * z;
7    }
8    return y;
9 }
```

- Number of program states
  - 9 (lines)*(states of x)*(states of y)*(states of z): 9 * 2^{32} * 2^{32} * 2^{32}
  - States are not represented explicitly, but symbolically as formulas
    - e.g. (z<y) represents all program states where z is less than y

- Many other enhancements are used ...
F-Soft Model Checker

Automatic tool for finding bugs in large C/C++ programs (NEC)

1: void pivot_sort(int A[], int n){
2:     int pivot=A[0], low=0, high=n;
3:     while ( low < high ) {
4:         do {
5:             low++ ;
6:         } while ( A[low] <= pivot ) ;
7:         do {
8:             high-- ;
9:         } while ( A[high] >= pivot );
10:        swap(&A[low],&A[high]);
11:     }
12: }

Array Buffer Overflow?

F-Soft

counterexample trace

Line 2: pivot=10, low=0, high=2
Line 3: low < high ? YES
Line 5: low = 1
Line 5: low = 2
Line 6: A[low] <= pivot ?

Buffer Overflow!!!
Summary

• Program verification
  • Provide *proofs of correctness* for classes of properties & programs
  • Testing *cannot* provide proofs of correctness (unless exhaustive)

• Proof systems based on logic
  • Users annotate the program with assertions
  • Theorem-provers perform search for proofs (with user guidance)

• Automatic verification techniques
  • Program assertions are derived automatically
  • But, scalability is an issue
    • Explosion in sets of reachable states
    • Worse for concurrent multi-threaded programs
      • Need to explore all possible interleavings of different threads

• COS 597B in Fall ’15: Automatic Reasoning about Software
Course Summary

We have covered:

Programming in the large
- The C programming language
- Testing
- Building
- Debugging
- Program & programming style
- Data structures
- Modularity
- Performance
Course Summary

We have covered (cont.):

Under the hood

- Number systems
- Language levels tour
  - Assembly language
  - Machine language
  - Assemblers and linkers
- Service levels tour
  - Exceptions and processes
  - Storage management
  - Dynamic memory management
  - Process management
  - I/O management
  - Signals
The Rest of the Course

Assignment 7
- Due on Dean’s Date (5/12) at 5PM
- Cannot submit late (University regulations)
- Cannot use late pass

Office hours and exam prep sessions
- Will be announced on Piazza

Final exam
- When: Tuesday 5/19, 1:30 PM
- Where: Friend Center 101
- Closed book, closed notes, no electronic devices
Thank you!