CS350: Operating Systems
Debugging

Ali Mashtizadeh

University of Waterloo

Finding and Fixing Bugs

- "An ounce of prevention is worth a pound of cure." - Benjamin Franklin

- Preventative Approaches:
  - Compiler Tools
  - Defensive Programming
  - Static/Dynamic Analysis
  - Runtime Checkers

- Debugging Approaches:
  - Debuggers
  - Log Debugging

Outline

1. Preventative Approaches

2. Debugging Approaches

Compiler Tools

- Compiler's can help you avoid bugs:

- Enable warning and convert warnings into errors.
- `CFLAGS=-Wall -Werror`

- Don't ignore warnings particularly:
  - Uninitialized variables
  - Undefined behavior

- Sometimes `-Wall` enables some benign warnings like unused arguments

Use the Language/Compiler To Detect Bugs

- Don't initialize variables until you need to
  - Allows the compiler to analyze your code
  - Show you cases were you may not intend to use a default value

```
void foo() {
   int status = 0;
   ...
   if (...) {
      status = 1;
      return status;
   }

   return status;
}
```

```
void foo() {
   int status;
   ...
   if (...) {
      status = 1;
      return status;
   }

   /*
    * Warning: use of unassigned
    * local variable.
    */
   return status;
}
```

Use the Language/Compiler To Detect Bugs

- Always check return values
  - Ignoring return values often make debugging hard
  - Happens often to students in our assignments

- Prevent developers from ignoring important results
  - Example: `int pthread_mutex_trylock(...) __attribute__((warn_unused_result));`
  - No correct way to use trylock without checking the return value

- Disable implicit casting
  - Force you to explicitly cast types and think about type safety

Defensive Programming: Asserts

- Use `assert` to check any pre-/post-conditions
  - If you aren't checking if an input is valid
  - Then your assuming a condition

- You can also insert compile time checks `static_assert`

```
void foo(...) {
    assert(precondition ...);

    ...

    assert(postcondition ...);

    return status;
}
```

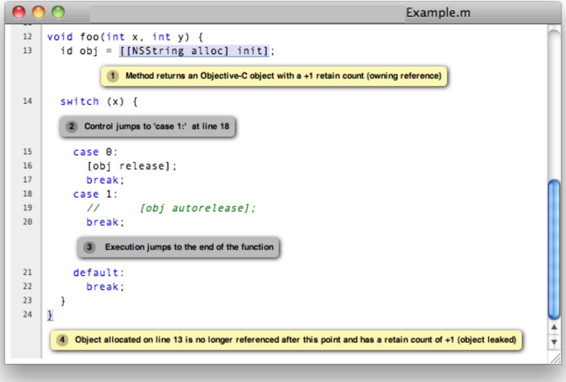
Defensive Programming

- Avoid bool, define flags that aren't easy to mix up
  - Worse: inverting the flag in software layers
  - Use `enum` to define explicit flags

- Use `enum` for switch-case statements
  - Avoid default case if possible
  - Compiler warns of missing enum cases

- Avoid confusing APIs
  - Example: `strncpy` vs. `strlcpy` and `strncat` vs. `strlcat`
  - `strncpy` doesn't null terminate the string when the buffer is too small!

Static Analysis



```
12  void foo(int x, int y) {
13    id obj = [[NSString alloc] init];
```

1 Method returns an Objective-C object with a +1 retain count (owning reference)

```
14    switch (x) {
```

2 Control jumps to 'case 1:' at line 18

```
15      case 0:
16        [obj release];
17        break;
18      case 1:
19        //      [obj autorelease];
20        break;
```

3 Execution jumps to the end of the function

```
21      default:
22        break;
23    }
24  }
```

4 Object allocated on line 13 is no longer referenced after this point and has a retain count of +1 (object leaked)

- Clang Static Analyzer
- See: [KLEE], [Coverity]

Runtime Checkers

```c
#include <pthread.h>
int Global;
void *Thread1(void *x) {
  Global = 42;
  return x;
}
int main() {
  pthread_t t;
  pthread_create(&t, NULL, Thread1, NULL);
  Global = 43;
  pthread_join(t, NULL);
  return Global;
}
```

- ThreadSanitizer, AddressSanitizer, …
- See: [Eraser], [ThreadSanitizer]

Runtime Checkers: ThreadSanitizer

```
% ./a.out
WARNING: ThreadSanitizer: data race (pid=19219)
  Write of size 4 at 0x7fcf47b21bc0 by thread T1:
    #0 Thread1 tiny_race.c:4 (exe+0x00000000a360)

  Previous write of size 4 at 0x7fcf47b21bc0 by main thread:
    #0 main tiny_race.c:10 (exe+0x00000000a3b4)

  Thread T1 (running) created at:
    #0 pthread_create tsan_interceptors.cc:705 (exe+0x00000000c790)
    #1 main tiny_race.c:9 (exe+0x00000000a3a4)
```

- ThreadSanitizer, AddressSanitizer, ...
- See: [Eraser], [ThreadSanitizer]

Outline

1. Preventative Approaches

2. Debugging Approaches

Debuggers

- Debuggers are great!
- Some classes push debuggers because it's an important skill
- Throughout VMware and Ph.D.:
  - Used debuggers to inspect crashes (rarely)
  - Used log debugging for everything else
  - Requires displined use of logging throughout the code

Effective Logging

- Basics
  - Log major operations
  - Turn on/off logging per subsystem
  - Compile out unnecessary logs
  - Timestamp every message
- Every log message should print a unique identifier (e.g., task/object)
  - Use `grep` to quickly filter relevant events
- Dump state on a crash: register signal handlers

Log Debugging Pitfalls

- Three examples of what can go wrong...

- Non-maskable Interrupts, Machine Check Exceptions, etc.
- Logging, Locks and Heisenbugs

Log Debugging Pitfalls: NMIs

- What can go wrong?

- Logging code is fairly complex: `*printf`, console, and serial devices
- Can't use Mutex locks inside of a spinlock region...

- `kprintf` avoids locking
- Console and serial driver implement spinlocks per character
- Similar to a Mutex, but disables interrupts.
- Unfortunately, certain interrupts cannot be disabled
- Result: thread deadlocks with itself

Log Debugging Pitfalls: NMIs

- What can go wrong?

- Logging code is fairly complex: `*printf`, console, and serial devices
- Can't use Mutex locks inside of a spinlock region...

- `kprintf` avoids locking
- Console and serial driver implement spinlocks per character
- Similar to a Mutex, but disables interrupts.
- Unfortunately, certain interrupts cannot be disabled
- Result: thread deadlocks with itself

- Bad choices: potential for deadlocks or expose potential races
- Similar problems can happen in complex software (e.g. with signals)

Log Debugging Pitfalls: NMIs - Solutions

- Solutions:
  - Drop log messages if device locks held
  - Attempt to write to device without locks
  - Attempt to acquire locks using trylock
  - Buffer locks in a lock-free buffer

Log Debugging Pitfalls: NMIs - Solutions

- Solutions:
  - Drop log messages if device locks held
  - Attempt to write to device without locks
  - Attempt to acquire locks using trylock
  - Buffer locks in a lock-free buffer

- Both result in unreliable logging
- Probably you want to attach a debugger

Log Debugging Pitfalls: Locks, Logs and Heisenbugs

- Logging infrastructure uses locks!

Log Debugging Pitfalls: Locks, Logs and Heisenbugs

- Logging infrastructure uses locks!

- Logging can:
  - Serialize operations hiding data races (implicit barriers and locks)
  - Change timing hiding data races

Log Debugging Pitfalls: Locks, Logs and Heisenbugs

- Logging infrastructure uses locks!
- Logging can:
  - Serialize operations hiding data races (implicit barriers and locks)
  - Change timing hiding data races
- What can you do?
  - Reproduce bug with and without logging
  - Look for data races

Summary

- Logs can be reordered if there's buffering
- Logging can deadlock or be dropped silently
- Logging can hide races (it slows you down)