Embedded Software

Debugging

Agenda

- Foreword
- Types of errors
- Debugging
- Error examples
- The art of debugging using gdb
- Other approaches
- Dynamic & Static code analysis tools

Foreword

- Writing correct programs - *honourable mention*
  - Using a development process that facilitates correct programs
    - TDD (Test Driven Development)
    - Extreme programming
    - Just using Test Cases - remember tests on all abstraction levels
    - Performing code review
      - Multiple responsible persons sign off

Types of errors

- The different types of programming errors
- Why is it a big challenge finding errors?

The different types of programming errors

- Syntax and type handling
- Design in terms of algorithms
- Logic errors
  - if, switch, loop constructs...
- Memory errors
  - Null pointers, array bound, stray pointers & leaks
- Interfaces
  - Communication between parts - classes, modules, programs
- Extraneous
  - Hardware failure, software failure in other systems
- Threads and synchronization
  - Deadlocks
  - Priority based (inversion etc.)

Why is it a big challenge finding errors?

- The bug is rare and seemingly non-deterministic - Heisenbug
  - Thread related
    - Timing related manifested as -> wrong incoming events, invalid syncing, data not there anymore, deadlocks
  - Incorrect handling of data resulting in corrupt memory - Symptom and Cause not related
  - External causes
  - They may not happen for long periods of time
  - Incorrect assumptions on data leading to faulty logic

Debugging (& testing)

- The process of finding errors via Debugging
- Proactively reduce errors

The process of finding errors via Debugging

- Reproducibility
- Reduction
- Deduction
- Experimentation
- Experience
- Tenacity

Proactively reduce errors

- Ensure you understand the premiss of the system you are working on
  - Always ensure that you have an idea what a piece of code should do
- Keep interfaces simple and intuitive
- Keep things simple e.g. “*One class - One responsibility*”
- Use asserts (compile or/and runtime)
- Enable compiler warnings - At max if possible
- Use different compilers
- Use “*lint*” or other tool to validate your software
- Test test test - Verify correctness via test cases
  - Write code that is testable... (next semester)
  - All levels of abstraction

Error examples

- Try to find all the error in each examples
- Error types you will encounter
  - Range error
  - Forgotten variables or initalizations
  - Assumptions
  - Typos
  - Threading

Example 1

```
const int MAX_COUNT = 100;

int main(int argc, char* argv[])
{
  int count;

  std::vector<int*> v = getData();

  std::cout << "Count iter: ";
  std::cin >> count;

  int i = 0;

  while (!(i = MAX_COUNT)); {
    std::cout << v[i] << std::endl;
    ++i;
  }
}
```

How many elements are there in v?

A while() loop that ends in ;
Does it do what I want it to?

An assignment in the while condition, is this what I want?

What element type does v contain, what am I writing to stdout

Example 2

```c
const int MAX_SIZE=16;

char *f() {
  char text[MAX_SIZE];
  int res = getResult();

  sprintf(text,"Got result: %d", res);

  return text;
}

int g()
{
  char *p = f();
  printf("f() returns: %s\n",p);
}
```

Can text contain the data it may have to?

Returning a local variable is that good?

Example 3

```
struct GarageDoorOpenReq : public Message
{
  MsgQueue* mq_;
};


void carhandler(CarInfo* ci, Message* msg, unsigned long id)
{
  switch(id)
  {
    case ID_START:
    {
      cout << "Car#" << ci->carId_ << ": Received ID_START" << endl;
      GarageDoorOpenReq* req = new GarageDoorOpenReq;

      cout << "Car#" << ci->carId_ << ": Requesting to enter PLCS" << end;

      //Sending request
      entryQueue.send(ENTRY_OPEN_REQ);
    }
      break;
  }
}
```

We have forgotten to set value mq_

send() method is missing the message to be send

Example 4

```
// Imagine that these variables are in different compilation units
const int DATA_SIZE = 16;
const int CTRL_SIZE = 3;

int ctrl[CTRL_SIZE] = {};
int data[DATA_SIZE] = {};
int running = true;

// these thread - likewise live in their own compilation unit
void thread1()
{
  for(int i = 0; i <= CTRL_SIZE; ++i)
    ctrl[i] = readFromPort();
}

void thread2()
{
  while(running)
  {
    awaitData(); // blocking call
    for(int i = 0; i < DATA_SIZE; ++i)
      writeToDevice(data[i]);
  }
}
```

thread1 writes to data that is private to thread2

Both threads live independent of each other - separate compilation units

Can be extremely difficult to track down

The art of debugging using gdb

- Short intro
- Debugging using gdb
- Cross debugging using gdb
- Enabling core dumps

GDB

- GNU Debugger
  - Written by Richard Stallman in 1986 as part of the GNU System
- Classic debugger with lots of features
- Text based debugger
  - Interactive shell with auto completion
    - Everything is done this way
  - Script support - now python
  - Upcoming support for *checkpoint* semantics
- GUI Frontends
  - Numerous frontends, parsing all actual commands to the gdb program
    - ddd, eclipse, Visual Studio etc.

```
import gdb
def print_node(value):
     frame = gdb.selected_frame()
try:
     val = gdb.Frame.read_var(frame, value)
except:
     print “No such variable”
     return
if str(val.type) == “struct _node *”:
     print “Weight: ” + str(val["weight"])
     print “Tag: ” + str(val["tag"])
else:
     print “Is not a node (” + str(val.type)
+ “)”
....
python print_node(“mynode”)
```

The art of debugging using gdb

- Precondition - compilation
  - Flags -O0 -g
    - Disable optimization and add debug information
  - *g++ -o main* ***-O0 -g*** *main.cpp*
- Why?
  - Missing symbols → Hard to debug
  - Optimization removes functions and variables, inlining etc.
    - Optimization might provoke the problem...

Debugging using gdb

```
stud@ubuntu:/tmp% g++ -O0 -g -o main main.cpp                    21:40
stud@ubuntu:/tmp% gdb ./main                                     21:4
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /tmp/main...done.
(gdb) b badFunc
Breakpoint 1 at 0x80486da: file main.cpp, line 5.
(gdb) r
Starting program: /tmp/main
Bad program...

Breakpoint 1, badFunc () at main.cpp:5
5         int* p = 0;
(gdb) n
6         *p = 1;
(gdb) n

Program received signal SIGSEGV, Segmentation fault.
0x080486e4 in badFunc () at main.cpp:6
6         *p = 1;
(gdb) bt
#0  0x080486e4 in badFunc () at main.cpp:6
#1  0x0804871e in main (argc=1, argv=0xbffff104) at main.cpp:13
(gdb) print p
$1 = (int *) 0x0
(gdb)
```

```cpp
#include <iostream>

void badFunc()
{
  int* p = 0;
  *p = 1;
}

int main(int argc, char* argv[])
{
  std::cout << "Bad program..."
  << std::endl;

  badFunc();

  return 0;
}
```

Printing the contents of variable p

Debugging using gdb - Demonstration time

Debugging using ddd - Demonstration time

Cross debugging

- Prerequisites
  - The program to be tested ready on the target
    - May be stripped - all symbols removed
  - Compiler has relevant files on the target, hack needed if distribution is *NOT* compiled with the compiler being used
  - Relevant libraries which the app may be using on target must be available on host
    - gdb must know where sysroot is:
      - set solib-absolute-prefix /home/stud/setup-scripts/build/tmp-angstrom_2010_x-eglibc/sysroots/beagleboard/

Cross debugging

- Target

```
root@DevKit8000:~# gdbserver localhost:1234 ./TestTimer_d
Process ./TestTimer_d created; pid = 1306
Listening on port 1234
```

- Host

```
stud@ubuntu:~/repo/Development/Code/C_CPlusPlus/Libs/OSApi$ arm-none-linux-gnueabi-gdb bin/arm-linux-gcc-4.2.1/TestTimer_d
GNU gdb (CodeSourcery Sourcery G++ Lite 2007q3-51) 6.6.50.20070821-cvs
Copyright (C) 2007 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-none-linux-gnueabi".
For bug reporting instructions, please see:
<https://support.codesourcery.com/GNUToolchain/>.
..
(gdb) set solib-absolute-prefix /home/stud/DevKit8000Rootfs
(gdb) target remote 10.9.8.2:1234
Remote debugging using 10.9.8.2:1234
0x400007c0 in _start () from /home/stud/DevKit8000Rootfs/lib/ld-linux.so.3
(gdb) b main
Breakpoint 1 at 0x9d5c: file test/TestTimer.cpp, line 96.
(gdb) c
Continuing.
[New Thread 1308]

Breakpoint 1, main (argc=1, argv=0xbed85d64) at test/TestTimer.cpp:96
96          TestTimer t;
(gdb)
```

Break point reached

Enabling core dumps

- Core dump?
  - *A core dump is file dump of the application at the time of its crash*
  - Can be loaded into gdb for inspection
  - Issue the command *ulimit -c unlimited,* to enable
    - Called prior to invoking your command

Using Core Dumps

```
int main()
{
   int* p = NULL;
   *p = 1;
}
```

SEG Fault

Enable core dumps

Compiled with debug

Run and core dumped

Loaded in gdb

Error found :-)

```
stud@ubuntu:/tmp% ulimit -c unlimited
stud@ubuntu:/tmp% g++ main.cpp -g -O0
stud@ubuntu:/tmp% ./a.out
[1]    4128 segmentation fault (core dumped)  ./a.out
stud@ubuntu:/tmp% gdb ./a.out core
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /tmp/a.out...done.
[New Thread 4128]

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /usr/lib/libstdc++.so.6...(no debugging symbols found)...
Loaded symbols for /usr/lib/libstdc++.so.6
Reading symbols from /lib/tls/i686/cmov/libm.so.6...(no debugging symbols four
Loaded symbols for /lib/tls/i686/cmov/libm.so.6
Reading symbols from /lib/libgcc_s.so.1...(no debugging symbols found)...done.
Loaded symbols for /lib/libgcc_s.so.1
Reading symbols from /lib/tls/i686/cmov/libc.so.6...(no debugging symbols four
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
#0  0x080485a4 in main () at main.cpp:6
6         *p = 1;
(gdb)
```

Core dumps

- Things to wary about
  - Can be very large
  - Can take considerable time to create
    - Highly depended on the media whereto they are written
  - Not a silver bullet - corrupt stack → No valid dump
- Their name can be controlled by
  - sysctl -w kernel.core_pattern = core.%e.%p

```
%p:         pid
%<NUL>:     '%' is dropped
%%:         output one '%'
%u:         uid
%g:         gid
%s:         signal number
%t:         UNIX time of dump
%h:         hostname
%e:         executable filename
%<OTHER>: both are dropped
```

Other approaches

- Dynamic code analysis
  - Either compiler *instrumentation* or simple run-time *wrapping*
- Static code analysis
  - Parse code *without* compiling it - exploring possible variable outcomes
    - Flagging no problem, uncertain, guaranteed problem
- ***Beware! Not a silver bullet***

Dynamic code analysis tools

- purify - commercial
- Valgrind, Hellgrind
- efence
- g++ -std=c++11 -fsanitize=undefined (or address, thread, …)
- clang++ -std=c++11 -fsanitize=address (or undefined, thread)
- *and more*

Static code analysis tools

- polyspace - commercial
- pc lint - commercial
- cppcheck
- scan-build (from clang)
- *and more*

Beware! Not a silver bullet

- False positives
- Does not find everything
- Only on testet code
- Dynamic analyzers
  - Changes timing
  - Changes memory allocation patterns
  - Only one sanitiser at a time

Valgrind/hellgrind in more detail (dynamic)

Valgrind/hellgrind in more detail

- Valgrind (memcheck)
  - Features (<url for valgrind>)
    - *Accessing memory you shouldn't, e.g. overrunning and underrunning heap blocks, overrunning the top of the stack, and accessing memory after it has been freed.*
    - *Using undefined values, i.e. values that have not been initialised, or that have been derived from other undefined values.*
    - *Incorrect freeing of heap memory, such as double-freeing heap blocks, or mismatched use of malloc/new/new[] versus free/delete/delete[]*
    - *Overlapping src and dst pointers in memcpy and related functions.*
    - *Memory leaks.*
  - Invoke
    - valgrind <program name> # Lots of options - read man page :-)

Valgrind/hellgrind in more detail

- Callgrind
  - Features
    - Profiling
  - Invoke
    - valgrind --tool=callgrind <program name>
- Nice to know
  - Install and use kcachegrind

Valgrind/hellgrind in more detail

- Helgrind
  - Features
    - *Misuses of the POSIX pthreads API*
    - *Potential deadlocks arising from lock ordering problems*
    - *Data races -- accessing memory without adequate locking or synchronization*
  - Invoke
    - valgrind --tool=helgrind <program name>

Clang sanitisers in more detail (dynamic)

clang sanitisers

- Address
- Memory
- Undefined behaviour
- Thread
- and more…

Clangs - ***memory*** sanitiser (from homepage)

- MemorySanitizer is a detector of uninitialized reads. It consists of a compiler instrumentation module and a run-time library.

- Typical slowdown introduced by MemorySanitizer is 3x.

Clangs - ***address*** sanitiser (from homepage)

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return (to some extent)
- Double-free, invalid free
- Memory leaks (experimental)

Clangs - ***undefined behaviour*** sanitiser (from homepage)

- Using misaligned or null pointer
- Signed integer overflow
- Conversion to, from, or between floating-point types which would overflow the destination
- *and a lot more…*

Cppcheck in more detail (static)

Feature list

- Out of bounds checking
- Memory leaks checking
- Detect possible null pointer dereferences
- Check for uninitialized variables
- Check for invalid usage of STL
- Checking exception safety
- Warn if obsolete or unsafe functions are used
- Warn about unused or redundant code
- Detect various suspicious code indicating bugs
- …

scan-build in more detail - from clang in development (static)

Feature list

- uninitialized arguments, null function pointers
- Check for division by zero
- Check for dereferences of null pointers
- Check that addresses of stack memory do not escape the function.
- Check for uninitialized values used as array subscripts
- Check for assigning uninitialized values
- Check for uninitialized values used as branch conditions
- Check for double-free, use-after-free and offset problems involving C++ delete
- *and more*

Testing

- Lets test this piece of code and see if we compilers/test tools detect a problem
- 2 errors
  - text variable too small
  - return local variable
- Dynamic
  - g++ -fsanitize=address
  - clang-g++ -fsanitize=address
- Static check
  - cppcheck & scan-build

```
const int MAX_SIZE=16;

int getResult() {
  return 10000000;
}

char *f() {
  char text[MAX_SIZE];
  int res = getResult();

  sprintf(text,"Got result: %d", res);

  return text;
}

int main()
{
  char *p = f();
  printf("f() returns: %s\n",p);
}
```

Demo!

Summary

- Debugging is an art
  - It takes practice and skill to master
  - Be aware of the different error types
  - Use structured approach (*follow the 6 bullet points in slide 8)*
- Even better
  - Utilize the compiler to help out