Arctic Region Supercomputing Center

Program Debugging

David Newman

(from Tom Logan

Slides from Ed Kornkven)

Introduction

- **The basic steps in debugging are:**
  - Recognize that a bug exists -- a consequence of:
    - successful ***testing*** (including normal use)
    - checking results
  - Isolate the source of the bug
  - Identify the cause of the bug
  - Determine a fix for the bug
  - Apply the fix and ***test*** it
- **In this talk, we will focus on**
  - Measures for avoiding bugs in the first place
  - Finding them when they do occur

So Many Kinds of Bugs!

- **Incorrect results/output due to:**
  - Logic / algorithmic errors
    - Incorrect loops - for example, infinite loops, off-by-one errors
  - Improper memory reads/writes
    - Pointer errors, array bounds, uninitialized memory references, alignment problems, exhausting memory, stack overflow, memory leaks
  - Misinterpretation of memory
    - Type errors, e.g. when passing parameters
    - Scope/naming errors (e.g., shadowing a global name with a local name)
  - Illegal numerical operations - Divide by zero, overflow, underflow
  - I/O errors
  - Build errors
    - Including source control, Makefile, preprocessor, compiler, linker
  - System errors – libraries, compilers, hardware
- **Poor performance**

Recognizing Bugs Before They Get You

Premise: The easiest bug to find is the one you were already watching for

Establish defensive practices

- **For coding, put error checking in your code, especially to check:**
  - conditions that will vary depending on inputs
  - conditions that should *not* vary
    - e.g. assumptions about function parameter values
  - computations with predictably bad possible effects
    - e.g. floating point exceptions, buffer overflow

Error Checking Facilities

- **E.g. Opening an input file in Fortran (pgf90 compiler)**

```
open (11, file='input_file')
read (11,*) x
```

- **Running this program without first creating "input_file" gives the following error:**

```
PGFIO-F-217/list-directed read/unit=11/attempt to read past end
  of file.
File name = input_file formatted, sequential access record = 1
In source file read_err1.f90, at line number 3
```

- **This message is misleading because there is no such file.**
- **Moreover, running the program creates (an empty) one!**

Fixing the OPEN Error

- **Let's tell the OPEN statement that we expect the file to exist:**

```
open (11, file='input_file', status='old')
read (11,*) x
```

- **Now running the program without first creating "input_file" gives this error:**

```
PGFIO-F-209/OPEN/unit=11/'OLD' specified for file which
does not exist.
File name = input_file
In source file read_err2.f90, at line number 2
```

- **Point: We are able to get a more accurate error message because we gave more information to the program about the expected state.**

Improving the OPEN Error Message

- Many Fortran routines have a *status variable* that will return an error code indicating the status of the call. For example, the OPEN has the IOSTAT option:

```
open (11, file='input_file', status='old', iostat=irc)
if (irc .ne. 0) then
   print*, 'OPEN failed with IOSTAT=', irc, '—stopping.'
   stop
endif
read (11,*) x
```

- We now have control over what happens if the OPEN fails:

```
OPEN failed with IOSTAT=          209 —stopping.
```

- PGI Users Guide (2010) reveals that IOSTAT=209 means:

```
'OLD' specified for file that does not exist
```

Recognizing Bugs Before They Get You

Another example - Buffer overrun in C

```
#include <string.h>
#include <stdio.h>
char name[8] = {'\0'}, password[8] = {'\0'};

int main(int argc, char **argv) {
    strcpy(name, argv[1]);
    printf("name = <%s>, password = <%s>\n", name, password);
}
```

A couple of runs

```
./a.out                          # Bad!
   name = <>, password = <>
   Bus error                     [Mac with gcc]
   Memory fault(coredump)        [Pacman gives no other output]


./a.out 12345678901234567890     # Worse!
   name = <12345678901234567890>, password = <901234567890> [Pacman]
```

Fixing the Buffer Overrun

- **Replace strcpy() with strncpy() to limit the copy**

```
#include <string.h>
#include <stdio.h>
char name[8] = {'\0'}, password[8] = {'\0'};

int main(int argc, char **argv) {
    strncpy(name, argv[1], 8);
    printf("name = <%s>, password = <%s>\n", name, password);
}
```

- **Now execute...**

```
./a.out 12345678901234567890
name = <12345678>, password = <>
```

- **Can you see the error that we introduced?**
- **What (original) error did we leave?**

Recognizing Bugs Before They Get You

- **The C Assertion Facility**

```
#include <assert.h>
assert(exp);
```

where `exp` is an integer expression

- **When this “function” executes…**
  - If **`exp`** evaluates to True (non-zero), do nothing
  - If **`exp`** evaluates to False however, halt and print the message:

```
"assertion \"%s\" failed: file \"%s\", line %d\n", \
        "expression", __FILE__, __LINE__);
```

Using assert()

- **We can use assert() to check for buffer overruns like this:**

```
#include <string.h>
#include <stdio.h>
#include <assert.h>
char name[8] = {'\0'}, password[8] = {'\0'};


int main(int argc, char **argv) {
    assert (strlen(argv[1]) < 8);
    strncpy(name, argv[1], 8);
    printf("name = <%s>, password = <%s>\n", name, password);
}
```

- **Now run it…**

```
pgcc -o buf3 buf3.c
./buf3 12345678901234567890
buf3: buf3.c:10: main: Assertion `strlen(argv[1]) < 8' failed.
Abort(coredump)
```

Using assert()

- **But try our first test**

```
./buf3
    Memory fault(coredump)
```

- **Looks like we're still missing the boat (or bus)**
  - The assertion is too weak -- it isn't enough that the input not be *too long*; it can't be *too short* either

```
assert (strlen(argv[1])> 0);
assert (strlen(argv[1])< 8);
```

- **Try again…**

```
pgcc -o buf4 buf4.c
./buf4
    Memory fault(coredump)
```

- **As Charlie Brown would say, "ARRGGGHHHH!!!"**
  - Looks like we need to add another bug category: the debugging process itself!
  - Why is this still failing?

Using assert()

- **What if I want to use assertions only in the development phase of my project?**
  - Because there is run-time overhead to be paid
  - Easy to disable at compile with #define of NDEBUG symbol
- **Continuing the previous example…**

```
pgcc –DNDEBUG –o buf4 buf4.c
```

```
./a.out 12345678901234567890
    name = <12345678>, password = <>
```

- **NB: assert() may be implemented as a macro**
  - If so, that may affect how parameters are treated
  - See Kate Hedstrom's article in ARSC Newsletter 326

Recognizing Bugs Before They Get You

- **Which brings up another rich source of C bugs - the preprocessor**
  - It's easy to forget what is going on in the preprocessor: *text substitution*
- **E.g.**

```
#include <stdio.h>
#define max(a,b)    (a>b?a:b)

main (int argc, char *argv[]) {
    int x = 20, y = 10, larger;
    larger = max(x++, y++);
    printf("The larger of %d and %d is %d\n", x, y, larger);
}
```

C Macro Abuse (cont.)

- **Code looks good (doesn't it always?) but output does not…**

```
gcc -g preproc.c
./a.out
  The larger of 22 and 11 is 21
```

- **We can see what the compiler "sees" (after the preprocessor has finished with it) using a compiler flag**

```
gcc -E preproc.c > preproc.out
```

C Macro Abuse (cont.)

- **Here is the C code produced by the preprocessor:**

```
main (int argc, char *argv[]) {
 int x = 20, y = 10, larger;
 larger = (x++>y++?x++:y++);
 printf("The larger of %d and %d is %d\n", x, y,
  larger);
}
```

- **No wonder we got the output we did! How do we fix it?**
  - Don't autoincrement in the macro call

C Macro Abuse (cont.)

- **Another trap in this macro, illustrated by another example:**

```
#define DBL(a)     (a*2)
x = DBL(y+1)
```

- **Expands to**

```
x = y+1*2          #Wrong!
```

- **In general, avoid problems with unintended combinations of macro parameter expansions by parenthesizing all occurrences of parameters in the macro definition**

```
#define DBL(a)     ((a)*2)
#define max(a,b)   ((a)>(b)?(a):(b))
```

Some Final Observations on Bug Prevention

- **Software engineering approaches can help catch bugs early. One example:**
  - Extreme Programming (XP)
    - 12 "Core Practices", including:
      - programming pairs
      - frequent small releases
      - continuous testing
        - » unit tests and acceptance tests
        - » write tests first
      - continuous integration
        - » integrate changes daily
        - » all tests must pass before and after integration
- **Notice the close connection between testing and quality (and therefore, to debugging)**

Recognizing Bugs Before They Get You

To summarize: “Safety First”

- Assume errors are in your code and data
- Practice defensive programming and check your data
- Make use of available language and compiler features

Recognizing Bugs After They Get You

- **As we saw earlier, there are lots of ways to get you!**
  - Compilation errors
    - Including Makefile, preprocessor, compiler, linker
  - Improper memory reads/writes
    - Pointer errors, array bounds, uninitialized memory references, alignment problems, exhausting memory, memory leaks
  - Misinterpretation of memory
    - Type errors, e.g. when passing parameters
    - Scope/naming errors (e.g., shadowing a global name with a local name)
  - Illegal numerical operations
    - Divide by zero, overflow, underflow
  - Infinite loops
  - Stack overflow
  - I/O errors
  - Logic / algorithmic errors
  - Poor performance

Recognizing Bugs After They Get You

Each of these kinds of errors merit an hour of discussion; we will touch on some here

- **Build errors**
  - Version errors
    - E.g., compiling the wrong version of a file; *losing* the new version; not remembering *why* you made a new version
  - Makefile
    - E.g., assuming a file is being recompiled when it isn't
  - Preprocessor
    - Oftentimes these spill into compiler errors
    - Suspect these if output is wrong and a macro is involved
  - Compiler, linker
    - Mostly easy because the computer finds the errors for you
    - A common version is "name-mangling" errors, esp. when mixing Fortran and C and/or libraries

Improper memory reads/writes

- **Run-time memory errors in Unix cause two broad kinds of errors**
  - Bus error -- the memory hardware was unable to perform a memory address request
    - detected by hardware
    - accessing a memory address that doesn't exist; or,
    - accessing memory starting at an address that isn't on a boundary appropriate to the data type
    - E.g., this will cause a bus error on some machines

```
double *xp;
char *cp;
cp = malloc(sizeof(char)*40);
xp = (double *) (cp+1);
```

Improper memory reads/writes (cont.)

- Segmentation fault
  - detected by the operating system
    - the program attempted to access memory that is outside the user's (virtual) data area
    - access to memory in an illegal way -- e.g., write to read-only

```
scanf("%d", number);  /* should be &number */
```

Improper memory reads/writes (cont.)

- **Pointer errors, invalid free(), uninitialized references and memory leaks can be reliably caught by memory reference monitoring packages**
- **Examples:**
  - Valgrind (http://valgrind.org/)
    - Six production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler
  - http://en.wikipedia.org/wiki/Memory_debugger gives a nice list of alternative packages

Improper memory reads/writes (cont.)

- **Exhausting memory**
  - Check the result of malloc()
  - malloc() returns NULL if there is an error

```
if ((ptr = malloc(n_objects * sizeof(object)) == NULL)
        { /* error handling here */ }
```

  - True? Linux does lazy allocation – no error until used!
- **Memory leaks**
  - Not freeing (and forgetting about) memory that is no longer used
  - Like a water leak, a little bit over a long time can do lots of damage
- **Array bounds errors**
  - Compiler-inserted run-time checks -- e.g.,

  `pg90 -C` …

Illegal Instruction

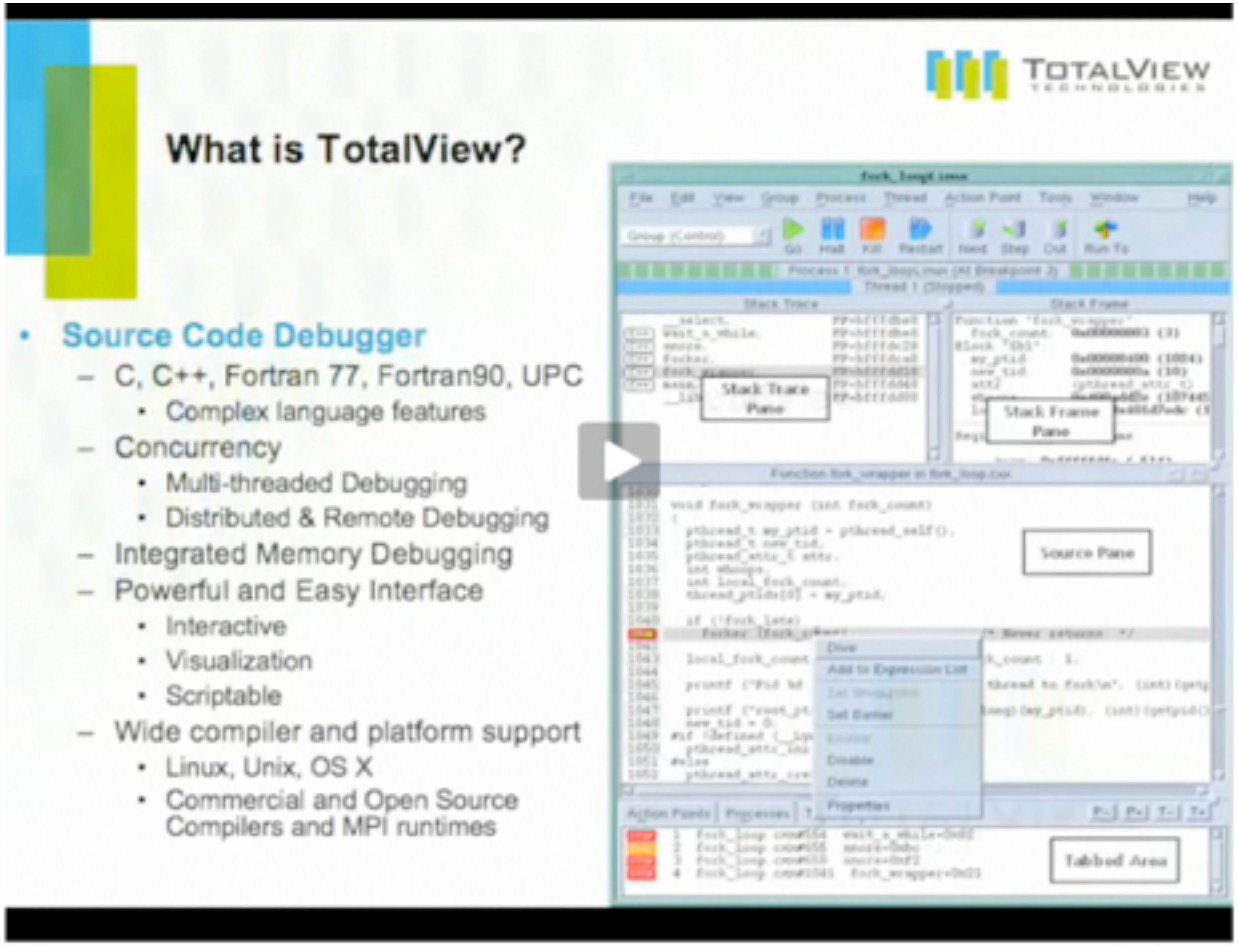
- **Coming "back in style"**
  - Mixture of processors on a machine (e.g. Copper)
  - ➔ mixture of instruction sets
- **How to get an illegal instruction error**
  - Compile for Interlagos processor
  - Execute on Istanbul
- **How to prevent an illegal instruction error**
  - Load the xtpe-istanbul module before compiling

Recognizing Bugs After They Get You

- **Tried-and-true generic bug-hunting: binary search**
  - Can use a variant to find difficult compile-time bugs -- delete code instead of inserting print statements
  - Downside: you have to modify the program (inserting prints), then you must remove those prints
- **For general-purpose bug finding in a crashing program, a debugger is often helpful**
  - Start the program in the debugger and let it run until it crashes
  - What this buys you
    - The program stops at the crash site
    - You can then browse the program's state at the time of the crash
    - Especially effective if the program's symbols are included with the executable program (by compiling with the -g option)
  - Downside: you might be modifying the program (by changing compiler options!)

Debuggers

- **gdb, dbx**
  - Comes with Unix
  - May not work for parallel codes
  - Example and discussion:
    - **http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html**
- **TotalView, DDT (commercial)**
  - The major vendors of debuggers for parallel codes
  - GUI front end
- **pgdbg**
  - Portland Group debugger – on Pacman



http://www.roguewave.com/products/totalview-family/totalview/resources/videos.aspx

Arctic Region Supercomputing Center

Debugging Applications

November 20, 2012

Ed Kornkven

kornkven@arsc.edu

Segmentation Fault

_pmiu_daemon(SIGCHLD): [NID 00067] [c0-0c2s1n1] [Mon Nov 19 17:19:01 2012] PE RANK 13 exit signal Segmentation fault

_pmiu_daemon(SIGCHLD): [NID 00066] [c0-0c2s1n0] [Mon Nov 19 17:19:01 2012] PE RANK 9 exit signal Segmentation fault

[NID 00067] 2012-11-19 17:19:01 Apid 24645: initiated application termination

Application 24645 exit codes: 139

Application 24645 resources: utime ~0s, stime ~0s

- **To locate the source of this error, use a core file**

For Core Files: ulimit -c

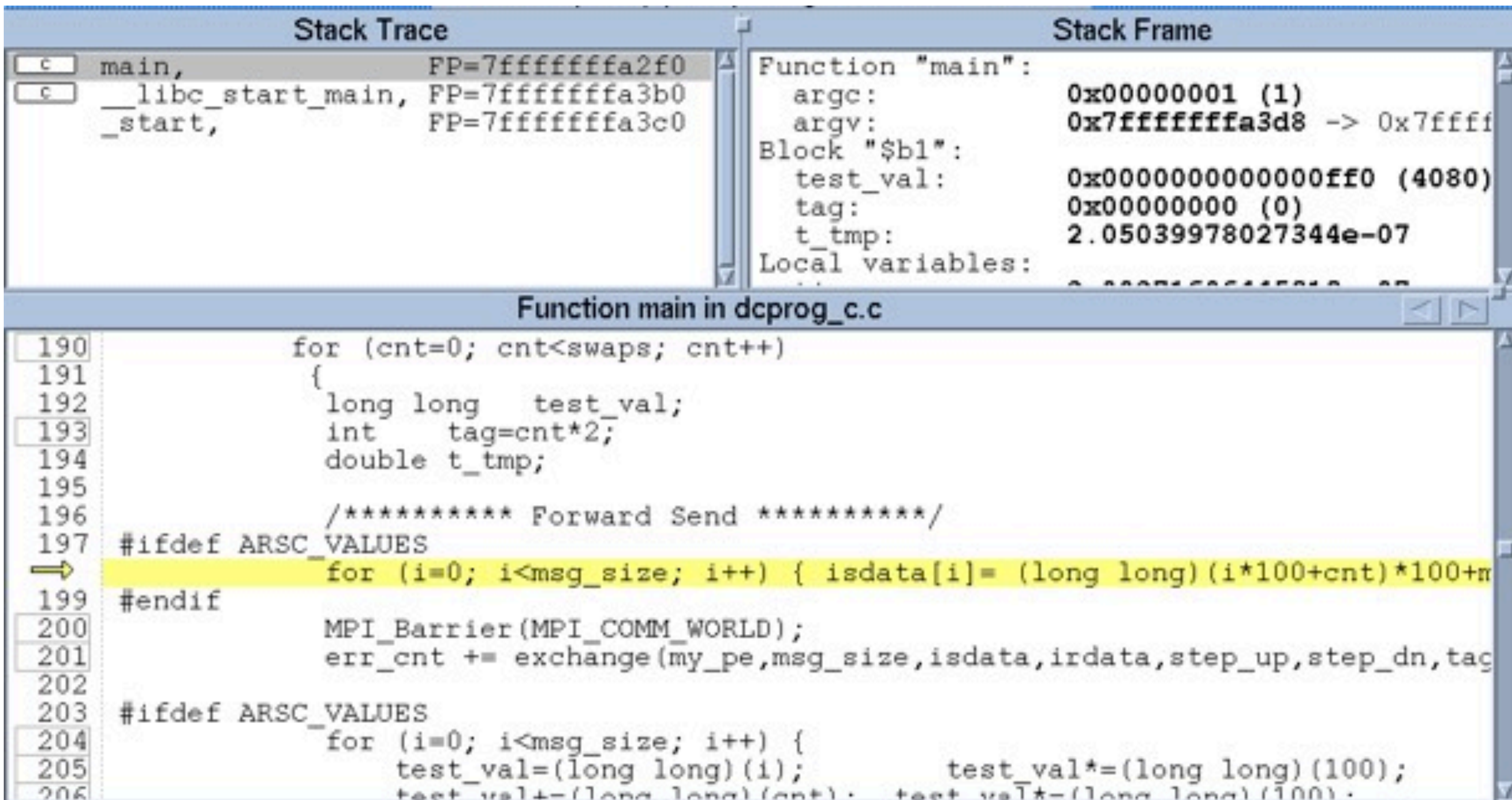
```
fish1> ulimit -a
core file size          (blocks, -c) 1
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
…
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 129125
virtual memory          (kbytes, -v) 13228800
file locks                      (-x) unlimited
```

- **fish1> ulimit -c unlimited**

gdb ./dcprog_c core.nid00066.dcprog_c.3872

```
fish1> gdb ./dcprog_c core.nid00066.dcprog_c.3872
GNU gdb (GDB) SUSE (7.3-0.6.1)
Reading symbols from /import/c/w/kornkven/Phys693/
DCPROG/code/dcprog_c...done.
[New LWP 3872]
Cannot access memory at address 0x9507c0258
(gdb) where
#0  0x0000000000400cb4 in main (argc=1,
argv=0x7fffffffa3d8) at ./dcprog_c.c:198
(gdb) print i
$1 = 4087944
(gdb) print msg_size
$2 = 4960144
```

totalview ./dcprog_c core.nid00066.dcprog_c.3872



TotalView Overview

- **Provides debugging capabilities for parallel and multithreaded codes**
- **Runs on most HPC platforms**
  - Available on Pacman and Fish
- **Has both GUI and command-line interfaces**
- **Supports C/C++, Fortran and mixed languages**
- **Debugs MPI, OpenMP and mixed**

Compiling for TotalView

- Compile with -g for symbol table support
- If possible, turn off optimization for more accurate source mapping

TotalView GUI on Fish

- `% ssh -X -Y username@fish1.arsc.edu`
- `fish1 % qsub -q standard -l nodes=2:ppn=12 -X -I`
- `fish-compute % cd $PBS_O_WORKDIR`
- `fish-compute % module load xt-totalview`
- `fish-compute % totalview aprun -a -n 24 ./dcprog_c`

Basic TotalView Functions

- **View source code and program counter**
  - For any process or thread
- **Set breakpoints**
  - A place in the code at which execution pauses
- **Examine variable contents**
  - Including "diving" into complex data structures
- **Execute in increments of lines or functions**
- **Change variable values**
- **"Watch" variables for changes in value**

References and Further Information

- **ARSC web pages**
  - http://www.arsc.edu/support
- **TotalView videos from RogueWave**
  - http://www.roguewave.com/products/totalview/resources/videos.aspx
- **A general debugging tutorial**
  - http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html
- **LLNL TotalView tutorial**
  - https://computing.llnl.gov/tutorials/totalview/