

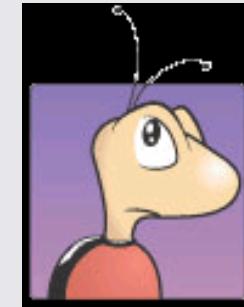
CSE 380: Lecture 9

Debugging

Dr. Christopher Simmons
csim@ices.utexas.edu

Debugging Scientific Applications

- Motivation for developing good debugging skills:
 - Unless you are from a new planet, you will introduce bugs at some point in your code
 - Even if you use community applications written by others, they will introduce bugs
 - And yes, commercial applications have bugs too
- Extra problems:
 - As scientific researchers, we cannot simply concern ourselves with bugs that prevent the application from running
 - We actually care deeply about the accuracy and repeatability of the result (*e.g. negative density values in a flow code are probably bad*)
 - Stability is a concern (*e.g. an iterative based solver that never converges*)
- The addition of strong debugging skills to your toolbox will greatly enhance your efficiency and add confidence to the numerical results



Defensive Programming Tips

- One of the best defenses against runtime bugs is to use basic defensive programming techniques:
 - Check ***all*** function return codes for errors
 - Check ***all*** input values controlling program execution to ensure they are within acceptable ranges (even those from flat text files in which you know there could not possibly be an error)
 - Echo all physical control parameters to a location that you will look at routinely (eg. *stdout*). Better yet, save all the parameters necessary to repeat an analysis in your solution files (*remember the metadata options in netCDF and HDF?*)
 - In addition to monitoring for obvious floating-point problems (eg. *divide-by-zero*), check for non-physical results in your simulations (eg. *supersonic velocities predicted in a low-speed aerodynamic simulation*)



Defensive Programming Tips

- Additional suggestions:
 - Maintain test cases for regression testing - is there an analytic test case you can benchmark against?
 - Use version control systems (Subversion, git, etc)
 - Maintain a clean, modular structure with documented interfaces: goto's, long-jumps, clever/obscure macros, etc. are dangerous over the long haul
 - Why not include some comments? Your colleagues will thank you and it just might save your dissertation when you are revisiting a tricky piece of code after a year or two
 - Strong error checking is the mark of a sage programmer and will give you more confidence in your numerical results

Defensive Programming

- Q: Isn't checking all the error codes a waste of time?
- A: It is substantially less wasteful than long debugging sessions which could have been avoided by simple error checks
- Did we mention that useful error checks are the mark of a sage programmer? - at an **absolute minimum**, please check your memory allocations (all of them, each and every time):

```
p = (float *)calloc(nnodes+1,sizeof(float));
if(p == NULL)
{
    printf("Allocation error for p!\n");
    exit(1);
}
```

C

```
allocate(buf(na), stat = ierror)
if(ierror > 0) then
    print*, 'ERROR: Unable to allocate array buf'
    stop
endif
```

Fortran

Defensive Programming

- C++: can leverage exception handling for convenient memory checking

```
try
{
    vector<double> data(blocksize);
    pool.reserve(max_num_records);

    for(int i=0;i<max_num_records;i++)
        pool.push_back(data);

    if(use_disk_overflow)
        data_tmp = new double[blocksize];
}

catch(const std::bad_alloc& ex)
{
    grvy_printf(GRVY_ERROR,"Unable to allocate raw storage pool...exiting.\n");
    exit(1);
}
catch(...)
{
    grvy_printf(error,"%s (p%5i): Unknown exception...exiting\n",prefix,mpi_rank);
    exit(1);
}
```

C++

Defensive Programming

- An easy defensive strategy for Fortran programmers is to use **IMPLICIT NONE** in all your routines (and specifically typecast all used variables). Avoids any undesired conversions
- Be sure to initialize all variables and arrays that require it (don't count on the architecture/OS to do this for you)
- During the testing and verification phase, make use of available compiler options to proactively flag potential errors:
 - Intel Fortran Examples:
 - check all** - enable runtime checks for out-of-bounds array subscripts, uninitialized variables, etc
 - warn all** - display all relevant warning messages
 - warn errors** - tells the compiler to change all warning-level messages into error-level messages
 - fpe0** - tells the compiler to abort when any floating point exceptions occur
 - GCC flags to display all warnings and catch errors:
 - Wall, -Wextra, -Wshadow, -Wunreachable-code**

Consult your compiler documentation for available runtime checks

Defensive Programming Example

- Consider the following example code ([problem.c](#)):

```
int main()
{
    int a, b;
    int x1, x2;

    if (a = b)
        printf("%d\n", x1);
    return 0;
}
```

- > gcc -o problem problem.c
> ./problem
1074729080

Any ideas?

Defensive Programming Example

Now, use the “-Wall” option to have the compiler point out possible trouble spots

```
> gcc -Wall -o problem problem.c
problem.c: In function main:
problem.c:6: warning: suggest parentheses around assignment
used as truth value
problem.c:7: warning: implicit declaration of function printf
problem.c:7: warning: incompatible implicit declaration of
built-in function printf
problem.c:4: warning: unused variable x2
```

**Warnings along with line
numbers are provided**

```
1      int main()
2      {
3          int a, b;
4          int x1, x2;
5
6          if (a = b)
7              printf("%d\n", x1);
8          return 0;
9      }
```

Defensive Programming

- Provide one or more levels of instrumentation in your code for debugging (eg. a *debug mode* and a *verbose debug mode*)
- C/C++ programmers should take advantage of the `assert` macro to ensure values fall within appropriate ranges

```
> gcc -o macro macro.c
> ./macro
> macro: macro.c:12: main: Assertion `n <= 100' failed.
Abort
> gcc -DNDEBUG -o macro macro.c
> ./macro
>
```

```
#include <stdio.h>
#include <cassert.h>

int main()
{
    int n;
    float x[100];
    n = 1000;

    /* Assert that n < 100 */

    assert ( n < 100);

    return 0;
}
```

C

Defensive Programming

- Q: What is the equivalent of assert in Fortran?
- You are on your own, but it is easy for programmers to improvise (some pre-processing is handy)
- This looks like mixed code, how do we compile this?

```
program main
    implicit none
    integer n
    real x(100)
    n = 1000

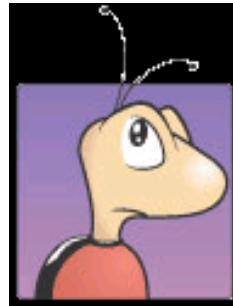
#ifdef DEBUG
    if ( n > 100) then
        print*, ' Assertion (n <= 100) is false'
        print*, ' File: ',__FILE__,' Line: ',__LINE__
        stop
    endif
#endif

stop
end
```

Fortran

```
> ifort -DDEBUG -cpp macro.f
> ./a.out
Assertion (n <= 100) is false
File: macro.f Line:          10
```

Bug-a-boos



Bug Identification

- Common instances in which bugs identify themselves:
 - Build errors (Makefile, preprocessor, compiler, linker)
 - Improper memory reads/writes
 - pointer errors, array bounds overruns, 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

Bug Identification

- What are the symptoms if your application has a memory bug?
 - wrong answers derived when using values from incorrect memory space
 - application behaves differently when different levels of optimization are applied; a classic memory bug symptom is as follows:
 - you compile with full optimization (-O3 for example) and your code crashes unexpectedly
 - you disable all optimization (-O0 for example) and the code runs fine
 - adding additional print statements in the program to try and isolate the bug seems to make it disappear
 - application receives an unexpected symbol and terminates
- We need to understand what it means for our application to receive an extern signal

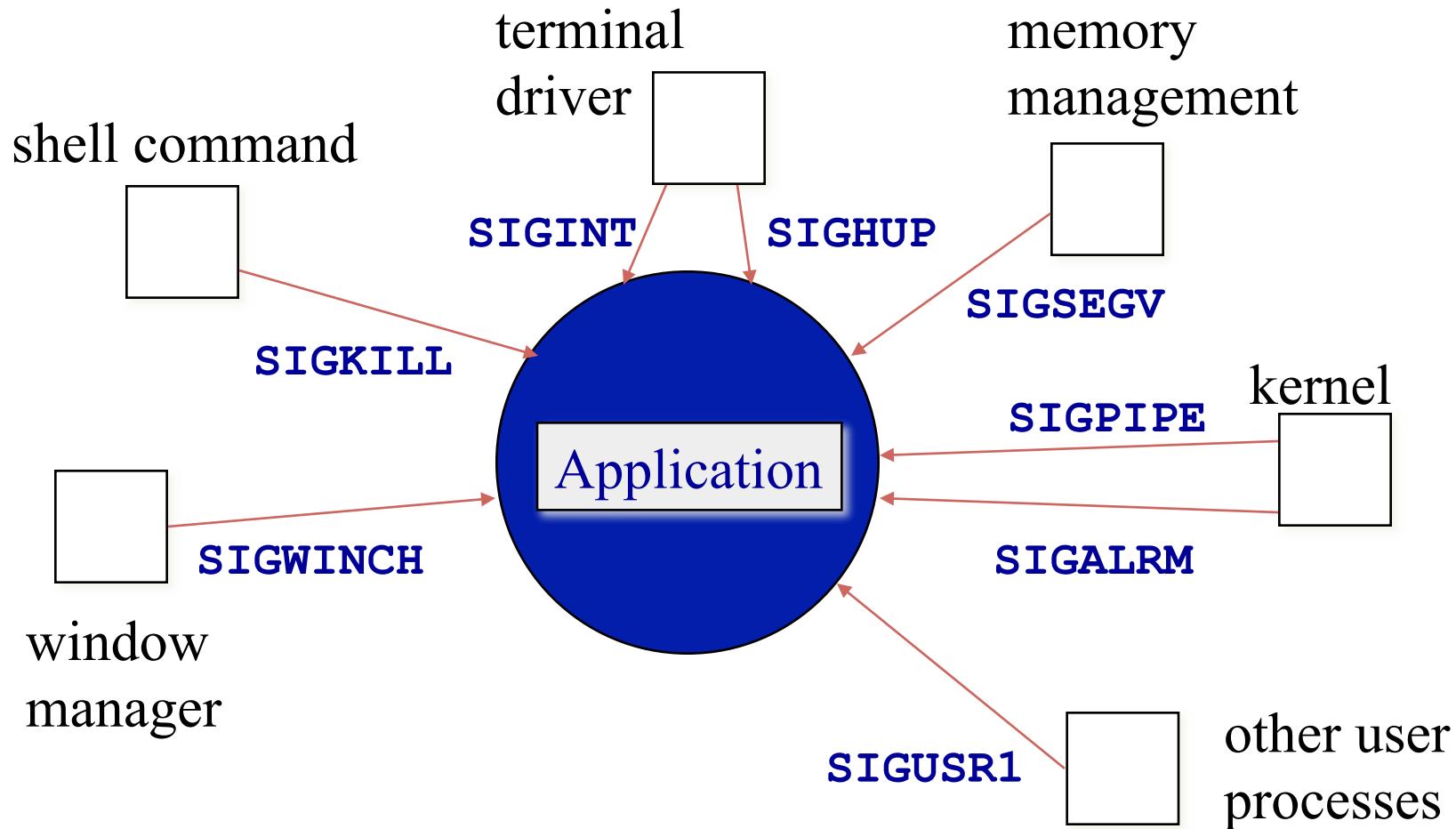
Signals

- A signal is an *asynchronous* event which is delivered to a process.
- Asynchronous means that the event can occur at any time
 - may be unrelated to the execution of the process
 - e.g. user types ctrl-C, or the operating system detects an error and sends a signal to your application

Common Signal Types

• <i>Name</i>	<i>Description</i>	<i>Default Action</i>
SIGINT	Interrupt character typed	terminate process
SIGQUIT	Quit character typed (^\\)	create core image
SIGKILL	Kill signal	terminate process
SIGFPE	Floating exception	create core image
SIGSEGV	Invalid memory reference	create core image
SIGPIPE	Write on pipe but no reader	terminate process
SIGALRM	alarm() clock ‘rings’	terminate process
SIGUSR1	user-defined signal type	terminate process
SIGUSR2	user-defined signal type	terminate process
:		
• See man 7 signal for more information		

Common Signal Sources for Applications



Core Dumps

- Recall that the default action for a SIGSEGV was to create a core image and abort
- What in the world is a core image?
 - a **core dump** is a record of the raw contents of one or more regions of working memory for an application at a given time
 - commonly used to debug a program that has terminated abnormally
 - Note: if your application is “segfaulting” and you are not getting a core file, you may need to alter your shell limits. For example:

```
> unlimit coredumpsize    (tcsh)
> ulimit -c unlimited     (bash)
```
 - Main benefit of a core file is that a post-mortem analysis can be performed on an application that failed. If the symbol table was included, you can backtrace to exactly where the application was when the exception occurred.
 - Note: the location of an exception may not be the original location of a bug (particularly for memory bugs)

Debugging Process

- We recognize that defensive programming can greatly reduce debugging needs, but at some point we all have to roll up our sleeves and track down a bug
- The basic steps in debugging are straightforward in principle:
 - Recognize that a bug exists
 - 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 practice, these can be difficult for particularly pesky bugs; hence, we need some more tools at our disposal (a debugger)

Standard Debuggers

- Command line debuggers are powerful tools to aid in diagnosing problematic applications and are available on all Unix architectures for C/C++ and Fortran
- Example debuggers:
 - Linux: gdb (or idbc with Intel compilers)
 - AIX: dbx
 - SUN: dbx
- The basic use of these debuggers is as a front-end for stepping through your application and examining variables, arrays, function returns, etc. at different times during the execution
- Gives you an opportunity to investigate the dynamic runtime behavior of the application

Note: we will focus primarily on gdb, but concepts and syntax are similar in dbx, and idbc

Debugging Basics

- For effective debugging a couple of commands need to be mastered:
 - show program backtraces (the calling history up to the current point)
 - set breakpoints
 - display the value of individual variables
 - set new values
 - step through a program

Debugging Basics

- A **breakpoint** is a pseudo instruction that the user can insert at any place into the program during a debugging session
- Conceptually, the execution is controlled by the debugger and the debugger will interpret the breakpoints
- When execution crosses a breakpoint, the debugger will pause program execution so that you can:
 - inspect variables,
 - set or clear breakpoints, and
 - continue execution

Debugging Basics

- The notion of a **conditional breakpoint** also exists in which additional logic can be associated with the breakpoint
- When a conditional breakpoint is crossed during execution, the program will pause only if the breakpoint's break condition holds
- **Example break conditions:**
 - A given expression is true
 - The breakpoint has been crossed N times ("hit count") - this is very handy when you know something bad is happening on a particular iteration
 - A given expression has changed its value

Debugging Basics

- For debugging sessions, you should compile your application with extra debugging information included (eg. the symbol table)
- The symbol table maps the binary execution calls back to the original source code definitions
- To include this information, add “-g” to your compilation directives:

```
> gcc -g -o hello hello.c
```

Running GDB

- gdb is started directly from the shell
- You can include the name of the program to be debugged, and an optional core file:
 - `gdb`
 - `gdb a.out`
 - `gdb a.out corefile`

spawns a new instance of ./a.out

examines trapped state in corefile
- gdb can also attach to a program that is already running; you just need to know the PID associated with the desired process
 - `gdb a.out 1134`

useful if an application seems to be slow or stuck and you want to see what it is doing currently

gdb Basics

- Common commands for gdb:
 - `run` - starts the program; if you do not set up any breakpoints the program will run until it terminates or core dumps - program command line arguments can be specified here
 - `print` - prints a variable located in the current scope
 - `next` - executes the current command, and moves to the next command in the program
 - `step` - steps through the next command. Note: if you are at a function call, and you issue `next`, then the function will execute and return. However, if you issue `step`, then you will go to the first line of that function
 - `break` - sets a break point.
 - `continue` - used to continue till next breakpoint or termination

Note: shorthand notations exist for most of these commands: eg. 'c' = continue

gdb Basics

- More commands for gdb:
 - `list` - show code listing near the current execution location
 - `delete` - delete a breakpoint
 - `condition` - make a breakpoint conditional
 - `display` - continuously display value
 - `undisplay` - remove displayed value
 - `where` - show current function stack trace
 - `help` - display help text
 - `quit` - exit gdb

gdb Basics

- Consider the following C code for subsequent examples (hello.c):

```
#include <stdio.h>
void foo();

int main()
{
    printf("inside main\n");
    foo();
    return;
}

void foo()
{
    int i, total=0;
    printf("inside foo\n");
    for(i=0;i<1000;i++)
        total += i;
}
```

Example GDB Session

```
> gcc -g -o hello hello.c
> gdb ./hello
GNU gdb Red Hat Linux (6.3.0.0-1.134.fc5rh)
Copyright 2004 Free Software Foundation, Inc.

(gdb) run
Starting program: /home/csim/cse380/hello
inside main
inside foo

(gdb) break main
Breakpoint 1 at 0x8048384: file hello.c, line 5.

(gdb) run
Starting program: /home/csim/cse380/hello

Breakpoint 1, main () at hello.c:5
5      {
(gdb) where
#0  main () at hello.c:5
```

Example GDB Session (continued)

```
(gdb) break foo
Breakpoint 2 at 0x80483b5: file hello.c, line 13.
(gdb) cont
Continuing.
inside main

Breakpoint 2, foo () at hello.c:13
13      int i, total=0;

(gdb) list
8          return;
9      }
10
11      void foo()
12  {
13          int i, total=0;
14          printf("inside foo\n");
15          for(i=0;i<1000;i++)
16              total += i;
17      }
```

Example GDB Session (continued)

```
(gdb) cont
Continuing.
inside foo

(gdb) delete breakpoints 1 2
(gdb) break hello.c:16
Breakpoint 3 at 0x80483d1: file hello.c, line 16.

(gdb) condition 3 i==401
(gdb) run
Starting program: /home/csim/cse380/hello
inside main
inside foo
Breakpoint 3, foo () at hello.c:16
16          total += i;
(gdb) print i
$1 = 401
(gdb) where
#0  foo () at hello.c:16
#1  0x080483a6 in main () at hello.c:7
```

Floating Point Checks

- Different compilers have various options for handling floating point exceptions
- You can also check for the existence of undesirable values directly in your code (e.g. *NAN*, “*not-a-number*”)
- This is a good practice to test periodically in order to preemptively trap a diverged solution

Floating Point Checks

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double x, y, z;

    x = -4.2;
    y = 0.0;

    // Divide by zero.

    z = x/y;
    printf("z = %g\n", z);

    // Imaginary number.

    z = sqrt(x);
    printf("z = %g\n", z);

    return 0;
}
```

C

- Consider the example C code to the left (nan.c)
- What kind of output are we going to get when we execute this code?

```
> gcc -o nan nan.c -lm
> ./nan
z = -inf
z = nan
```

Floating Point Checks

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    double x, y, z;

    x = -4.2;
    y = 0.0;

    // Divide by zero.

    z = x/y;
    if( finite(z) )
        printf("z = %g\n", z);
    else
    {
        printf("Error: non-finite (%g)\n", z);
        return 1;
    }

    .
}
```

C

- In C, we can query whether a floating point number is *infinite* or *not-a-number* via the following functions:

```
#include <math.h>
```

```
int isinf(double value);
int isnan(double value);
int finite(double value);
```

```
> gcc -o nan nan.c -lm
```

```
> ./nan
```

```
Error: non-finite (-inf)
```

Floating Point Checks

```
program main
  real *8 x, y, z
  x = -4.2
  y = 0.0

  ! Divide by zero.

  z = x/y
  if ( isnan(z) ) then
    print*, 'z is not-a-number'
  endif

  print*, 'z = ', z
  ! Imaginary number.

  z = sqrt(x)
  if ( isnan(z) ) then
    print*, 'z is not-a-number'
  endif
  print*, 'z = ', z

  stop
end program main
```

Fortran

- A similar `isnan` function is available as an intrinsic in most Fortran compilers
- However, no default `finite` function

```
> ifort -o nan nan.f90
> ./nan
z = -Infinity
z is not-a-number
z = NaN
```

Debugging

- Recall the example C code from our gdb example (`hello.c`):

```
#include <stdio.h>
void foo();

int main()
{
    printf("inside main\n");
    foo();
    return;
}

void foo()
{
    int i, total=0;
    printf("inside foo\n");
    for(i=0;i<1000;i++)
        total += i;
}
```

gdb and Emacs

- Emacs has a lot of functionality to support software development and debugging:
 - Supports compilations (eg. can access your Makefile)
 - Can run gdb directly
 - Can integrate with your revision control system (eg. *svn*)
- To illustrate, consider the *hello.c* example and a trivial Makefile

```
all: hello

hello: hello.c
        gcc -g -o hello hello.c
```

gdb and Emacs

- To work within emacs, load up the src file:
`> emacs hello.c`
- Then, to compile, issue **M-x compile** which will run make -k by default
- emacs will create a new window for you and show you the result of your compilation
- To run the program within the debugger, issue **M-x gdb** and provide the name of your executable (eg. *hello*)
- emacs will then provide you an interface to gdb and show you the current execution location in your src code

gdb and Emacs

```
File Edit Options Buffers Tools Gud Complete In/Out Signals Help
#include <stdio.h>
void foo();

int main()
=>
    printf("inside main\n");
    foo();
    return;
}

--u---F1  hello.c          (C Abbrev)--L5--Top-----

(gdb) break main
Breakpoint 1 at 0x8048384: file hello.c, line 5.
(gdb) run
Starting program: /home/karl/cs395t/hello
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0x24f000

Breakpoint 1, main () at hello.c:5
(gdb)
--uu:***-F1  *gud-hello*          (Debugger:run)--L43--Bot-----
```

gdb and Emacs

```
File Edit Options Buffers Tools Gud Complete In/Out Signals Help
void foo()
{
    int i, total=0;
    printf("inside foo\n");
    for(i=0;i<1000;i++)
=>    total += i;
}

--u---F1  hello.c      (C Abbrev)--L16--Bot-----
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0x24f000

Breakpoint 1, main () at hello.c:5
(gdb) step 8
inside main
inside foo
(gdb) print i
$1 = 1
(gdb)
--uu: **-F1  *gud-hello* |      (Debugger:run)--L48--Bot-----
```

Memory Debugging

- **gdb** and **dbx** are very useful for querying the runtime behavior of your application; however, it can still be difficult to trap memory errors
- Additional runtime checks and libraries are available to help in this area for C/C++ applications that utilize malloc:
 - glibc on Linux
 - dmalloc library
 - Electric Fence
 - valgrind
- The runtime checks can have a dramatic impact on performance (a slow-down of 2-50X is not unrealistic depending on the level of checking desired)
- Runtime memory checks should generally be used in a debugging/regression testing state (not for production runs)

Memory Debugging - glibc

- glibc malloc checks:
 - Recent versions of Linux libc (later than 5.4.23) and GNU libc (2.x) include a malloc implementation which is tunable via environment variables
 - When **MALLOC_CHECK_** is set, a special (less efficient) implementation of malloc is used which is designed to be tolerant against simple errors:
 - double calls of free() with the same argument
 - overruns of a single byte
 - Options:
 - **MALLOC_CHECK_=0** -> Any detected heap corruption is silently ignored
 - **MALLOC_CHECK_=1** -> a diagnostic is printed on stderr
 - **MALLOC_CHECK_=2** -> abort() is called immediately

useful in tracking original cause of a code crash

Memory Debugging - dmalloc

- The *dmalloc* library has been designed as a drop in replacement for the system's malloc, realloc, calloc, free and other memory management routines with configurable runtime options
- Debugging options include memory-leak tracking, fence-post write detection, file/line number reporting, and general logging of statistics
- Library is reasonably portable (eg. *Linux*, *AIX*, *HPUX*, *Irix*, *Solaris*)
- Basic usage is to re-link your application against the dmalloc library to override default memory allocations

Memory Debugging - *dmalloc*

General Steps for using *dmalloc*:

- Download or build the *dmalloc* library (www.dmalloc.com)
- Setup a *dmalloc* function for your shell:
 - **BASH**: add the following to your `~/.bashrc`
`function dmalloc { eval `command dmalloc -b $*` ; }`
 - **TCSH**: add the following to your `~/.cshrc`
`alias dmalloc 'eval `dmalloc -c !*`'`
- Relink your application against the *dmalloc* library (you may want to include “*dmalloc.h*” if you want to report the file/line numbers of calls that generate problems)
- Enable debugging features by using the shell functions to set runtime parameters

Memory Debugging - Electric Fence

- Electric Fence (efence) stops your program on the exact instruction that overruns (or underruns) a malloc() memory buffer
- It works by using the virtual-memory hardware to create a red-zone at the border of each of your buffers - if you touch the red zone with a memory bug, your program stops
- Basic usage is to re-link your application against the efence library and rerun

Memory Debugging - Electric Fence

General Steps for using *efence*:

1. Download or build the efence library: (
<http://directory.fsf.org/ElectricFence.html>)
2. Relink your application against the efence library
(**-l`efence`**)
3. Rerun the application (either standalone, or thru a debugger for more information); again, this will slow down the execution significantly

Memory Debugging - Valgrind

- Valgrind is a collection of tools for performing dynamic analysis; known for its memory-error detection tool (memcheck), but also has:
 - thread-error detection
 - cache and branch-prediction profiler
 - heap profiler
- It works by running your code on a synthetic CPU provided by the Valgrind core
 - valgrind adds its own instrumentation to the execution code
 - amount of instrumentation can vary
- Valgrind's memory error detector used in C/C++ programs for:
 - accessing memory you shouldn't
 - using undefined variables
 - memory leaks
 - incorrect freeing of memory
 - Some support for fortran (particularly using newer gfortran)
 - Program likely to run 20-30X slower
 - No relinking required

Valgrind is available on Stampede

Memory Debugging Examples

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    float *array;
    int i;

    array=(float*)calloc(20000,sizeof(float));

    for(i=1;i<=20000;i++)
        array[i] = sqrt(i);

    printf("Program complete\n");
    return;
}
```



dmalloc Example

```
> gcc -g hello_alloc.c -lm
> ./a.out
Program complete
```

Now retry with dmalloc:

```
> gcc hello_alloc.c -lm -lmalloc
> malloc -l logfile -i 100 high
```

debug-malloc library: dumping program, fatal error

Error: failed OVER picket-fence magic-number check (err 27)

Aborted

Memory Debugging Examples

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    float *array;
    int i;

    array=(float*)calloc(20000,sizeof(float));

    for(i=1;i<=20000;i++)
        array[i] = sqrt(i);

    printf("Program complete\n");
    return;
}
```

C

Electric Fence Example

```
> gcc -g hello_alloc.c -lm -lefence
> ./a.out
Electric Fence 2.2.0
Segmentation fault (core dumped)
```

Ok, that was great, now what do we do?

```
> gdb ./a.out core.7184
GNU gdb Red Hat Linux
(6.3.0.0-1.134.fc5rh)
Core was generated by `./a.out'.
Program terminated with signal 11,
Segmentation fault.
```

```
#0  0x08048527 in main () at
    hello_alloc.c:13
13          array[i] = sqrt(i);
(gdb) print i
$1 = 20000
```

Memory Debugging Examples

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    float *array;
    int i;

    array=(float*)calloc(20000,sizeof(float));

    for(i=1;i<=20000;i++)
        array[i] = sqrt(i);

    printf("Program complete\n");
    return;
}
```

C

Valgrind Example

```
> gcc -g hello_alloc.c -lm
> ./a.out
Program complete

> module load valgrind
> valgrind ./a.out

==13043== Memcheck,
==13043== Command: ./a.out
==13043==

==13113== Invalid write of size 4
==13113==   at 0x4005EE: main
(hello_alloc.c:13)
==13113==   Address 0x4c358c0 is 0 bytes
           after a block of size 80,000 alloc'd
==13113==   at 0x4A05140: calloc
(vg_replace_malloc.c:418)
==13113==   by 0x40058E: main
(hello_alloc.c:10)
==13113==

Program complete
```

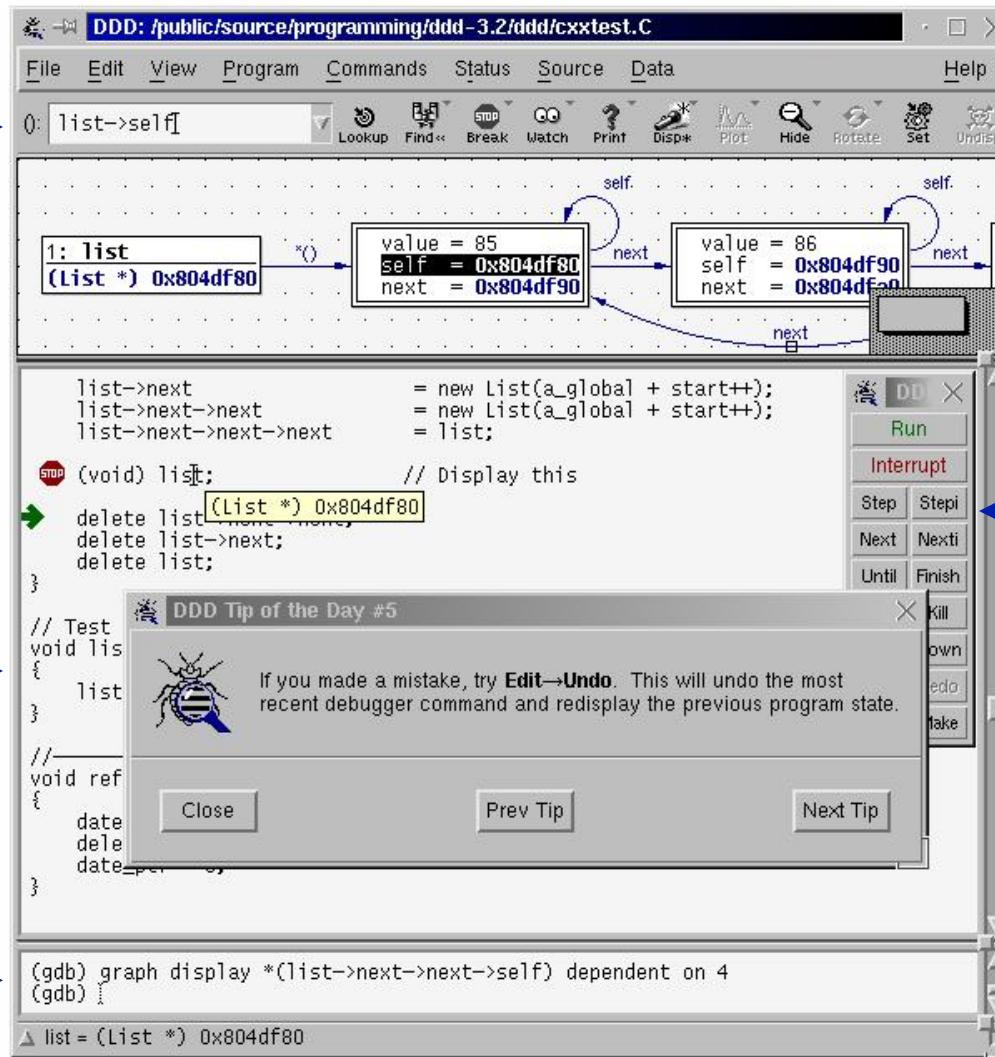
Graphical Debuggers

- In addition to the very capable command-line debuggers available (*gdb, dbx*) - there are graphical versions
- Once you've mastered the basic concepts with a command-line debugger, a graphical version can make your life even easier
- Common graphical debuggers:
 - DDD (*free!*)
 - TotalView (Commercial, supports threading and MPI)
 - DDT (Commercial, supports threading and MPI)
- These all require X-windows support under Unix to display the graphical interface

***DDD is really just a front-end
for a command-line debugger***

Graphical Debugging: DDD

Argument
Field



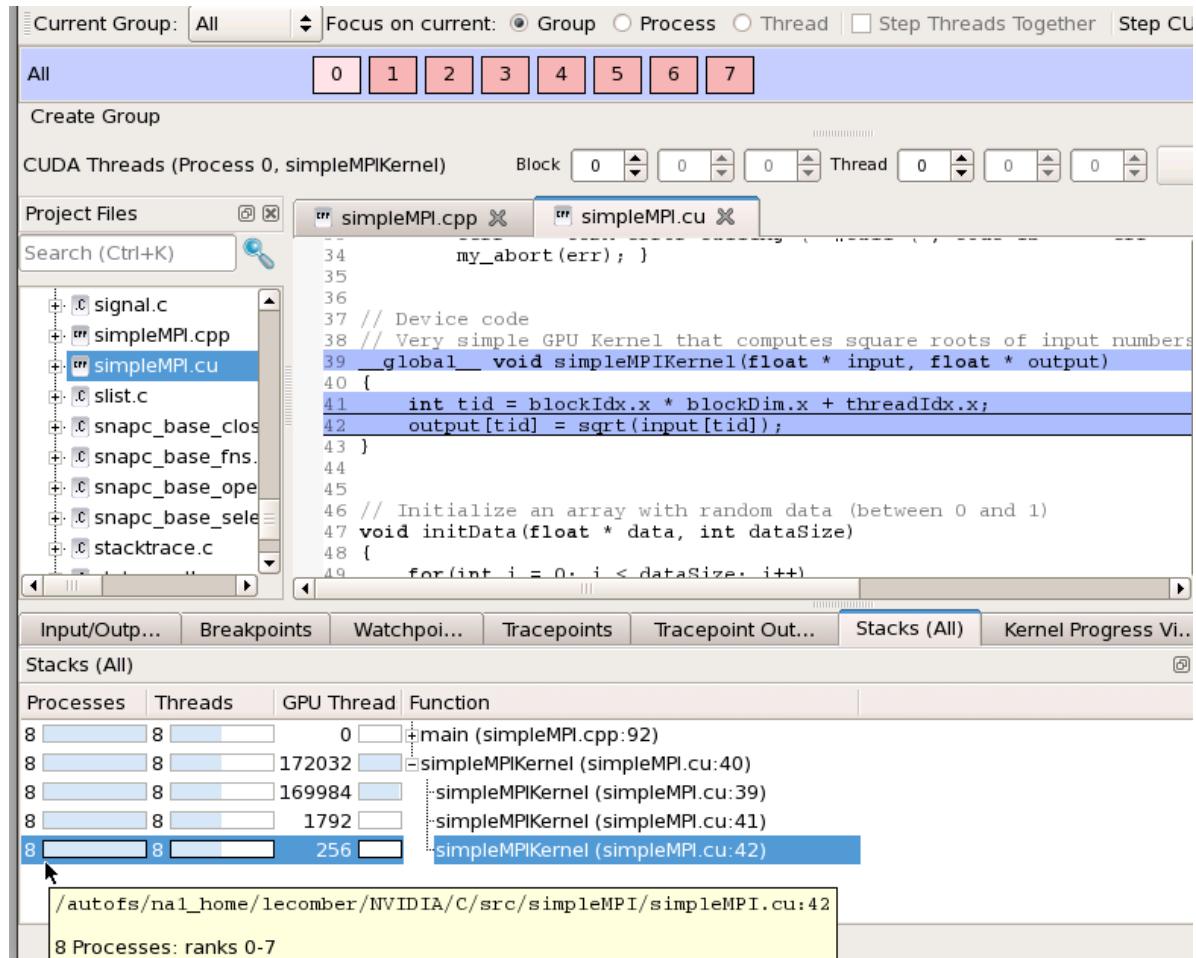
Debugger
Console

Data
Window

Command
Tool

Graphical Debugging: DDT

- DDT is a commercial debugger for serial and parallel processing
- Available on Stampede and Lonestar
- Extremely capable
 - allows you to interrogate individual MPI tasks (with over 100K tasks)
 - includes some basic memory debugging tools (eg. overruns)
- See
<http://www.tacc.utexas.edu/user-services/user-guides/ddt-manual> for usage on TACC systems



The screenshot shows the DDT graphical debugger interface. At the top, there are buttons for 'Current Group' (All), 'Focus on current' (Group selected), 'Step Threads Together', and 'Step CU'. Below this is a row of numbered buttons (0-7) and controls for 'Block' and 'Thread' counts. The main window has tabs for 'Project Files' (simpleMPI.cpp, simpleMPI.cu selected), 'Search (Ctrl+K)', and a code editor showing CUDA device code for MPI operations. Below the code editor are tabs for 'Input/Output...', 'Breakpoints', 'Watchpoi...', 'Tracepoints', 'Tracepoint Out...', 'Stacks (All)' (selected), and 'Kernel Progress Vi...'. The 'Stacks (All)' tab displays a table with columns 'Processes', 'Threads', 'GPU Thread', and 'Function'. It lists four processes, each with 8 threads, running at GPU thread 0, 172032, 169984, and 1792 respectively, all executing the 'simpleMPIKernel' function at line 42 of simpleMPI.cu. A tooltip at the bottom indicates '8 Processes: ranks 0-7'.

References/Acknowledgements

- Arctic Region Supercomputing Center, Core Skills for Computational Science: <http://people.arsc.edu/~cskills/>
- Debugging with GDB:
http://sources.redhat.com/gdb/current/onlinedocs/gdb.html#SEC_Top
- Debug malloc library: <http://dmalloc.com/>
- Electric Fence <http://directory.fsf.org/ElectricFence.html>
- Valgrind: <http://valgrind.org>
- Data Display Debugger (DDD): <http://www.gnu.org/manual/ddd/>