# CS395T: Introduction to Scientific and Technical Computing
# Debugging

*Instructors:*

Dr. Karl W. Schulz, TACC

Dr. Victor Eijkhout, TACC

# 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 (*eg. negative density values in a flow code are probably bad*)
  - Stability is a concern (*eg. 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 (CVS, Subversion, 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
- Useful error checks indicate you know what you are doing - at an **absolute minimum**, please check your memory allocations:

C

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

Fortran

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

# 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 validation phase, make use of available compiler options to debugging options to trap:
  - Intel Fortran Examples:

    **-check all** - enable runtime checks for out-of-bounds array subscripts, unitialized 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
  ```
  Use the -Wall function to help find errors at compile time
  ```

# 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 programmers should take advantage of the `assert` macro to ensure values fall within appropriate ranges

```c
#include <stdio.h>
#include <assert.h>

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

  /* Assert that n <= 100 */

  assert ( n <= 100);

  return 0;
}
```

C

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

# 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?

**Fortran**

```
 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
```

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

# Bug Identification

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

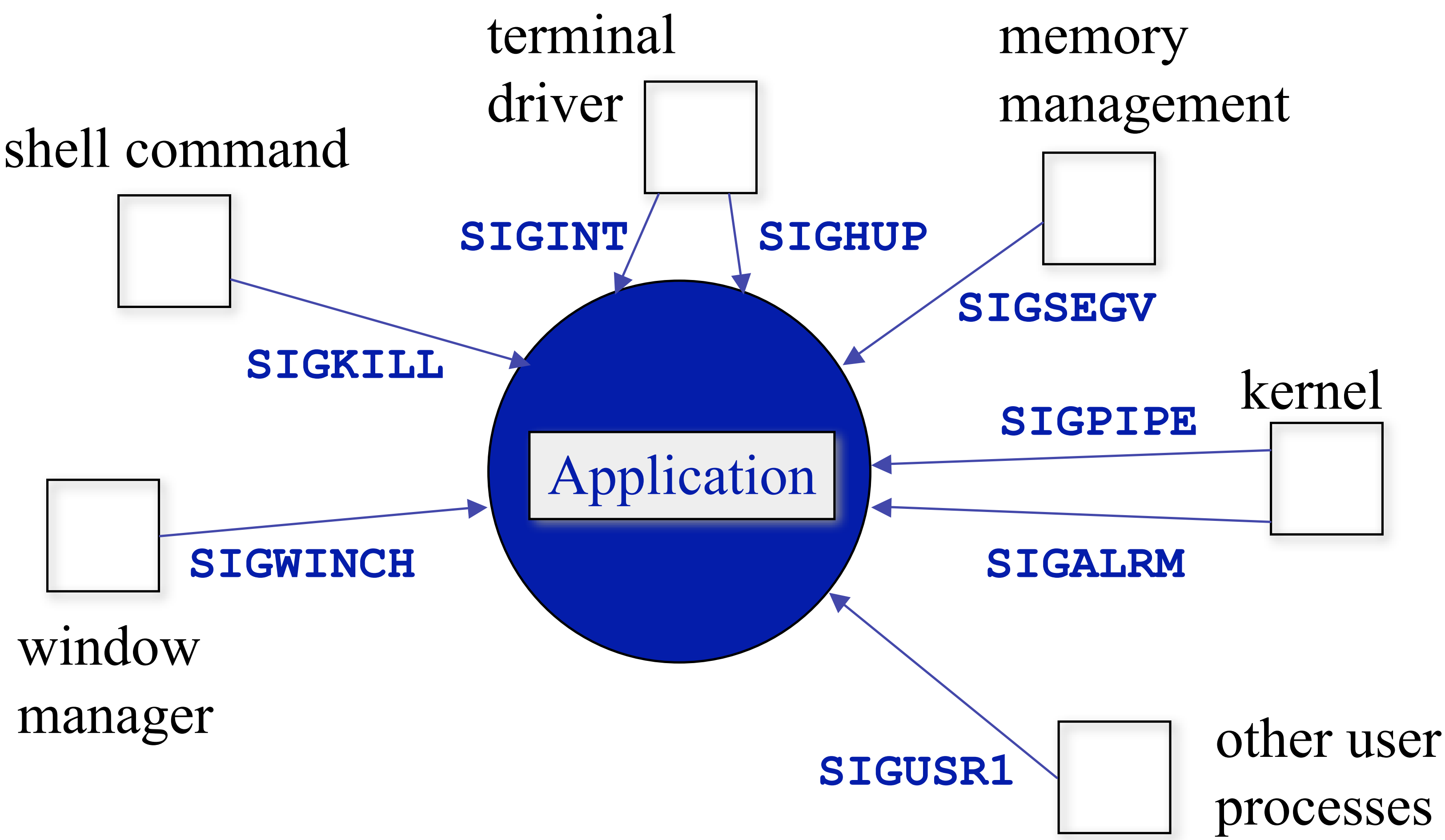
- Asynchronous means that the event can occur at any time
  - may be unrelated to the execution of the process
  - e.g. user types `ctrl-C`, or the operating system detects an error and sends a signal to your application

# Common Signal Types

| Name | Description | Default Action |
|---|---|---|
| `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, in TCSH:

    ```
    > unlimit coredumpsize
    ```

  - 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 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
  - AIX: dbx
  - SUN: dbx
  - TACC machines: DDT graphical parallel debugger

- 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

# 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***

TACC

# 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/karl/cs395t/hello
inside main
inside foo

Program exited with code 0347.

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

(gdb) run
Starting program: /home/karl/cs395t/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        }
```

TACC

# Example GDB Session (continued)

```
(gdb) cont
Continuing.
inside foo

Program exited with code 0347.
(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/karl/cs395t/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
```

TACC

# gdb and Emacs

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

```
all: hello

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

TACC

# 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 in gdb

Null pointers and illegal addresses are easy to recognize

```
#include <stdio.h>
int main()
{
  double *a;
  a = 2;
  a[1] = 5.1;
  return 0;
}
```

```
(gdb) run
Starting program:a.out
Reading symbols for shared libraries . done

Program received signal EXC_BAD_ACCESS, Could not access memory
Reason: KERN_PROTECTION_FAILURE at address: 0x0000000a
0x00001f98 in main () at segfault.c:6
(gdb) print a
$1 = (double *) 0x2
(gdb)
```

TACC

# References/Acknowledgements

- 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
- Data Display Debugger (DDD): http://www.gnu.org/manual/ddd/

# 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