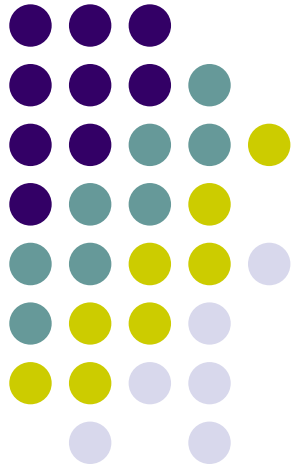
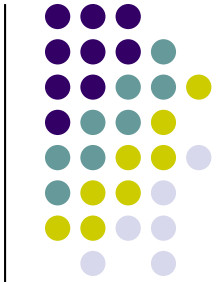
ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Elements of Program Debugging



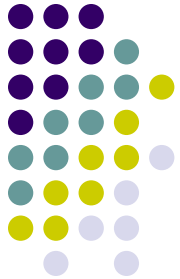


Debugging on Euler

[with gdb]

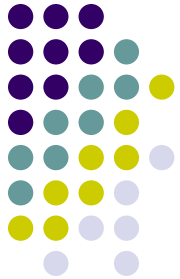
Slides on gdb include material from Purdue University

gdb: Intro



- `gdb`: a utility that helps you debug your program
- Learning `gdb` is a good investment of your time
  - Yields significant boost of productivity
- A debugger will make a good programmer a better programmer
- In ME759, you should go beyond sprinkling "`printf`" here and there to try to debug your code
  - Avoid: Compile-link, compile-link, compile-link, compile-link, compile-link, compile-link, compile-link, compile-link,…

Compiling a Program for gdb

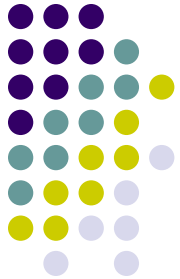


- You need to compile with the “-g” option to be able to debug a program with gdb.

- The “-g” option adds debugging information to your program

```
gcc –g –o hello hello.c
```

Running a Program with gdb



- To run a program called **`progName`** with **`gdb`** type
  `>> gdb progName`

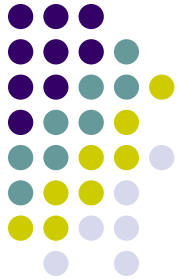
- Then set a breakpoint in the main function
  `(gdb) break main`

- A breakpoint is a marker in your program that will make the program stop and return control back to gdb

- Now run your program
  (**`gdb) run`**

- If your program has arguments, you can pass them after run.

Stepping Through your Program



- Your program will start running and when it reaches “main()” it will stop:
  `(gdb)`

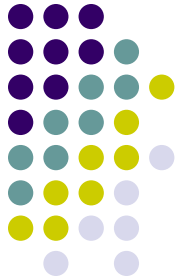
- You can use the following commands to run your program step by step:
  `(gdb) step`

  It will run the next line of code and stop. If it is a function call, it will enter into it

  `(gdb) next`

  It will run the next line of code and stop. If it is a function call, it will not enter the function and it will go through it.

Printing the Value of a Variable



- The command
  (gdb) print varName
  … prints the value of a variable

E.g.

```
(gdb) print i
$1 = 5
(gdb) print s1
$1 = 0x10740 "Hello"
(gdb) print stack[2]
$1 = 56
(gdb) print stack
$2 = {0, 0, 56, 0, 0, 0, 0, 0, 0, 0}
(gdb)
```

Setting Breakpoints

- A breakpoint is a location in a program where the execution stops in **gdb** and control is passed back to you

- You can set breakpoints in a program in several ways:

(gdb) break functionName

Set a breakpoint in a function. E.g.

(gdb) break main

(gdb) break lineNumber

Set a break point at a line in the current file. E.g.

(gdb) break 66

It will set a break point in line 66 of the current file.

(gdb) break fileName:lineNumber

It will set a break point at a line in a specific file. E.g.

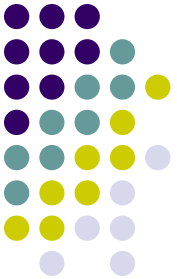
(gdb) break hello.c:78

(gdb) break fileName:functionName

It will set a break point in a function in a specific file. E.g.

(gdb) break subdivision.c:paritalSum

Watching a Variable



- Many times you want to keep an eye on a variable that for some reason assumes a value that is not in line with expectations
- To that end, you can “watch” a variable and have the code break as soon as the variable is read or changed
- You can watch a variable in several ways:

  `(gdb) watch varName`

  Program breaks whenever **`varName`** gets written by the program

  `(gdb) rwatch varName`

  Program breaks whenever **`varName`** gets read by the program

  `(gdb) awatch varName`

  Program breaks whenever **`varName`** gets read/written by the program

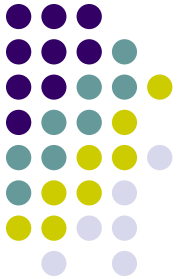
- Get a list of all watchpoints, breakpoints, and catchpoints in your program:
  `(gdb) info watchpoints`

Example:

[watching a variable]

```
#include <iostream>

int main(){
  int arr[2]={266,5};

  int * p;
  short s;

  p = (int*) malloc(sizeof(int)*3);

  p[2] = arr[1] * 3;

  s = (short)( *(p+2) );

  free( p );

  p=NULL;

  p[0] = 5;
  return 0;
}
```

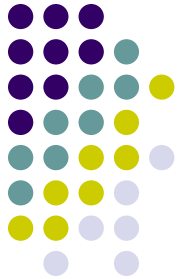
Example: Watching Variable “s”



- Below is a copy-and-paste from gdb, for our short program

```
(gdb) awatch s
Hardware access (read/write) watchpoint 2: s
(gdb) continue
Continuing.
Hardware access (read/write) watchpoint 2: s

Old value = 0
New value = 15
main () at pointerArithm.cpp:15
15              free( p );
```

Regaining the Control



- When you type

  `(gdb) run`

  the program will start running and it will stop at a breakpoint

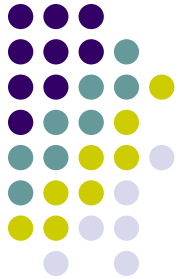
- If the program is running without stopping, you can regain control again typing ctrl-c

- When you type

  `(gdb) continue`

  the program will run until it hits the next breakpoint, or exits

Where Are You?



- The command

  (gdb)where

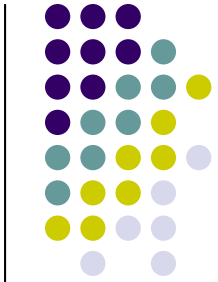
  Will print the current function being executed and the chain of functions that are calling that fuction.

  This is also called the backtrace.

  Example:

```
(gdb) where
#0  main () at test_mystring.c:22
(gdb)
```

Seeing Code Around You…



- The command **`list`** shows you code around the location where the execution is “break-ed”

  `(gdb)list`

  It will print, by default, 10 lines of code.

  There are several flavors:

  `(gdb)list lineNumber`

  …prints code around a certain line number

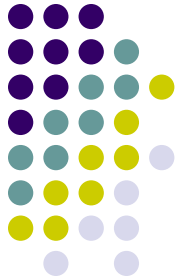
  `(gdb)list functionName`

  …prints lines of code around the beginning of a function

  `(gdb)set listsize someNumber`

  …controls the number of lines showed with **`list`** command

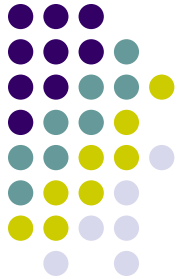
Exiting gdb



- The command “quit” exits **gdb**.

```
(gdb) quit
The program is running.  Exit anyway?
  (y or n) y
```

Debugging a Crashed Program



- Also called “postmortem debugging”
- When a program segfaults, it writes a **core file**.

```
bash-4.1$ ./hello
Segmentation Fault (core dumped)
bash-4.1$
```

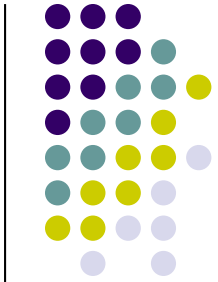
- The core is a file that contains a snapshot of the state of the program at the time of the crash
  - Information includes what function the program was running upon crash

Example:
[Code crashing]

```
#include <iostream>

int main(){
  int arr[2]={266,5};

  int * p;
  short s;

  p = (int*) malloc(sizeof(int)*3);

  p[2] = arr[1] * 3;

  s = (short)( *(p+2) );

  free( p );

  p=NULL;

  p[0] = 5;
  return 0;
}
```

This is why it's crashing…

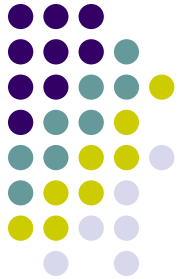
Running gdb on a Segmentation fault



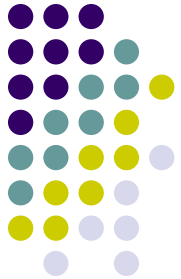
- Here's what gdb says when running the code…

```
[user@euler CodeBits]$ gdb badPointerArithm.out
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-50.el6)
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 "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/user/ME964/Spring2012/CodeBits/badPointerArithm.out...done.
(gdb) run
Starting program: /home/user/ME964/Spring2012/CodeBits/badPointerArithm.out
warning: the debug information found in "/usr/lib/debug//lib64/libc-2.12.so.debug" does not match
"/lib64/libc.so.6" (CRC mismatch).

warning: the debug information found in "/usr/lib/debug/lib64/libc-2.12.so.debug" does not match
"/lib64/libc.so.6" (CRC mismatch).


Program received signal SIGSEGV, Segmentation fault.
0x0000000000400641 in main () at pointerArithm.cpp:19
19                  p[0] = 5;
(gdb)
```

Debugging – Departing Thoughts

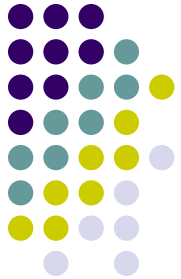


- Debug like a pro (graduate from the use of **`printf`**…)
- **dbg** saves you time
- If a GUI is helpful, use "**ddd**" on Euler – under the hood it uses **gdb**
- Under Windows, Visual Studio has an excellent debugger



MOVING BEYOND GDB

What Else is Out There?

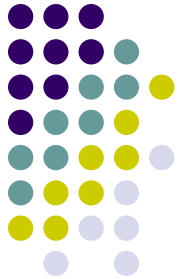


- Debuggers like gdb extremely helpful in tracking down runtime errors
- However, using a debugger like gdb
  - Is a very manual process
  - Requires the user to remember lots of commands to use it effectively

- New tools provided by the `Clang` developer group simplify debugging and provide features beyond those of gdb
  - These are the “sanitizers”, there’re several of them

- Four sanitizers you may find useful
  - Address: for memory corruption
  - Thread: for threaded code (does not work with CUDA)
  - Memory: for uninitialized reads
  - Undefined behavior: for C++ undefined behavior (UB) detection

An Example



```c
#include <stdio.h>

int sum(int *x, int len) {
    int total = 0;
    for (int i = 0; i<len; i++) {
        total += x[i];
    }
    return total;
}

int main() {
    int x[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    printf("sum = %d\n", sum(x, 9));
}
```

Should be 8, not 9

Does the code compile?
Yes (gcc -g test.c -o test)

Does this code run?
Yes

Does it produce correct results?
No

Does it give you an error of any kind?
No

What does gdb say?
> gdb ./test
> (gdb) run
> Starting program: test
> sum = -7932
> [Inferior 1 (process 20543) exited normally]

Now what?

[Tim Haines]→

An Example

```c
#include <stdio.h>

int sum(int *x, int len) {
    int total = 0;
    for (int i = 0; i<len; i++) {
        total += x[i];
    }
    return total;
}

int main() {
    int x[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    printf("sum = %d\n", sum(x, 9));
}
```

Should be 8, not 9

- If we had no other tools, we might have to resort to inserting `print` statements everywhere to try and chase down the bug. That's time consuming & ineffective. Your own `print` statement might change the results…
- Rather, use sanitizers. Recompile with some additional flags:

```
>> gcc -g -O0 -fno-omit-frame-pointer --sanitize=address -o test test.c
```

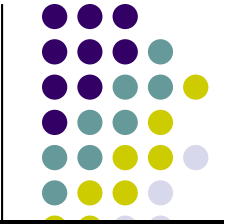
- Next, simply run the executable

```
>> ./test
```

- This produces lots of output, see next slide

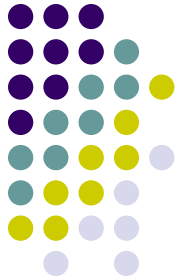
[Tim Haines]→

An Example



```
=================================================================
==20626==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffc07666b90 at pc 0x0000004009e2 bp 0x7ffc07666b10 sp 0x7ffc07666b00
READ of size 4 at 0x7ffc07666b90 thread T0
    #0 0x4009e1 in sum /home/tim/workspace/CPPTest/test.c:6
    #1 0x400c85 in main /home/tim/workspace/CPPTest/test.c:13
    #2 0x7f138e20582f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
    #3 0x4008c8 in _start (/home/tim/workspace/CPPTest/test+0x4008c8)

Address 0x7ffc07666b90 is located in stack of thread T0 at offset 64 in frame
    #0 0x400a07 in main /home/tim/workspace/CPPTest/test.c:11

  This frame has 1 object(s):
    [32, 64) 'x' <== Memory access at offset 64 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/tim/workspace/CPPTest/test.c:6 in sum
Shadow bytes around the buggy address:
  0x100000ec4d20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000ec4d30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000ec4d40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000ec4d50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000ec4d60: 00 00 00 00 00 00 00 00 00 00 f1 f1 f1 f1 00 00
=>0x100000ec4d70: 00 00[f3]f3 f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000ec4d80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000ec4d90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000ec4da0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000ec4db0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000ec4dc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==20626==ABORTING
```

An Example



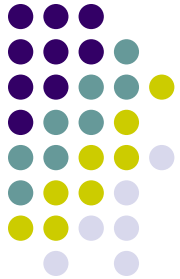
- There are a couple of important pieces
  - The call stack that shows the line in the source file where the error occurred

```
READ of size 4 at 0x7ffc07666b90 thread T0
    #0 0x4009e1 in sum /home/tim/workspace/CPPTest/test.c:6
    #1 0x400c85 in main /home/tim/workspace/CPPTest/test.c:13
    #2 0x7f138e20582f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
    #3 0x4008c8 in _start (/home/tim/workspace/CPPTest/test+0x4008c8)
```

  - The variable that caused the error

```
 This frame has 1 object(s):
   [32, 64) 'x' <== Memory access at offset 64 overflows this variable
```

Other Sanitizers



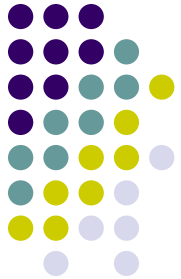
- The sanitizers' official names are
  - `AddressSanitizer`
  - `MemorySanitizer`
  - `ThreadSanitizer`
  - `UndefinedBehaviorSanitizer`

- Documentation for these and other sanitizers
  - https://clang.llvm.org/docs/index.html

[Tim Haines]→

Departing Thoughts

Improving your productivity as a programmer



- Draw on `gdb` (or some other debugger) but don't limit yourself to that
- Turn on flags that make the compiler be picky and whiny
- Use `clint` – good semantic checker (at compile time)
- Use `valgraind` – dynamically monitors execution of your program (at run time)
- Keep your code simple
- Comment your code
- Use revision control (`git`, `svn`, etc.)