Introduction to Debugging Tools

Don Bahls
User Consultant
dmbahls@alaska.edu
Overview

- Debugging Thoughts
- strace
- valgrind
- gdb
- totalview

Bugs Suck!
Coding / Debugging Thoughts

- **Defensive programming** is always a good idea.
- Write your code to **handle the unexpected** (not always an easy thing to do).
- When errors occur, provide a **useful error message**.
- It can be useful to have the capability to print out debugging messages (enabled at runtime or compile time).
- When possible, it’s best to **avoid changing your code** to attempt and debug it (e.g. print statements).
The Truth

- The code you need to debug may not be your own.
- It’s probably not worth your time to completely rewrite a piece of code.
- Adding print statements can make a bug go away!
Tools

- **strace** – tracks system calls.
- **valgrind** – memory debugger and more
- **gdb** – source code debugger for C/C++/Fortran
- **totalview** - gui based debugger with support for C/C++/Fortran and GPUs. Also supports parallel applications.
These tools all let you observe a compiled application without changing the underlying source code.

**NOTE:** Recompiling may be beneficial!
strace command

- The **strace** command traces systems calls
  - Does not require code to be recompiled
  - Can be very cryptic if you aren’t familiar with low level OS calls.
- Useful for: basic network traffic tracing, seeing which files a program opens, seeing forked processes, attaching to running programs.
Basic Usage of strace

```
pacman6 % strace ./open_file
execve("./open_file", ["./open_file"], [/* 45 vars */]) = 0
brk(0) = 0x67d8000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2ab03553a000
...  
brk(0) = 0x67d8000
brk(0x67f9000) = 0x67f9000
open("/u1/uaf/bahls/myfile.not_here.out", O_RDONLY) = -1 ENOENT
exit_group(0) = ?
```
More strace options

- Trace subprocesses
  ```
  % strace -f ./a.out
  ```
- Save strace output to a file (normally goes to stderr)
  ```
  % strace -o strace.out ./a.out
  ```
- Limit tracing to particular group of system calls
  ```
  % strace -e trace=network ./a.out
  ```
A few more strace options

- Enable verbose output (avoid using unless necessary)
  
  \% strace -v ./a.out

- Limit tracing to particular group of system calls
  
  \% strace -e trace=open,execve ./a.out

- Include timing information in output
  
  \% strace -tt ./a.out

- Attach to a running process (doesn’t require an executable)
  
  \% strace -p PID
A Word of Warning

- Be careful when running strace on password authenticated processes!
- Depending on the options used, your password could be shown or worse yet saved in clear text!
Valgrind is suite of tools for debugging and profiling programs. Provides a “simulated” CPU to run an application.

This talk focuses on the MEMCHECK functionality of Valgrind.

**NOTE**: some options can *significantly slow* code execution!
Code Preparation

- With valgrind, gdb and totalview recompiling with debugging information (-g option) can result in more useful information from the tool.

- e.g.

  ```
  % gcc -g myfile.c -o myfile
  ```
Basic Usage of Valgrind (MEMCHECK)

```sh
pacman1 % valgrind ./mem_leak
==212003== Memcheck, a memory error detector
...
==212003== LEAK SUMMARY:
==212003==   definitely lost: 400 bytes in 1 blocks
==212003==   indirectly lost: 0 bytes in 0 blocks
==212003==   possibly lost: 0 bytes in 0 blocks
==212003==   still reachable: 0 bytes in 0 blocks
==212003==   suppressed: 0 bytes in 0 blocks
==212003== Rerun with --leak-check=full to see details of leaked memory
==212003==
==212003== For counts of detected and suppressed errors, rerun with: -v
==212003== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```
More valgrind options

- Enable leak checking (shows line numbers)
  ```
  % valgrind --leak-check=full ./a.out
  ```

- Track leak origins
  ```
  % valgrind --leak-check=full \ 
  --track-origins=yes ./a.out
  ```

- Attach to a debugger on certain errors
  ```
  % valgrind --db-attach=yes \ 
  --leak-check=full \ 
  --track-origins=yes ./a.out
  ```
pacman1 % valgrind --leak-check=full --track-origins=yes ./mem_leak
==355180== Memcheck, a memory error detector
...
==355180== HEAP SUMMARY:
==355180==     in use at exit: 400 bytes in 1 blocks
==355180== total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==355180== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==355180==     at 0x4C2210C: malloc (vg_replace_malloc.c:195)
==355180==     by 0x400539: leaky (mem_leak.c:14)
==355180==     by 0x4005A6: main (mem_leak.c:35)
==355180==
==355180== LEAK SUMMARY:
==355180==     definitely lost: 400 bytes in 1 blocks
==355180==     indirectly lost: 0 bytes in 0 blocks
==355180==     possibly lost: 0 bytes in 0 blocks
==355180==     still reachable: 0 bytes in 0 blocks
==355180==     suppressed: 0 bytes in 0 blocks
==355180==
==355180== For counts of detected and suppressed errors, rerun with: -v
==355180== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
gdb command

- A command line debugger.
  - Lets you run an application under the control of the debugger
  - Inspect core dumps
  - Attach to a running process to see what it is doing
  - Works on serial and threaded programs. MPI programs maybe?
Basic Usage of gdb

```
pacman1 % gdb ./seg_fault
...  
(gdb) run
...  
(gdb) backtrace
#0  0x00002aaa00000000 in ?? ()
#1  0x00000000000400370 in __libc_start_main@plt ()
#2  0x00007fffffffe688 in ?? ()
#3  0x0000000100000000 in ?? ()
#4  0x0000000000040047d in zero_array (a=Cannot access memory at address 0xffffffffffffffe8 ) at seg_fault.c:11
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```
Description

➢ The “run” command runs the binary
  ➢ Programs automatically quit running when a signal is encountered (e.g. SIGSEGV)
  ➢ The “backtrace” command shows a stack trace.
Common Commands

- **list** – show source code
- **step** – next instruction goes into subroutines.
- **next** – next instruction in the current subroutine.
- **continue** – continue running code until the code ends, or reaches a watch or break point.
- **finish** – complete the current subroutine.
- **print foo** – print the current value of `foo`.
Breakpoints

- Tell the debugger to stop at a function or file or line number.

  - `break main`  # stop at main function
  - `break 10`    # stop at line 10 in file
  - `break foo.c:100`  # stop at line 100 in file foo.c
  - `info break`  # list current break points
  - `clear foo.c:100`  # clear a break point
(gdb) break main
Breakpoint 1 at 0x400488: file seg_fault.c, line 17.
(gdb) break 7
Breakpoint 2 at 0x400453: file seg_fault.c, line 7.
(gdb) info break
Num Type Disp Enb Address What
1 breakpoint keep y 0x0000000000000400488 in main at seg_fault.c:17
2 breakpoint keep y 0x0000000000000400453 in zero_array at seg_fault.c:7
(gdb) run
...
Breakpoint 1, main () at seg_fault.c:17
17 zero_array( array, 1027 );
(gdb)
Watch Points

- Report the value of a variable when it is read from, written to, or both.

  - `watch X`  # print X on write (change).
  - `rwatch X`  # print X on read
  - `awatch X`  # print X on read or write
  - `info watch`  # list current watch points
  - `clear foo.c:100`  # clear a break point
(gdb) list
...
6      s=1
7    ret = add_them( 1, s )
(gdb) break 6
Breakpoint 1 at 0x400797: file by_reference.f90, line 6.
(gdb) run
...
Breakpoint 1, MAIN__ () at by_reference.f90:6
6      s=1
(gdb) watch s
Hardware watchpoint 2: s
(gdb) continue
Old value = -1431527424
New value = 2
add_them (a=@0x4009f8, b=@0x7fffffffde768) at
by_reference.f90:21
21  add_them = b
More stack command

- **backtrace full** – shows local variables for each level of the stack trace.
- **up** – go up the call stack one level
- **down** – go down the call stack one level
Text User Interface to gdb

➢ gdb also has a text user interface
➢ This shows break points and current instruction pointer.
➢ Start with “gdb –tui ./a.out”
➢ May need to start running code for the code to show up.
seg_fault.c
{
    int array[1024];
    zero_array( array, 1027 );
}

exec No process In:
Copyright (C) 2009 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 /import/home/ul/uaf/bahls/debugging_tools/seg_FAULT...
(gdb) focus cmd
Focus set to CMD window.
(gdb)
break point

switch control to source window

switch control back to command window
TUI Navigation

- **focus src** – Move focus to source code window
- **focus cmd** – Move focus to command window.
- **info win** – shows window size and current focus.
- **layout src** – show source window when started from standard gdb.
- **layout regs** – show registers, source and command windows
TUI Pictures

- REGS
- SRC
- CMD

layout regs  layout src

focus cmd
Other activities

- View a core file
  - `% gdb ./a.out core`
  - Or
    - `(gdb)core corefile.name`
- Attach to a running process
  - `% gdb ./a.out -p PID`
  - Or
    - `(gdb)attach PID`
Totalview

- Available on pacman and fish.
- GUI based debugger with MPI, OpenMP, pthread, CUDA support.
- Similar basic options as gdb. Some nice features such as array visualization.
Basics

```
pacman6 % qsub -q debug -lnodes=1:ppn=16 -X -I
qsub: waiting for job 562161.scyld to start

n44 % module load PrgEnv-pgi totalview
n44 % cd $PBS_O_WORKDIR

n44 % totalview mpirun -a -np 4 ./ring
```
Thread is running

Thread must be stopped for frame display.

Process mpirun is a parallel job. Do you want to stop the job now?

Yes  No
move up and down the stack

click to set break point

local vars

registers
Set break points or watch points
Parallel Application Control
Other Totalview Features

- Graph arrays
- Playback engine (haven’t tried it)
- Memory debugger (haven’t tried it)
- CUDA support
Additional Information

- Watch [www.arsc.edu](http://www.arsc.edu) for upcoming training and additional information about ARSC
- A simple intro on gdb with good TUI information:
  - [http://beej.us/guide/bgdb/](http://beej.us/guide/bgdb/)
Exercises

➢ If you would like some practice on things covered today, see:
  $SAMPLES_HOME/training/debugTools

➢ This covers strace basics, valgrind usage as well as some examples usable with gdb or totalview.
Questions and Feedback

➢ If you have additional questions, feel free to contact us:
  ARSC Help Desk
  Phone: (907) 450-8602
  Email: consult@arsc.edu
  Web: http://www.arsc.edu/support

➢ We welcome your feedback on this class!