COL7001: Systems Concepts

Kolin Paul

Debugging

- Testing
  - What should I do to try to break my program?
- Debugging
  - What should I do to try to fix my program?
- Why?
  - Debugging large programs can be difficult
- GDB
  - A great debugging tool
- The GDB Debugger
  - Part of the GNU development environment
- What is a debugger?

An execution monitor?

- What would you like to "see inside" and "do to" a running program?
- Why might all that be helpful?
- What are reasonable ways to debug a program?
- A "debugger" is a tool that lets you stop running programs, inspect (sometimes set) values, etc.
  - A "MRI" for observing executing code

Understand Error Messages

- Debugging at build-time is easier than debugging at run-time, if and only if you…
  - Understand the error messages!!!

Misspelled #include file

```
#include <stdioo.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0.
{
   printf("hello, world\n");
   return 0;
}
```

Missing */

```
$ gcc217 hello.c -o hello
hello.c:1:20: stdioo.h: No such file or directory
hello.c:3:1: unterminated comment
hello.c:2: error: syntax error at end of input
```

```
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0. */
{
   printf("hello, world\n")
   retun 0;
}
```

Misspelled keyword

```
$ gcc217 hello.c -o hello
hello.c: In function `main':
hello.c:7: error: `retun' undeclared (first use in this function)
hello.c:7: error: (Each undeclared identifier is reported only once
hello.c:7: error: for each function it appears in.)
hello.c:7: error: syntax error before numeric constant
```

```
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0. */
{
   prinf("hello, world\n")
   return 0;
}
```

Misspelled function name

Compiler **warning** (not **error**): prinf() is called before declared

Linker error: Cannot find definition of prinf()

```
$ gcc217 hello.c -o hello
hello.c: In function `main':
hello.c:6: warning: implicit declaration of function `prinf'
/tmp/cc43ebjk.o(.text+0x25): In function `main':
: undefined reference to `prinf'
collect2: ld returned 1 exit status
```

gdb (GNU Debugger)

- Debuggers are programs which allow you to execute your program in a controlled manner, so you can look inside your program to find a bug.
- gdb is a reasonably sophisticated text based debugger. It can let you:
  - Start your program, specifying anything that might affect its behavior.
  - Make your program stop on specified conditions.
  - Examine what has happened, when your program has stopped.
  - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.
- SYNOPSIS

  gdb [prog] [core|procID]

GDB

- GDB is a powerful debugger that has the capabilities to
  - Set breakpoints stop at line of code
  - Set watchpoints stop when variable changes
  - Print values
  - Step through execution
  - Backtrace see previous function calls
- GDB has many functionalities check out this link for more features
    - https://sourceware.org/gdb/current/onlinedocs/gdb/

Quick Walkthrough

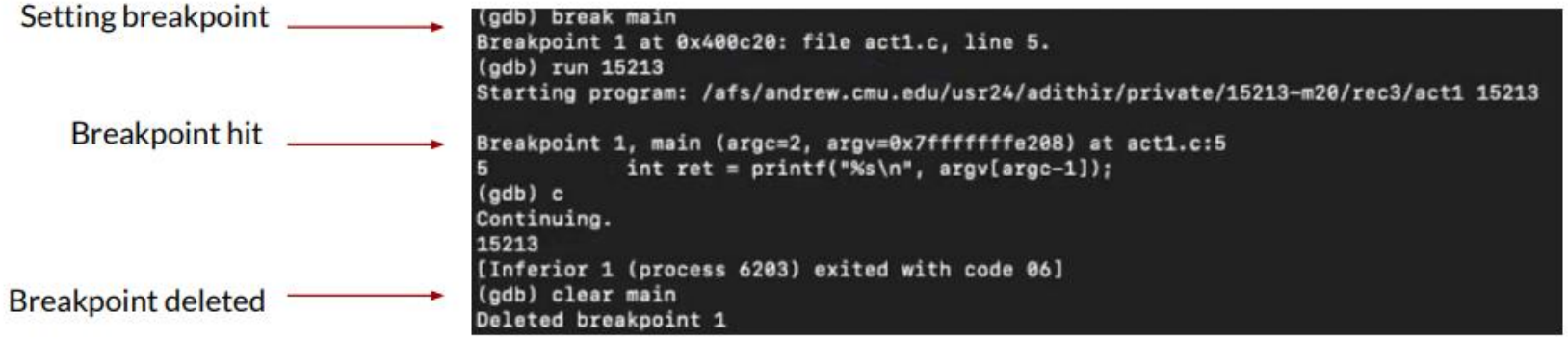
- Compile with –g [so that symbol table is available]
- You can open gdb by typing into the shell:
  - $ gdb
  - (gdb) run 15213 // run program
- Type gdb and then a binary to specify which program to run
  - $ gdb <binary> ($ gdb ./a.out)
  - You can optionally have gdb pass any arguments after the executable file using --args
    - $ gdb --args gcc -O2 -c foo.c
- Quitting GDB:
  - (gdb) quit [expression]
    - (gdb) q
    - or type an end-of-file character (usually Ctrl-d)

Quick Walkthrough

- Controlled Program Execution
  - (gdb) CTRL + c: stops execution
  - (gdb) next (n): run next line of program and does NOT step into functions
  - (gdb) next X (n X): run next X lines of function
  - (gdb) nexti: run next line of assembly code and does NOT step into functions
  - (gdb) step (s): run next line of program AND step into functions
  - (gdb) step X (s X): step through next X lines of function
  - (gdb) stepi: step through next line of assembly code
  - (gdb) continue (c): continue running code until next breakpoint or error
  - (gdb) finish (f):run code until current function is finished

Quick Walkthrough

- Connecting Execution with Code
  - (gdb) disassemble (disas): disassemble source code into assembly code
    - NOT dis: dis == disable breakpoints
  - (gdb) list (l0): list 10 lines of source code from current line
    - (gdb) list X (l X): list 10 lines of source code from line number X
  - (gdb) list fnName (l fnName): list 10 lines of source code from fnName function
- Breakpoints
  - A breakpoint makes your program stop whenever a certain point in the program is reached
    - (gdb)break function_name: breaks once you call a specific function. (break abbreviated b)
    - (gdb)break *0x…: breaks when you execute instruction at a certain address
    - (gdb)info b: displays information about all breakpoints currently set
    - (gdb)disable #: disables breakpoint with ID equal to # ($disa is short form not $disas!!!)
    - (gdb)clear [location]: delete breakpoints according to where they are in your program.



Quick Walkthrough

- Watchpoints
  - A special breakpoint that stops your program when the value of an expression changes
  - The expression may be a value of a variable, or involve values combined by operators
  - Enable, disable, and delete both breakpoints and watchpoints
  - (gdb)delete [watchpoint]:delete individual breakpoints/ watchpoints by specifying breakpoint numbers
    - If no argument is specified, delete all breakpoints , (gdb)d

Examples:

- `(gdb) watch foo:` watch the value of a single variable
- `(gdb) watch *(int *) 0x600850:` watch for a change in a numerically entered address

(output) `Watchpoint 1: *(int *) 6293584`

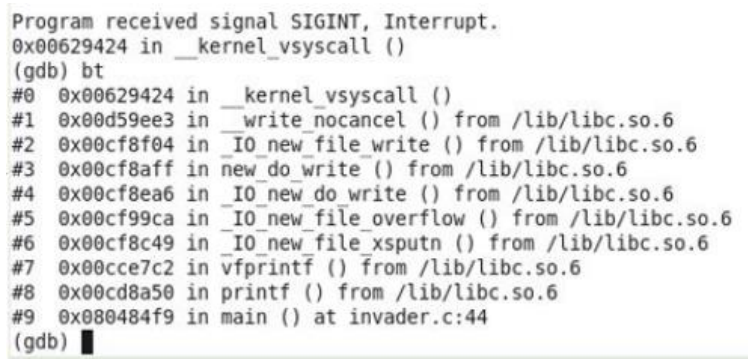
Quick Walkthrough

- Printing Values
  - (gdb) print (p) [any valid expression]
    - Print local variables or memory locations
      - Be sure to cast to the right data type
        - (e.g. p *(long*)ptr)
      - (gdb) print (p) *pntr: prints value of pointer
      - (gdb) print (p) *(struct_t*) tmp:
        - casts tmp to struct_t* and prints internal values
      - (gdb) print (p) expr: prints value of data type
- Inspect Memory
  - (gdb) x/nfu [memory address]:
    - equivalent to (gdb) print *(addr)
  - n: inspect next n units of memory
    - f (format): can be represented as:
    - d (decimal), x (hexadecimal), s (string)
    - u (unit): can be represented as:
      - b (bytes), w (words/ 4 bytes)

Quick Walkthrough

- Backtrace
  - (gdb) backtrace (bt): prints a summary of how program got where it is
    - Print sequence of function calls that led to this point
  - Helpful to use when programs crash
    - (gdb) up N (u N): go up N function calls
    - (gdb) down N (d N): go down N function calls

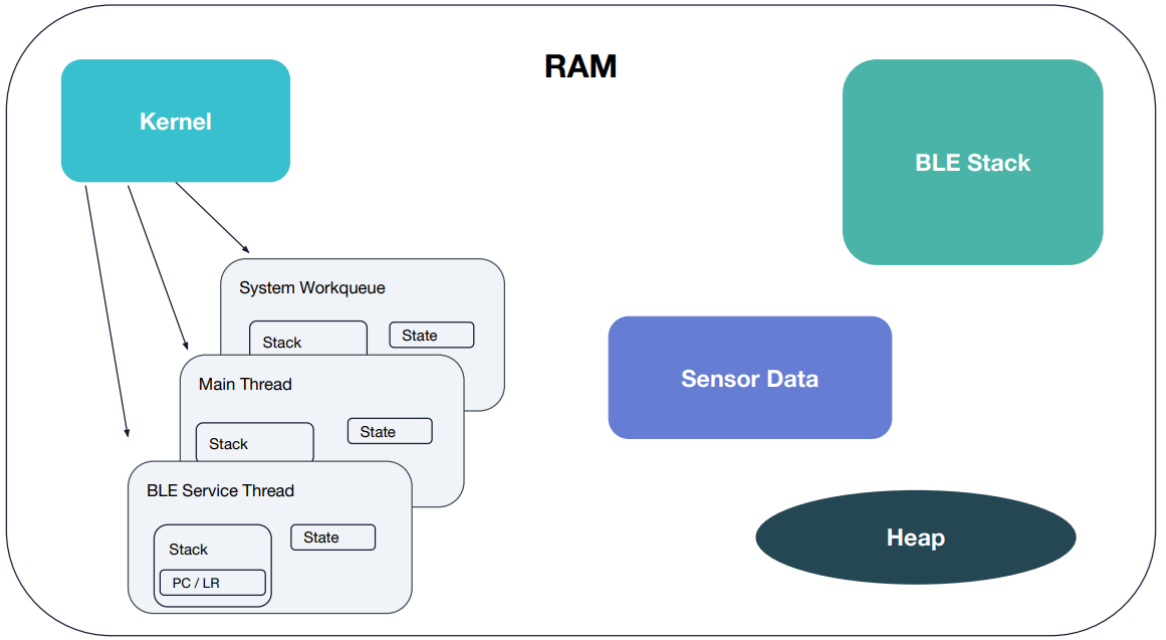
Previous "frames"



- Set Values
  - (gdb) set [variable] expression: change the value associated with a variable, memoryaddress, or expression
  - Evaluates the specified expression. If the expression includes the assignment operator ("="), that operator will be evaluated and the assignment will be done
- Call functions
  - (gdb) call expr: Evaluate the expression expr without displaying void returned values

Debug After Release

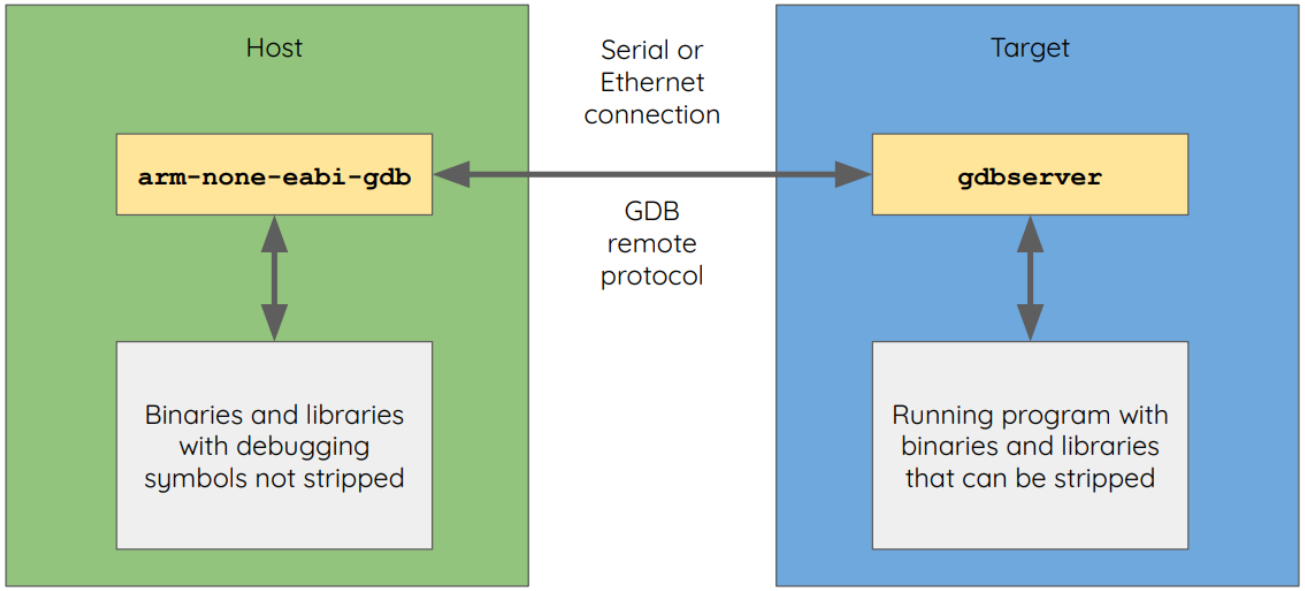
- Typically, GDB is a tool used on the test bench during development.
  - But – you can leverage it after shipping firmware too
- Coredump
  - Captures registers and memory for analysis
    - Triggered by faults, kernel panics, and asserts
    - Data can be streamed out immediately stored in non-volatile memory
  - Coredumps can be used to debug crashes from field devices
    - Collect coredumps when a crash occurs

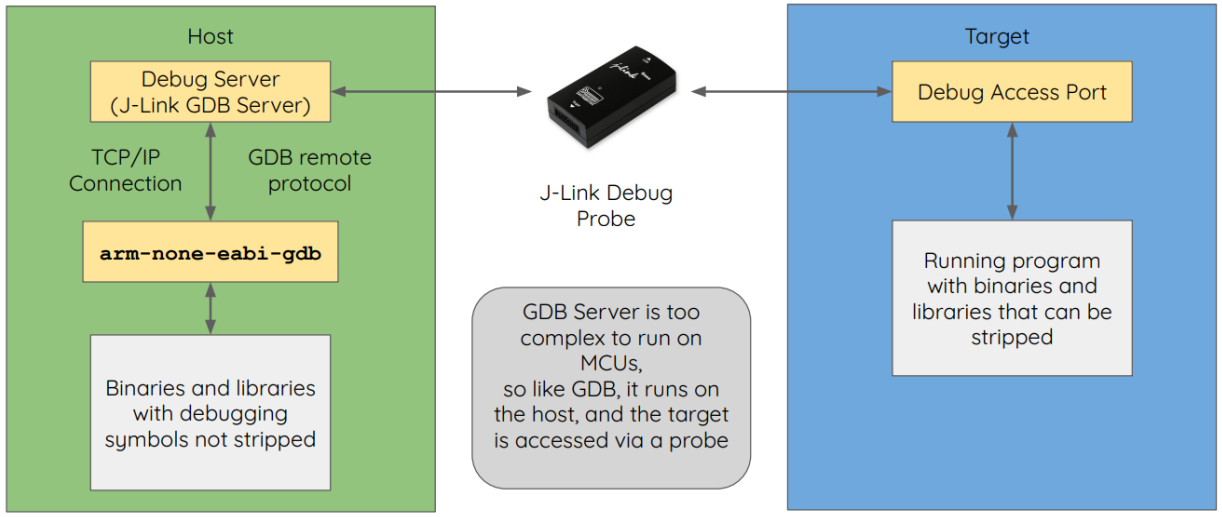



```
$ gdb --se /path/to/symbols.elf --core /path/to/coredump.elf
```

Remote Debug

- Embedded target environments are too limited to run GDB directly
  - gdb is 2.4 MB on x86
- Also, you will likely want host-supported visual debugging options
  - GDB Server can run on target (40 KB on arm)





Demo

Demo

- Example Program (segfault):

```c
#include <stdio.h>

void cause_segfault() {
        int *p = NULL;
        *p = 42;
}
int main() {
        printf("Starting program...\n");
        cause_segfault();
        return 0;
}
~
```

- Example Program (highCPU):

```c
#include <pthread.h>
#include <stdio.h>
void* spin(void* arg) {
        while (1);  // Infinite loop (high CPU)
        return NULL;
}

int main() {
        pthread_t t1, t2;
        pthread_create(&t1, NULL, spin, NULL);
        pthread_create(&t2, NULL, spin, NULL);
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
        return 0;
}
```

Deadlock Analysis Using GDB

- Demonstration with a C program example
- Two threads acquiring two mutexes in opposite order
  - Classic circular wait condition
- Method:
  - Run: ./deadlock_example (program hangs)
    - Find PID: ps aux | grep deadlock_example
    - Attach: gdb -p <PID>
      - Use: info threads, thread <n>, bt
  - All threads stuck in __lll_lock_wait → suspicious
    - Backtrace shows waiting on pthread_mutex_lock
      - Confirms circular wait = deadlock

```c
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;

void* thread1(void* arg) {
    pthread_mutex_lock(&m1);
    sleep(1);
    pthread_mutex_lock(&m2); // waiting...
    return NULL;
}

void* thread2(void* arg) {
    pthread_mutex_lock(&m2);
    sleep(1);
    pthread_mutex_lock(&m1); // waiting...
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```