Data and File Structures Laboratory

Tools: GDB, Valgrind

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

August, 2018

A programmer's experience – Case I

```
int x = 10, y = 25;
x = x++ + y++;
y = ++x + ++y;
printf("%d %d", x, y);
```

A programmer's experience – Case I

```
int x = 10, y = 25;
x = x++ + y++;
y = ++x + ++y;
printf("%d %d", x, y);
```

Output: 36 63 (Output might be ambiguous)

A programmer's experience – Case I

```
int x = 10, y = 25;
x = x++ + y++;
y = ++x + ++y;
printf("%d %d", x, y);
```

Output: 36 63 (Output might be ambiguous)

```
int x = 10, y = 25;
x = x++ + y++;
printf("%d", x);
y = ++x + ++y;
printf(" %d %d", x, y);
```

Output: 35 36 63 (Adding a `printf()` explains the logic)

A programmer's experience – Case II

```
int x = 65, y = 3;
x = x>>1 + y>>1;
printf("%d", x);
```

A programmer's experience – Case II

```
int x = 65, y = 3;
x = x>>1 + y>>1;
printf("%d", x);
```

Output: 2

A programmer's experience – Case II

```
int x = 65, y = 3;
x = x>>1 + y>>1;
printf("%d", x);
```

Output: 2

```
int x = 65, y = 3;
printf("%d %d", x>>1, y>>1);
x = x>>1 + y>>1;
printf(" %d", x);
```

Output: 32 1 2

A programmer's experience – Case II

```
int x = 65, y = 3;
x = x>>1 + y>>1;
printf("%d", x);
```

Output: 2

```
int x = 65, y = 3;
printf("%d %d", x>>1, y>>1);
x = x>>1 + y>>1;
printf(" %d", x);
```

Output: 32 1 2

```
int x = 65, y = 3;
printf("%d", x>>(1+y));
x = x>>1 + y>>1;
printf(" %d", x);
```

Output: 4 2 (What to print is tricky)

A programmer's experience – Case III

```
int x = 10;
printf("%d %d", x, x*x++);
```

A programmer's experience – Case III

```
int x = 10;
printf("%d %d", x, x*x++);
```

Output: 11 110 (May vary compiler to compiler)

A programmer's experience – Case III

```
int x = 10;
printf("%d %d", x, x*x++);
```

Output: 11 110 (May vary compiler to compiler)

```
int x = 10;
printf("%d", x);
printf(" %d", x*x++);
```

Output: 10 110

A programmer's experience – Case III

```
int x = 10;
printf("%d %d", x, x*x++);
```

Output: 11 110 (May vary compiler to compiler)

```
int x = 10;
printf("%d", x);
printf(" %d", x*x++);
```

Output: 10 110

```
int x = 10;
printf("%d", x*x++);
printf(" %d", x);
```

Output: 110 11 (Placing the `printf()` statement is tricky)

Problems with the naive approach of using printf()

1. Too many `printf()`s jumble up the code.
2. Adding `printf()`s at the appropriate places of failure is tricky.
3. Inserting a new code may change program behavior and thus the nature and/or existence of error and/or crash.
4. Inserting new debug codes implies recompilation.

The necessity of a debugger

“If I had 8 hours to chop down a tree, I would spend 6 of those hours sharpening my axe.” – Abraham Lincoln

The necessity of a debugger

“`If I had 8 hours to chop down a tree, I would spend 6 of those hours sharpening my axe.`” – Abraham Lincoln

- The problems like crashes, faults, errors, unexpected outputs, warnings, etc. are common programming experiences.
- Problems can grow numerously (exponentially!!!) with the growth of the code.
- A debugger allows the programmer to have a deeper look at the program *in execution* to explore reasons of the aforementioned problems.
- A debugger can debug single as well as multithreaded programs.

GDB

GDB stands for `The GNU Project Debugger`

- It can start a program, specifying anything that might affect its behavior.
- It can stop a program on specified conditions.
- It can examine what has happened when a program has stopped.
- It can change things in a program so that the effects of one bug can be corrected to learn about another.
- GDB currently supports the following languages: Assembly, Ada, Fortran, C, C++, Objective-C, D, Go, OpenCL, Modula-2, Pascal, Rust.
- The latest release is GDB 8.1.1 (July 31, 2018).

Compiling a program in GDB

- Compile a program to allow the compiler to collect the debugging information
  `$gcc -g <option_number> -o progx progx.c`
- The `<option_number>` denotes amount of debugging information collected (1 = min, 2 = default, 3 = max).
- Access the debugging information of preprocessor macros
  `$gcc -g3 -o progx progx.c`

1. GDB preserves type information from variables.
2. GDB preserves function prototypes.
3. GDB provides a correspondence between the line numbers of source code and instructions.

Note: Adding `-Wall` turns on most of the warning flags related to program constructions.

Running a program in GDB

- Get help from GDB
  `$gdb -h`
- Launch GDB
  `$gdb ./progx`
- Run the program
  `(gdb) run` or `(gdb) r`
- Run the program with command line arguments
  `(gdb) r <argument 1> <argument 2> ...`
- Get help from GDB
  `(gdb) help <command_name>`
- Quit from GDB
  `(gdb) quit` or `(gdb) q`

Running a program in GDB – An example

```
#include<stdio.h>
#include<string.h>
int main(){
    char *str;
    printf("Size of the string = %d", strlen(str));
    return 0;
}
```

Running a program in GDB – An example

```
#include<stdio.h>
#include<string.h>
int main(){
    char *str;
    printf("Size of the string = %d", strlen(str));
    return 0;
}
```

Output:
Segmentation fault (core dumped)

Running a program in GDB – An example

```
#include<stdio.h>
#include<string.h>
int main(){
    char *str;
    printf("Size of the string = %d", strlen(str));
    return 0;
}
```

Output:
Segmentation fault (core dumped)

Output in GDB:
Program received signal SIGSEGV, Segmentation fault.
strlen () at ..

Loading symbol table

- Compile your program to allow the compiler to collect the debugging information

```
$gcc -g -o progx progx.c
```

- Launch GDB

```
$gdb ./progx
```

- Load the symbol table

```
(gdb) file progx
```

Setting a *breakpoint*

- List the source code with line numbers
  `(gdb) l`
- Set a breakpoint at a particular line number
  `(gdb) break <line_number>` or
  `(gdb) b <line_number>`
- Set a breakpoint some lines after the current line
  `(gdb) b +<line_count>`
- Set a breakpoint at the beginning of a function
  `(gdb) b <function_name>`
- Remove all the breakpoints
  `(gdb) delete`

Setting a *breakpoint* – An example

```
1   #include<stdio.h>
2   int main(){
3       int x = 10, y = 25;
4       x = x++ + y++;
5       y = ++x + ++y;
6       printf("%d %d", x, y);
7       return 0;
8   }
```

1. `(gdb) b +3`
   Breakpoint 1 at 0x40054c: file progx.c, line 4.
2. `(gdb) b 5`
   Breakpoint 2 at 0x400563: file progx.c, line 5.

Enabling/disabling a *breakpoint*

- Enable a particular breakpoint
  `(gdb) enable <breakpoint_number>`
- Disable a particular breakpoint
  `(gdb) disable <breakpoint_number>`
- List all breakpoints present in the program
  `(gdb) info break` or `(gdb) info b` or `(gdb) i b`

Note: Breakpoints are enabled by default. They have to be marked disabled using the command.

Enabling/disabling a *breakpoint* – An example

1. `(gdb) b 6`
   Breakpoint 3 at 0x400571: file progx.c, line 6.
2. `(gdb) disable 2`
3. `(gdb) i b`

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x40054c in main	at progx.c:4
2	breakpoint	keep	n	0x400563 in main	at progx.c:5
3	breakpoint	keep	y	0x400571 in main	at progx.c:7

Note: The 'Enb' column (marked as y/n) shows whether a breakpoint is enabled or disabled. The address refers to program counter.

Break condition and command

- Sets a condition on the breakpoint
  ```
  (gdb) b <line_number> if <condition>
  ```
- Run the program
  ```
  (gdb) r
  ```

Break condition and command – An example

```
1    #include<stdio.h>
2    int main(){
3        int x = 65, y = 3;
4        x = x>>1 + y>>1;
5        printf("%d", x);
6        return 0;
7    }
```

1. `(gdb) b main`
   Breakpoint 1 at 0x40053e: file progx.c, line 3.
2. `(gdb) b 4 if x==32`
   Breakpoint 2 at 0x40054c: file progx.c, line 4.
3. `(gdb) run`
   Breakpoint 1, main() at progx.c:3
4. `(gdb) c`

Using Next, Continue and Set commands

- Run the program
  ```
  (gdb) r
  ```
- Run the next line in the program and pause
  `(gdb) next` or `(gdb) n`
- Run the program until a breakpoint/error occurs
  `(gdb) continue` or `(gdb) c`
- Set the value of a locally used variable or argument inside the current function
  ```
  (gdb) set variable <variable_name> = <value>
  ```

Stepping into a function and return

- Run the program
  `(gdb) r`
- Enter into a function
  `(gdb) step` or `(gdb) s`
- Run the next instruction line in the program (single stepping)
  `(gdb) step 1` or `(gdb) s 1`
- Run upto the end of current function and display return value
  `(gdb) finish` or `(gdb) f`

Note: If the instruction involves calling a function, it makes the call to the function and pauses.

Ignoring a *breakpoint* for *N* occurrences

- Ignore the occurrence of a particular breakpoint for the next *N* times

```
(gdb) ignore <breakpoint_number> N
```

- Run the program

```
(gdb) r
```

Listing variables and examining their values

- Set a breakpoint at the line where the required variable appears
  `(gdb) b <line_number>`
- Run the program
  `(gdb) r`
- List the value of the variable
  `(gdb) print <variable_name>` or
  `(gdb) p <variable_name>`
- Print the variables whenever they change
  `(gdb) display <variable_name>`
- List the locally stored variables for the current function
  `(gdb) info local` or `(gdb) i local`

Listing variables and examining their values – An example

```
#include<stdio.h>
int main(){
    int x = 65, y = 3;
    x = x>>1 + y>>1;
    printf("%d", x);
    return 0;
}
```

1. (gdb) b main
2. (gdb) r
3. (gdb) display x
4. (gdb) n
   1: x = 65
5. (gdb) n
   1: x = 2

Printing content of an array or contiguous memory

- Set a breakpoint at the line just after the array
  `(gdb) b <line_number>`
- Run the program
  `(gdb) r`
- Print the array contents
  `(gdb) print <array_name>` or
  `(gdb) p <array_name>`

Printing function arguments

- Set a breakpoint at the line where the function appears

  `(gdb) b <line_number>`

- Run the program

  `(gdb) r`

- Print the function arguments

  `(gdb) info args` or `(gdb) i args`

Examining *stack frame*

- The *stack frame* is a structure which holds execution information for a function call
- Components of the *stack frame* are:
  1. Return pointer
  2. Space for the function's return value (to be populated)
  3. Arguments to the function
  4. Local variables
- A *stack frame* is created for each function call and placed on the top of the stack.
- When a function call returns, the frame corresponding to the function call is removed from the top of the stack.

Examining *stack frame*

- Display backtrace (that is the program stack upto the current point)
  `(gdb) backtrace` or `(gdb) bt`
- Move up the stack frame
  `(gdb) up`
- Move down the stack frame
  `(gdb) down`
- Display a particular frame
  `(gdb) frame <frame_number>`
- Display information about the current stack frame
  `(gdb) info frame` or `(gdb) i frame`
- Display backtrace of all the threads
  `(gdb) thread apply all bt`

Examining *stack frame* – An example

```
#include<stdio.h>
void function1();
void function2();
int main(){
    int i = 10;
    function1();
    printf("In main(): %d\n", i);
}
void function1(int j){
    printf("In function1(): %d\n",j+10);
    function2();
}
void function2(){
    int k = 30;
    printf("In function2() : %d\n",k);
}
```

Examining *stack frame* – An example

1. `(gdb) r`
2. `(gdb) bt`
   No stack. (No stack frame has been created)
3. `(gdb) b 14`
4. `(gdb) r`
5. `(gdb) bt`
   0 function2 () at progx.c:14
   1 0x400568 in function1 () at progx.c:11
   2 0x400525 in main () at progx.c:6
6. `(gdb) delete` (Breakpoints are deleted)
7. `(gdb) r`
8. `(gdb) bt`
   No stack.

Core file debugging

- Run the program

  ```
  (gdb) r
  ```

- Generate core file

  ```
  (gdb) generate-core-file
  ```

- Show the stack frame

  ```
  (gdb) bt
  ```

- Read the contents of a core file

  ```
  $gdb -c core progx  or
  $gdb progx <corefile_name>
  ```

- Invoke GDB on the last core dump

  ```
  $coredumpctl gdb
  ```

Core file debugging – An example

```
#include<stdio.h>
int main(){
    int x = 10;
    printf("%d", x/0);
    return 0;
}
```

1. `r`
   Program received signal SIGFPE, Arithmetic exception.
   ...
2. `(gdb) generate-core-file`
   Saved corefile core.15784 (The file core.15784 is created)
3. `$coredumpctl gdb`
   ... PID: ... UID: ... GID: ...

Debugging of an already running program

- Get the PID of the running program
  `getpid()` // Use unistd.h and sys/types.h
- Attach GDB with the running program
  `$sudo gdb -p <PID_number>`

Setting a *watchpoint*

- Set a watchpoint on a particular variable
  `(gdb) watch <variable_name>`
- Set a read watchpoint on a particular variable
  `(gdb) rwatch <variable_name>`
- Set a read/write watchpoint on a particular variable
  `(gdb) awatch <variable_name>`
- List all watchpoints (along with breakpoints) present in the program
  `(gdb) info break` or `(gdb) info b` or `(gdb) i b`
- Enable a particular watchpoint
  `(gdb) enable <watchpoint_number>`
- Disable a particular watchpoint
  `(gdb) disable <watchpoint_number>`

Setting a *watchpoint* – An example

```c
#include<stdio.h>
int main(){
    int x = 10;
    printf("%d %d", x, x*x++);
}
```

1. `(gdb) b main`
2. `(gdb) r`
3. `(gdb) watch x`
   Hardware watchpoint 2: x
4. `(gdb) c`
   Old value = 0
   New value = 11

User-defined commands

- Define a function of your choice

```
(gdb) define <function_name>
(gdb) ><function_body>
(gdb) >end
```

Example:

1.
```
(gdb) define DIFF
(gdb) >print $arg0 - $arg1
(gdb) >end
```
2.
```
(gdb) DIFF 20 100
```
$1 = -80

Note: The constructs like `if-else-end` and `for` can be used in user-defined commands.

Remote debugging

- Open the gdbserver for the object file along with a port number (user specific)

```
$gdbserver host:5421 progx
```

- Remotely debug the object file

```
(gdb) target remote <IP_address>:<port_number>
```

Valgrind

Valgrind is a framework that includes tools for automatically detecting many memory management and threading bugs and profiling C/C++ programs. The current Valgrind distribution includes the following production-quality tools:

1. A memory error detector
2. Two thread error detectors
3. A cache and branch-prediction profiler
4. A call-graph generating cache
5. A branch-prediction profiler
6. A heap profiler.

It also includes three experimental tools: a stack/global array overrun detector, a second heap profiler that examines how heap blocks are used, and a SimPoint basic block vector generator.

Memcheck

The *Memcheck* can detect memory-management problems. It checks all read/write operations on memory and intercept the calls to malloc/new/free/delete.

- Compile a program to allow the compiler to collect the debugging information

```
$gcc -g <option_number> -o progx progx.c
```

- The `<option_number>` denotes amount of debugging information collected (1 = min, 2 = default, 3 = max).
- Running the Memcheck tool of Valgrind and create a log file

```
$valgrind --tool=memcheck --leak-check=full
--log-file=progx_memcheck.log ./progx
```

- As Memcheck tool is the default in Valgrind and mentioning the log file is optional, alternatively

```
$valgrind --leak-check=yes progx.c
```

Memcheck – An example

1. `$valgrind --tool=memcheck --leak-check=full --log-file=progx_memcheck.log ./progx`
2. `$vi progx_memcheck.log`

   ...

   ==9202== HEAP SUMMARY:

   ==9202== in use at exit: 0 bytes in 0 blocks

   ==9202== total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated

   ...

Memcheck – Understanding the leak summary

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int i, *a;
    for(i=0; i < 10; i++)
        a = malloc(sizeof(int) * 100);
    free(a);
    return 0;
}
```

1 `$vi progx_memcheck.log`

==24810== LEAK SUMMARY:
==24810== definitely lost: 3,600 bytes in 9 blocks
==24810== indirectly lost: 0 bytes in 0 blocks
==24810== possibly lost: 0 bytes in 0 blocks
==24810== still reachable: 0 bytes in 0 blocks
==24810== suppressed: 0 bytes in 0 blocks

Memcheck – Understanding the leak summary

- **definitely lost** – memory is lost for sure (should be fixed!!!)
- **indirectly lost** – memory for the root node of a tree is free but not the rest of it (should be fixed!!!)
- **possibly lost** – memory is actually lost due to reasons like function allocates a buffer and returns it but the caller never frees the memory or a lot of allocated memory is actually not in need (should be fixed!!!)
- **still reachable** – memory is probably still in use at the end of the program (not a serious issue!!!)

Cachegrind

The *Cachegrind* is a cache profiler. It performs detailed simulation of the I1, D1 and L2 caches in your CPU and so can accurately pinpoint the sources of cache misses in your code. It identifies the number of cache misses, memory references and instructions executed for each line of source code, with per-function, per-module and whole-program summaries.

- Running the Cachegrind tool of Valgrind and create a log file

```
$valgrind --tool=cachegrind --log-file=
progx_cachegrind.log ./progx
```

- Open the log file

```
$vi progx_cachegrind.log
```

Cachegrind – An example

1. `$valgrind --tool=cachegrind --log-file=progx_cachegrind.log ./progx`
2. `$vi progx_cachegrind.log`

...

–9411– warning: L3 cache found, using its data for the LL simulation.

==9411== I refs: 264,449

==9411== I1 misses: 981

==9411== LLi misses: 967

==9411== I1 miss rate: 0.37%

==9411== LLi miss rate: 0.37%

...

Callgrind

The *Callgrind* is an extension to Cachegrind. It provides all the information that Cachegrind does, plus extra information about callgraphs.

- Running the Callgrind tool of Valgrind and create a log file
  ```
  $valgrind --tool=callgrind --inclusive=yes
  --tree=both --log-file=progx_callgrind.log ./progx
  ```
- Open the log file
  ```
  $vi progx_callgrind.log
  ```

Massif

The *Massif* is a heap profiler. It performs detailed heap profiling by taking regular snapshots of a program's heap. It produces a graph showing heap usage over time, including information about which parts of the program are responsible for the most memory allocations.

- Running the Massif tool of Valgrind and create a log file

```
$valgrind --tool=massif --log-file=
progx_massif.log ./progx
```

- Open the log file

```
$vi progx_massif.log
```

Helgrind

The *Helgrind* is a thread debugger which finds data races in multithreaded programs. It looks for memory locations which are accessed by more than one (POSIX p-)thread, but for which no consistently used (pthread_mutex_) lock can be found. Such locations are indicative of missing synchronisation between threads, and could cause hard-to-find timing-dependent problems.

- Running the Helgrind tool of Valgrind and create a log file
  ```
  $valgrind --tool=helgrind --log-file=
  progx_helgrind.log ./progx
  ```
- Open the log file
  ```
  $vi progx_helgrind.log
  ```

Other tools

- The *Lackey* and *Nulgrind* are also included in the Valgrind distribution. They are used for testing and demonstrative purposes.
- The *DHAT* is a tool for examining how programs use their heap allocations. It tracks the allocated blocks, and inspects every memory access to find which block, if any, it is to.
- The *BBV* tool provides a list of all basic blocks (a linear section of code with a single entry and exit point) entered during program execution, and a count of how many times each block was run.
- The *SGCheck* is a tool for finding overruns of stack and global arrays. It works by using a heuristic approach derived from an observation about the likely forms of stack and global array accesses.

Problems – Day 12

1. Debug the program Test1.c with GDB using breakpoints and other features.
2. Debug the program Test2.c with GDB using watchpoints and other features.
3. Debug the program Test3.c with GDB using core debugging and other features.
4. Pick up any problem of your choice (preferably function-pointers.c) from the DFS Lab course page that is yet unfinished by you and examine it with GDB.

Problems – Day 12

5. Test the program Test2.c with Valgrind to figure out the reason of stack smashing shown while debugging it with GDB.
6. Work on the program Test4.c with the Memcheck tool of Valgrind to examine the details of memory management.
7. Work on the program Test5.c with the various Valgrind tools to examine the possible errors and weaknesses.