Linux Application Debugging

1. Debugging

- A "bug" is an error, flaw, or fault in a computer program that prevents it from behaving as intended. — Wikipedia
- Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program thus making it behave as expected. — Wikipedia

2. Debugging Techniques

- Debugging by printing - debug information is displayed at suitable points.
- Debugging by querying - debug information is provided when requested by developer using debuggers.
- Debugging by watching - code is debugged by watching its interaction with another program.

3. Compiling with debug symbols

The *C* program need to be compiled with debug symbols as shown below to use with gdb. gcc has option *-g* to enable debug symbols in an executable.

```
$gcc -g code.c -o codedbg
```

4. CoreDump Analysis

- To enable a coredump file to be created when a application crashes, we may need to set resource limit to *unlimited*.

```
$ulimit -c unlimited
```

- execute the program which crashes

```
$./codedbg
```

- The crashing application would print `Segmentation fault (core dumped)` and would produce a *core* file in the same folder.
- the core file can be analyzed with help of gdb as shown below.

```
 gdb <executable> <corefile>
```

5. Source Level Debugging

- The application can be debugged line by line of a source code using gdb. GDB on invocation provides a prompt to accept gdb commands

```
$gdb codedbg
(gdb)
```

- The application can be run using run command of gdb

```
(gdb) run <arguments>
```

- The application can be initiated and stopped at main() function as

```
(gdb) start <arguments>
```

- Continue the stopped application as

```
(gdb) continue
```

- Application can be stepped, line by line with *step* command. Subsequent carriage return(Enter) would repeat the previously executed command. *step* command would step into the function calls. If we don't need to step into function call, *next* can be used.

```
(gdb) step
(gdb)
(gdb) next
```

- the source file can be listed in gdb by function name or line number as

```
(gdb) list [function | line]
```

- to print the current line under debugging info command can be used

```
(gdb) info line
(gdb) info functions
```

- the breakpoints can be set in the program lines using

```
(gdb) br [line | function | address]
```

- the call stack trace of a program can be printed using *bt* command with *full* as argument, it can print the local variables in the functions along with stack trace.

```
(gdb) bt [full]
```

6. Inspecting Variables

- the variable values can be printed with help of print command, it can take switches to print in hexadecimal/octal format.

```
(gdb) print [variable]


(gdb) print string
(gdb) print array[1]
(gdb) print /x array[8]
(gdb) print structure
(gdb) print structure.name
(gdb) print structure.age
```

- A memory area can be dumped to examine as

```
(gdb) x/fmt <addr>
(gdb) x/10 array
(gdb) x/10x array
```

- The variables defined in current functions can be listed as below

```
(gdb) info locals
```

- A type of variable can be understood as below

```
(gdb) whatis structure
```

- Assembly instructions and current registers values can be viewed using

```
(gdb) info registers
(gdb) disassemble
```

- testing certain functions by passing values from debugger

```
(gdb) print add(5,10)
```

- changing a variable value in debugger is possible by

```
(gdb) set var c=25
(gdb) print c=25
```

7. Helper Calculations

- basic math operations can be done as

```
(gdb) print 1000 * 1000
```

- format conversions can also be done as

```
(gdb) print /x 0x100
(gdb) print /t 0x100
(gdb) print 0x400
```

8. Remote Debugging

- In a target machine the debugging can be started using a gdbserver with a any portno and host can be left as optional.

```
target$ gdbserver host:port bin
```

- it is possible to debug a running process with its PID as

```
target$ gdbserver host:port --attach <pid>
```

- From the linux host machine, it is possible to connect to the target using the ipaddress.

```
(gdb) target remote host:port
```

- To the gdb, need to provide path of the rootfs for target board where it could look for libraries

```
(gdb) set sysroot /path/to/rootfs
```

9. Debugging by Watching

- Some of the errors related to calling library functions can be understood and analyzed by watching how a function is called by a application.

- `ltrace` - traces the library calls done by a application.
- Sometimes it is possible to weed out bugs by watching the interaction between user space and kernel space.
- `strace` - traces system calls and signals of a process.
- It intercepts and records the system calls which are called by a process and the signals which are received by a process.
- Usually errors from system calls are printed using `perror()`. But this does not provide information like the arguments passed and exactly which call triggered the error.
- `strace` provides information like the arguments to the system call, return values, errors if any, the time spent in a system call, whether signals occurred during the operation.
- For the simplest case has the following format

```
strace <program>
```

- Options are available to provide additional information and filter away unwanted information.
- `-e trace=syscall1,syscall2,...`
- `-e trace=class` class is `file`, `process`, `network`, `signal`, `ipc`, ...
- `-e signal=signal1,signal2...`
- `-T` print time spent in each syscall
- `-c` per syscall statistics - time spent, no. of calls, errors
- `-p pid` attach to a running process
- Stracing a process can show the syscalls used, their arguments and the return value, as in below example

```
$ strace cat /dev/null
....
open("/dev/null", O_RDONLY|O_LARGEFILE) = 3
read(3, "", 4096)                       = 0
close(3)                                = 0
....
```

- When there is an error, strace can print the error value and the error description.

```
$ strace cat /nofile
....
open("/nofile", O_RDONLY|O_LARGEFILE)   = -1 ENOENT (No such file or directory)
write(2, "cat: can't open '/nofile': No su"..., 53cat: can't open ./nofile': No such fi
...
```

- Strace can print the signal received, when the traced process receives a signal. Below example of killing a sleep command can produce trace log for signal.

```
$ strace sleep 100 &
$ killall sleep
....
--- SIGTERM (Terminated) @ 0 (0) ---
# +++ killed by SIGTERM +++
....
```

10. Resource Usage

- The `top` command can be used to analyze the cpu utilization and memory utilization of a process.
- The `free` command can be used to get memory details, and `df` command can be used to get storage details of the system.
- The *pmap* command can be used to get memory details, and *du* command be used to get disk usage of a process.