Advanced multi-process debugging using GDB

ABSTRACT

By default, GDB is designated to debug a single process with its threads.
In the following document we will go through some usefull advance options
that allows us to have more control over the program debugged.

GDB options

inline shell call

It is possible to run shell commands in gdb.
usage example

```
shell umount /some/folder
shell rmuser some_user
```

etc..

Logging option

by default, gdb shell is not saved to a file,
using
`set logging overwrite on` would save any proceeding output in the ternimal also to `gdb.txt`
in case you want to override the file or change the file

```
set logging file FILE_NAME
```

it is required to enable logging so output will be written to the logging file using

```
set logging enabled on
```

fork options

gdb does not keep tracked on child forked, in case we want to follow child after fork we set
`set follow-fork-mode child`
in case we wish to keep both process under gdb domain we use
`set detach-on-fork off`

scheduling options

usually gdb “freezes” all activity when getting into breakpoint,
sometimes we wish to let other threads continue and reach timeout or just do what they do,
this behavior can be modified using the followings,

```
set schedule-multiple on #Resuming the execution of threads of all processes
is on.
```

we can also disable context switch which debugging single thread using

```
set scheduler-locking on #default is reply
```

breakpoint options

breakpoints are the basic of debugging, otherwise why are we here for?
but it is important to know that breakpoint have many other options

breakpoints can be stored to a file
`save breakpoint break_point_for_pr1111`
and we can load then on a different run instead of defining them over and over
`` `source break_point_for_pr1111` ``

we can defile an action we wish to happen upon breakpoint.
using `command` option
suppose we have `int *intPtr = (int * ) malloc(sizeof(SOME_SIZE)`
and we wish to test our false branch
we can do:

```
command br BR_NUMBER
p intPtr=NULL
end
```

we can do that for range of breakpoints
e.g we with to do backtrace on when getting into breakpoint 3-10

```
commands 3-10
echo \n\n
thread apply all bt
end
```

we can also define to break only when we are on thread_id

```
(gdb) command
Type commands for breakpoint(s) 20, one per line.
End with a line saying just "end".
>if $_thread_id=1
 >thread apply all bt
 >end
>end
```

watchpoint option

watch point are on of the best ways to trace variable that have been free by some context
you can add do watch for adress or by value
the value option with -L (location) will trace the address and it is the prefered way because it is simpler and remains global

```
watch *0x7fffffffe1bc
(gdb) c
Continuing.

Hardware watchpoint 21: *0x7fffffffe1bc

Old value = 1
New value = 2
(gdb) watch -l i
Hardware watchpoint 23: -location i

(gdb) info b 23
Num     Type           Disp Enb Address            What
23      hw watchpoint  keep y                      -location i
(gdb) c
Continuing.

Hardware watchpoint 23: -location i

Old value = 4
New value = 5
```

putting most of it together

creating `gdb_config_file` with the following

```
# gdb -x gdb_config_file --args build/bin/nqdrv [args]

# umount to get clean env
shell umount /home/nlevy/shares/mnt

# logging setting
```

```
set logging overwrite on
set logging file gdb_full_report
set logging enabled on

# do not dettach and follow child on fork
set detach-on-fork off
set follow-fork-mode child

# breakpoints
b main
r
break some_func2
break some_func3
break some_func4
break some_func5
break some_func6


commands 2-6
echo \n\n
thread apply all bt
continue
end
```

result with `gdb_full_report` that show threads througout the program main execution path where we see all threads created by quic, trace, etc..

References:

[GDB Documentation (sourceware.org)]
[gdb.pdf (sourceware.org)]