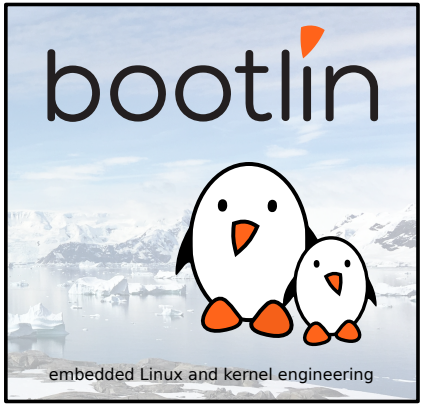
Debugging with GDB and remote GDB

Luca Ceresoli
luca.ceresoli@bootlin.com



Luca Ceresoli

- Embedded Linux engineer at **Bootlin**
  - Embedded Linux experts
  - Engineering services: Linux BSP development, kernel porting and drivers, Yocto/Buildroot integration, real-time, boot-time, security, multimedia
  - Training services: Embedded Linux, Linux kernel drivers, Yocto, Buildroot, graphics stack, boot-time, real-time, debugging, audio
- Linux kernel and bootloader development, Buildroot and Yocto integration
- Open-source contributor
- Living in **Bergamo**, Italy
- luca.ceresoli@bootlin.com

https://bootlin.com/company/staff/luca-ceresoli/

Introduction

Debugging

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"
– Brian Kernighan

Good practices

- Keep your code clean and simple!
  - Simple to implement, understand, debug... also helps writing less bugs!
- Compiler are now smart enough to detect a wide range of errors at compile-time using warnings
  - Using `-Werror -Wall -Wextra` is recommended if possible to catch errors as early as possible
- Compilers now offer static analysis capabilities
  - GCC allows to do so using the -fanalyzer flag
  - LLVM provides dedicated tools that can be used in build process

GDB

GDB: GNU Project Debugger

- The debugger on GNU/Linux, available for most embedded architectures.
- Supported languages: C, C++, Pascal, Objective-C, Fortran, Ada...
- Command-line interface
- Integration in many graphical IDEs
- Can be used to
  - control the execution of a running program, set breakpoints or change internal variables
  - to see what a program was doing when it crashed: post mortem analysis
- https://www.gnu.org/software/gdb/
- https://en.wikipedia.org/wiki/Gdb
- New alternative: *lldb* (https://lldb.llvm.org/) from the LLVM project.

GDB crash course

GDB crash course (1/3)

- GDB is used mainly to debug a process by starting it with *gdb*
  - `$ gdb <program>`
- GDB can also be attached to running processes using the program PID
  - `$ gdb -p <pid>`
- When using GDB to start a program, the program needs to be run with
  - `(gdb) run`

GDB crash course (2/3)

A few useful GDB commands

- `break foobar` (`b`)
  Put a breakpoint at the entry of function `foobar()`
- `break foobar.c:42`
  Put a breakpoint in `foobar.c`, line 42
- `print var`, `print $reg` or `print task->files[0].fd` (`p`)
  Print the variable `var`, the register `$reg` or a more complicated reference. GDB can also nicely display structures with all their members
- `info registers`
  Display architecture registers

GDB crash course (3/3)

- `continue` (`c`)
  Continue the execution after a breakpoint
- `next` (`n`)
  Continue to the next line, stepping over function calls
- `step` (`s`)
  Continue to the next line, entering into subfunctions
- `stepi` (`si`)
  Continue to the next instruction
- `finish`
  Execute up to function return
- `backtrace` (`bt`)
  Display the program stack

GDB advanced commands

GDB advanced commands (1/3)

- `info threads` (`i threads`)
  Display the list of threads that are available
- `info breakpoints` (`i b`)
  Display the list of breakpoints/watchpoints
- `delete <n>` (`d <n>`)
  Delete breakpoint <n>
- `thread <n>` (`t <n>`)
  Select thread number <n>
- `frame <n>` (`f <n>`)
  Select a specific frame from the backtrace, the number being the one displayed when using `backtrace` at the beginning of each line

GDB advanced commands (2/3)

- `watch <variable>` or `watch \*<address>`
  Add a watchpoint on a specific variable/address.
- `print variable = value` (`p variable = value`)
  Modify the content of the specified variable with a new value
- `break if condition == value`
  Break only if the specified condition is true
- `watch if condition == value`
  Trigger the watchpoint only if the specified condition is true
- `x/<n><u> <address>`
  Display memory at the provided address. `n` is the amount of memory to display, `u` is the type of data to be displayed (`b/h/w/g`). Instructions can be displayed using the `i` type.

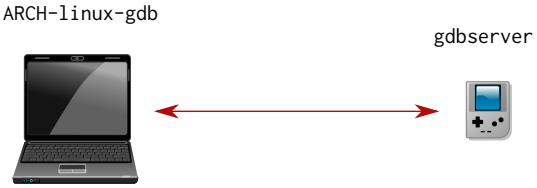
GDB advanced commands (3/3)

- `list <expr>`
  Display the source code associated to the current program counter location.
- `disassemble <location,start_offset,end_offset>` (`disas`)
  Display the assembly code that is currently executed.
- `p $newvar = value`
  Declare a new gdb variable that can be used locally or in command sequence
- `p function(arguments)`
  Execute a function using GDB. NOTE: be careful of any side effects that may happen when executing the function
- `define <command_name>`
  Define a new command sequence. GDB will prompt for the sequence of commands.

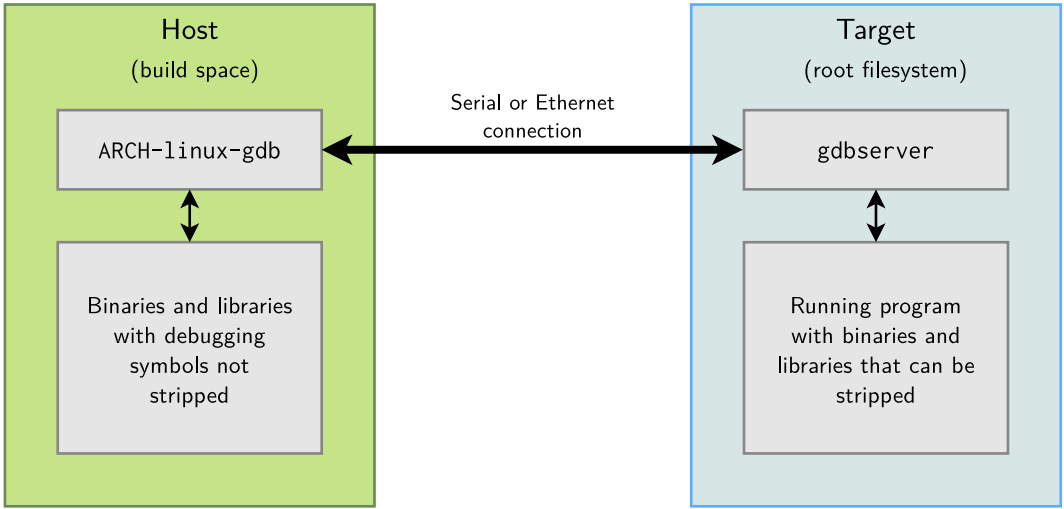
Remote GDB debugging

Remote debugging

- In a non-embedded environment, debugging takes place using gdb or one of its front-ends.
- gdb has direct access to the binary and libraries compiled with debugging symbols.
- However, in an embedded context, the target platform environment is often too limited to allow direct debugging with gdb (2.4 MB on x86).
- Remote debugging is preferred
  - ARCH-linux-gdb is used on the development workstation, offering all its features.
  - gdbserver is used on the target system (only 400 KB on arm).



Remote debugging: architecture



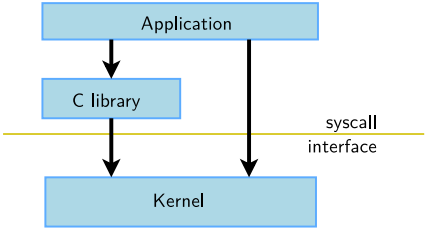
Remote debugging: usage

- On the target, run a program through `gdbserver`.
  Program execution will not start immediately.
  `gdbserver localhost:<port> <executable> <args>`
  `gdbserver /dev/ttyS0 <executable> <args>`
- Otherwise, attach `gdbserver` to an already running program:
  `gdbserver --attach localhost:<port> <pid>`
- Then, on the host, start `ARCH-linux-gdb <executable>`,
  and use the following `gdb` commands:
  - To tell `gdb` where shared libraries are:
    `gdb> set sysroot <library-path>` (typically path to build space without `lib/`)
  - To connect to the target:
    `gdb> target remote <ip-addr>:<port>` (networking)
    `gdb> target remote /dev/ttyUSB0` (serial link)

How a debugger works (within an Operating System)

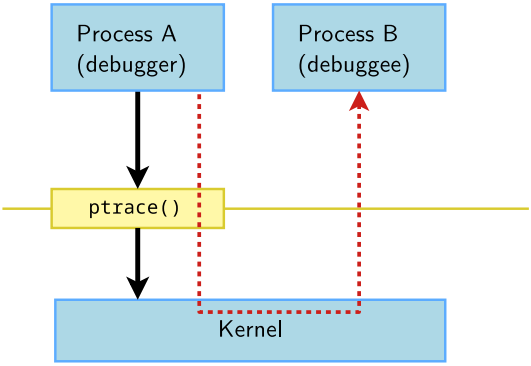
System architecture

- Porcesses do not access the hardware resources directly
- The kernel isolates processes from teh hardware and from other processes
- Processes ask the kernel to provide its services via syscalls (usually wrapped by the C library)



Debugger and debuggee

- *debugger* and *debuggee* are different processes, which normally cannot acccess each other memory and control execution
- A debugger uses the `ptrace()` syscall to control a process execution and read/write its data



ptrace

- The *ptrace* mechanism allows processes to trace other processes by accessing tracee memory and register contents
- A tracer can observe and control the execution state of another process
- Works by attaching to a tracee process using the `ptrace()` system call (see `man 2 ptrace`)
- Can be executed directly using the `ptrace()` call but often used indirectly using other tools.

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

- Used by *GDB*, *strace* and all debugging tools that need access to the tracee process state

Questions? Suggestions? Comments?

Luca Ceresoli
luca.ceresoli@bootlin.com

Rights to copy

Document sources:
https://bootlin.com/pub/conferences/
https://github.com/bootlin/training-materials/