Yocto Project Summit 2021.11

Hands on class
Linux debugging on Yocto Project based systems

Embedded Labworks

By Sergio Prado. São Paulo, November 2021

Embedded Labworks

$ WHOAMI

- Embedded software developer for more than 25 years.
- Consultant and trainer at Embedded Labworks.
  https://e-labworks.com/en
- Open source contributor (Buildroot, Yocto Project, Linux, etc).
- Blogger.
  https://embeddedbits.org/

AGENDA

1. Introduction to debugging on YP based systems & Hands on 1
2. Core dump analysis & Hands on 2
3. Remote debugging with GDB & Hands on 3
4. Kernel oops analysis & Hands on 4
5. Memory leak & Hands on 5

You can find the layer created for the exercises on GitHub.
https://github.com/sergioprado/meta-yps2021-debugging

THIS CLASS IS NOT ABOUT...

- ✗ How Yocto Project/OpenEmbedded works.
- ✗ Debugging BitBake metadata (recipes, classes, etc).
- ✗ Debugging build problems.

This talk will cover tips and tricks on debugging kernel and userspace applications on Linux systems generated with OpenEmbedded and the Yocto Project.

Embedded Labworks

Linux debugging

Introduction to debugging on YP based systems

DEBUGGING TOOLS AND TECHNIQUES

- **Logging and memory dump analysis**: dmesg, kernel oops/panic, addr2line, core dump, etc.
- **Interactive debugging**: GDB, KGDB.
- **Tracing and profiling**: ftrace, systemtap, perf, LTT-NG, gprof, strace, ltrace, etc.
- **Debugging frameworks**: kmemleak, valgrind, mtrace, etc.

APPROACHES TO DEBUGGING

- There are a few approaches to debugging an embedded Linux device:
  - Directly on the target (may not be possible, depending on available resources).
  - Remote debugging via some connection to the target (network, serial port, etc).
  - Post-mortem analysis (here we collect debug information in the target but do the analysis in the host).

WHAT WE (MIGHT) NEED TO DEBUG

- **Source code** of the binary (and libraries) to be debugged.
- Binary (and libraries) compiled with **debugging symbols**.
- Binary (and libraries) compiled with **optimizations disabled**.
- Binary (and libraries) compiled with **security flags disabled**.
- Many host and target **tools**!

SOURCE CODE

- The source code of the binary (and libraries) is not shipped in the final image, but may be needed by a debugging tool like GDB.
- The source code can be included in the image with the `-src` package.
- Including all source packages:

```
IMAGE_FEATURES:append = " src-pkgs"
```

- Including the source package of a specific application:

```
IMAGE_INSTALL:append = " busybox-src"
```

DEBUGGING SYMBOLS (1)

- Debug symbols are special entries in the symbol table of an object file that make it possible for tools like GDB and `addr2line` to gain access to information from the source code of the binary (and ultimately convert addresses into file names and line numbers).

- On Yocto, debugging symbols are removed by default from binaries and split into a separate package (`-dbg`).

- Including all debug packages:

```
IMAGE_FEATURES:append = " dbg-pkgs"
```

- Including the debug package of a specific application:

```
IMAGE_INSTALL:append = " busybox-dbg"
```

DEBUGGING SYMBOLS (2)

- Another option is to just disable stripping when packaging (this is usually not necessary):

  ```
  INHIBIT_PACKAGE_STRIP:pn-busybox = "1"
  ```

- Using `debuginfod` is yet another option:

  https://docs.yoctoproject.org/dev-manual/common-tasks.html#using-the-debuginfod-server-method

  https://www.youtube.com/watch?v=S3QLr113mx8

- If you are doing remote debugging, symbol resolution will be done in the host, and probably you don't need to care about installing debugging packages in the target.

Embedded Labworks

OPTIMIZATION (1)

- When building an application, the compiler may generate an optimized binary that does not match the source code, impacting the debug experience.
- The variable `SELECTED_OPTIMIZATION` specifies the optimization flags passed to the C compiler when building for the target.
- `SELECTED_OPTIMIZATION` takes by default the value of `FULL_OPTIMIZATION`:

```
FULL_OPTIMIZATION = "-O2 -pipe ${DEBUG_FLAGS}"
```

- If `DEBUG_BUILD="1"`, then the variable `SELECTED_OPTIMIZATION` will take the value of `DEBUG_OPTIMIZATION`:

```
DEBUG_OPTIMIZATION = "-Og ${DEBUG_FLAGS} -pipe"
```

- This may vary depending on the distro you are building (e.g. `poky-tiny` sets `-Os` to enable size optimization).

OPTIMIZATION (2)

- Compiling all applications without optimization:

```
DEBUG_BUILD = "1"
```

- Compiling a specific application without optimization:

```
DEBUG_BUILD:pn-busybox = "1"
```

- Be aware that disabling optimizations may influence the system's behavior (it may even hide bugs!), so do it only when necessary.

SECURITY FLAGS

- If your distro is building applications with security flags enabled, the debugging experience might be impacted.

- This is true for Poky-based distros (currently it includes `security_flags.inc`), which enables security-related compiler flags by default.

```
require conf/distro/include/security_flags.inc
```

- In particular, if PIE (Position Independent Executable) is enabled, we are not able to easily resolve symbols, and you might want to disable it for a specific application during a debugging session:

```
SECURITY_CFLAGS:pn-busybox = "${SECURITY_NOPIE_CFLAGS}"
```

TARGET DEBUGGING TOOLS (1)

- To debug an application directly on the target, you will probably need to add a few extra tools to the image.

- Adding GDB to the rootfs for native debugging:

```
IMAGE_INSTALL:append = " gdb"
```

- Adding GDB server to the rootfs for remote debugging:

```
IMAGE_INSTALL:append = " gdbserver"
```

- The same can be achieved by adding `tools-debug` to `IMAGE_FEATURES` (it will add `gdb`, `gdbserver`, `mtrace` and `strace`):

```
IMAGE_FEATURES:append = " tools-debug"
```

TARGET DEBUGGING TOOLS (2)

- Adding Valgrind to the rootfs for memory analysis (the dbg package might also be needed for symbol resolution):
- `IMAGE_INSTALL:append = " valgrind valgrind-dbg"`

- Adding profiling tools (perf, blktrace, powertop, systemtap, lttng, valgrind, etc):

```
IMAGE_FEATURES:append = " tools-profile"
```

- Enable Eclipse debugging (gdbserver, tcf-agent, openssh-sftp-server):

```
IMAGE_FEATURES:append = " eclipse-debug"
```

- For easy connection to the target, `debug-tweaks` is useful:

```
EXTRA_IMAGE_FEATURES = "debug-tweaks"
```

HOST DEBUGGING TOOLS (1)

- In the host machine, for remote debugging or post-mortem analysis, you will need:
  - **Sysroot** (including the source code and binaries with debugging symbols).
  - **Debugging tools** (GDB, addr2line, etc).
- One can generate a stripped-down version of the filesystem for debugging with the following configuration:

```
IMAGE_GEN_DEBUGFS = "1"
IMAGE_FSTYPES_DEBUGFS = "tar.bz2"
```

HOST DEBUGGING TOOLS (2)

- Then a minimal toolchain with only the tools can be generated with:

```
$ bitbake meta-toolchain
```

- Another approach is to generate a complete SDK that will contain the sysroot with debugging symbols and all required tools:

```
$ bitbake -c populate_sdk <your-image>
```

HANDS ON 1: ENVIRONMENT (1)

- Connect to the build server:

```
$ ssh <username>@<hostname>
```

- Go to the poky directory and inspect its content (don't inspect `meta-yps2021-debugging` yet, since there are some patches in this layer that were created to inject bugs in the OS for the exercises):

```
$ cd yp-summit-nov-21/poky/
$ ls
```

- Initialize the build environment:

```
$ source ./oe-init-build-env build-debug
```

- Inspect the `local.conf` file (some packages are already enabled and prebuilt to prevent long build times during the class):

```
$ cat conf/local.conf
```

HANDS ON 1: ENVIRONMENT (2)

- The image `core-image-base` is already built, so you don't need to build it again.
- Run the image with QEMU - there are a few bugs in the image, but the boot should work just fine ;-)

```
$ runqemu nographic qemuarm64
```

- Login with the `root` user (no password) and play around with the OS. When you are done, exit QEMU by typing `CTRL-a` + `x`.
- Open another terminal (use a terminal multiplexer like `tmux` or create a new ssh connection) and test the SDK:

```
$ cd ~/yp-summit-nov-21/poky/sdk-debug
$ source ./environment-setup-cortexa57-poky-linux
$ echo $CC
```

Linux debugging

Core dump analysis

CORE DUMP

- Core dump is a file that contains a snapshot of the memory of a process.
- The core dump is generated when a process receives some signals, including `SIGQUIT`, `SIGILL`, `SIGABRT`, `SIGFPE` and `SIGSEGV`.
- One of the most common situations of a core dump generation is when an application abruptly terminates its execution due to a crash (invalid memory access, division by zero, etc).
- The core dump can be used in a debugger to inspect the state of the process at the time it was terminated.

CORE DUMP REQUIREMENTS

- There are a few requirements for the core dump file to be generated, including:
  - The process needs permission to create the core dump file.
  - It's mandatory to have enough space on the file system.
  - The core file size limit setting (RLIMIT_CORE) for the process needs to be properly configured.

- The process's RLIMIT_CORE setting is usually zero, which prevents the core dump file from being generated. To change this setting, we can use the ulimit command:

```
# ulimit -c unlimited
```

Embedded Labworks

CORE DUMP GENERATION

- By default, the name of the core dump file is `core`, generated in the working directory of the application.
- The name of the core dump file can be changed in `/proc/sys/kernel/core_pattern`.
- More information is available on the `core` man page:

```
$ man 5 core
```

CORE DUMP ANALYSIS ON YOCTO (1)

- Let's say an application called `myapp` is crashing:

```
# myapp
Segmentation fault
```

- The first step is to recompile the application without the security flags:

```
# add to local.conf
SECURITY_CFLAGS:pn-myapp = "${SECURITY_NOPIE_CFLAGS}"
```

- Now you can generate the core dump file and transfer it to your host machine:

```
# ulimit -c unlimited

# myapp
Segmentation fault (core dumped)

# ls -l core
-rw------- 1 root root 438272 Nov  5 18:14 core
```

CORE DUMP ANALYSIS ON YOCTO (2)

- Next, setup the SDK environment and go to the sysroot directory:

```
$ source <SDK_DIR>/environment-setup-cortexa57-poky-linux
$ cd <SDK_DIR>/sysroots/cortexa57-poky-linux/
```

- To do the core dump analysis with GDB, you will need the **source code** of the application and the binary with **debugging symbols**.

- To make it easier for GDB to find the sources, you can set the application's source code directory in its configuration file:

```
$ echo "dir usr/src/debug/myapp/1.0-r0/myapp-1.0/src" >> ~/.gdbinit
```

- To search for the binary with debugging symbols, you can simply use `find`:

```
$ find . -name myapp 2>/dev/null | grep "\.debug"
./usr/sbin/.debug/myapp
```

CORE DUMP ANALYSIS ON YOCTO (3)

- Now you can run GDB to analyse the core dump file:

```
$ aarch64-poky-linux-gdb ./usr/sbin/.debug/myapp -c ~/core
...
Core was generated by `myapp'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0 0x0000000000401950 in main (void) at ../../myapp-1.0/src/main.c:15
15            *ptr = 10;
(gdb)
```

- You can also use `-tui` for a more graphical view of the source code:

```
$ aarch64-poky-linux-gdb -tui ./usr/sbin/.debug/myapp -c ~/core
```

HANDS ON 2: CORE DUMP ANALYSIS

- Run the command below in the target device (QEMU). The execution will fail with a `Segmentation Fault` error.

  ```
  # fping -c 5 192.168.0.1
  ```

- Generate a core dump file and transfer it to the host machine.
- In the host machine, use GDB to find the line of code in the source code of the `fping` application that is causing the issue.
- Obs.: You don't have to fix the issue, just find the root cause.

Embedded Labworks

Linux debugging

Remote debugging with GDB

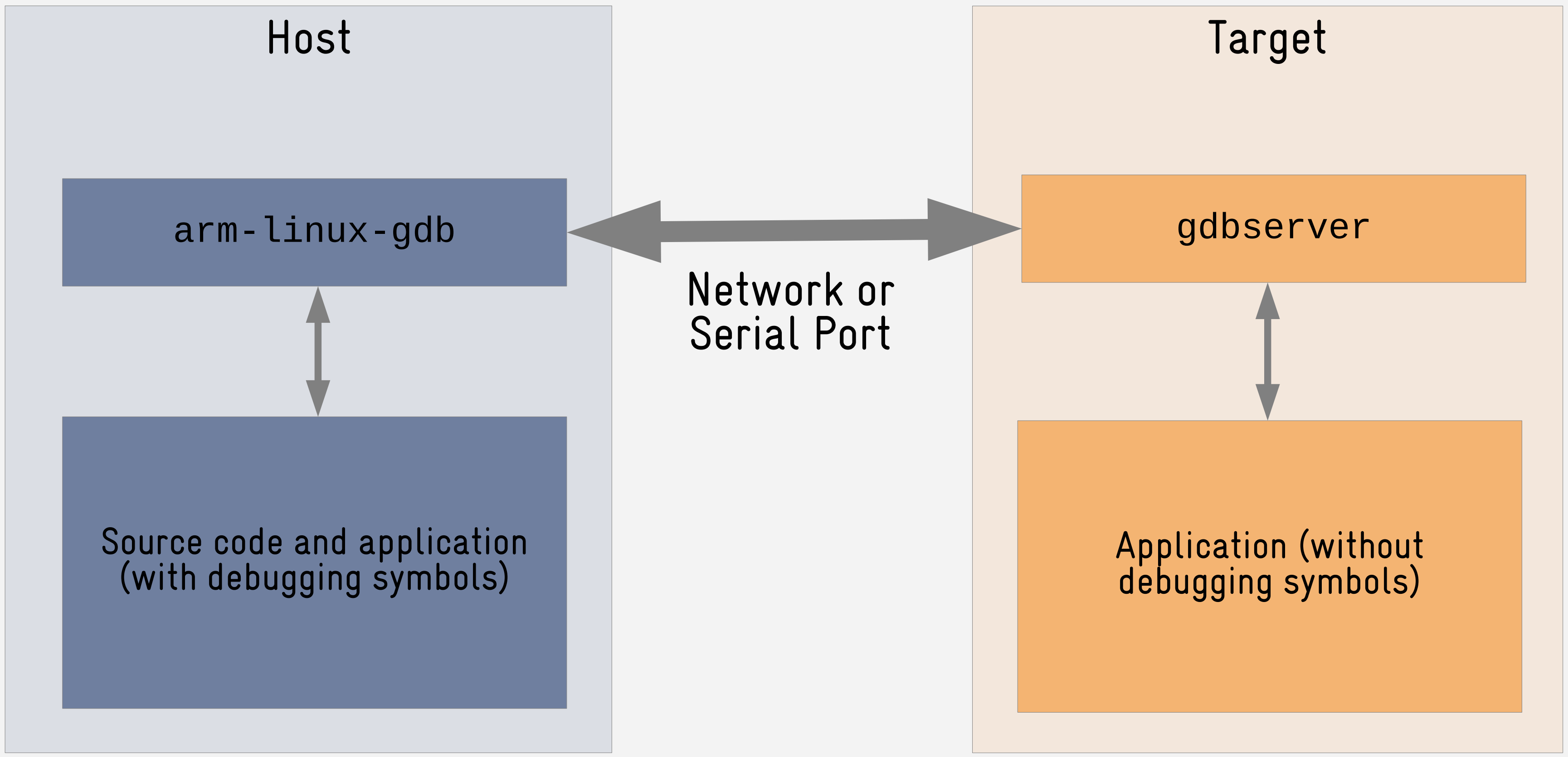
REMOTE DEBUGGING WITH GDB (1)

- GDB (GNU Project debugger) is the main choice for an interactive debugging tool on Linux.

  http://www.gnu.org/software/gdb/

- We can debug an application with GDB directly on the target (local debugging) or in the host machine (remote debugging).
  - Local debugging requires the binary with debug symbols, the source code and development tools (GDB) available on the target.
  - Remote debugging implies a client-server architecture, having a GDB server running on the target, and everything else (binary with debug symbols, source code, GDB client) in the host machine.

REMOTE DEBUGGING WITH GDB (2)


Host
arm-linux-gdb
Source code and application (with debugging symbols)
Network or Serial Port
Target
gdbserver
Application (without debugging symbols)

REMOTE DEBUGGING ON YOCTO (1)

- Let's say you want to debug an application called `myapp`.
- The first step is to recompile the application with optimization and the security flags disabled:

```
# add to local.conf
SECURITY_CFLAGS:pn-myapp = "${SECURITY_NOPIE_CFLAGS}"
DEBUG_BUILD:pn-myapp = "1"
```

- Now you can run GDB server on the target:

```
# gdbserver :3000 /usr/sbin/myapp
```

REMOTE DEBUGGING ON YOCTO (2)

- Next, setup the SDK environment and go to the sysroot directory:

```
$ source <SDK_DIR>/environment-setup-cortexa57-poky-linux
$ cd <SDK_DIR>/sysroots/cortexa57-poky-linux/
```

- To start remote debugging with GDB, you will need the **source code** of the application and the binary with **debugging symbols**.

- To make it easier for GDB to find the sources, you can set the application's source code directory in his configuration file:

```
$ echo "dir usr/src/debug/myapp/1.0-r0/myapp-1.0/src" >> ~/.gdbinit
```

- To search for the binary with debugging symbols, you can simply use `find`:

```
$ find . -name myapp 2>/dev/null | grep "\.debug"
./usr/sbin/.debug/myapp
```

REMOTE DEBUGGING ON YOCTO (3)

- Now you can run GDB to start the remote debugging session:

```
$ aarch64-poky-linux-gdb -tui ./usr/sbin/.debug/myapp
```

- Start the debugging session by adding a breakpoint in the `main` function:

```
(gdb) b main
```

- Then connect to the device and start debugging the application:

```
(gdb) target remote 192.168.7.2:3000
(gdb) continue
(gdb) next
(gdb) ...
```

HANDS ON 3: REMOTE DEBUGGING

- Run the command below in the target device (QEMU). The execution will fail with a `Segmentation Fault` error.

```
# fping -c 5 192.168.0.1
```

- Start a remote debugging session and run the application step-by-step to find the issue.

- Obs.: You don't have to fix the issue, just find the root cause.

Embedded Labworks

Linux debugging

Kernel oops analysis

KERNEL OOPS

- Kernel oops is a way for the Linux kernel to communicate the user that a certain error has occurred.

- When the kernel detects a problem, it kills any offending process and prints an oops message in the logs, including the current system status and a stack trace.

- Different kind of errors can generate a kernel oops, including an illegal memory access or the execution of invalid instructions.

- The official Linux kernel documentation about handling oops messages is available at `Documentation/admin-guide/bug-hunting.rst`.

KERNEL OOPS

```
[   35.427342] Unable to handle kernel NULL pointer dereference at virtual address 0000000000000000
[   35.427763] Mem abort info:
[   35.427916]   ESR = 0x96000045
[   35.428093]   EC = 0x25: DABT (current EL), IL = 32 bits
[   35.428308]   SET = 0, FnV = 0
[   35.428437]   EA = 0, S1PTW = 0
[   35.428531]   FSC = 0x05: level 1 translation fault
[   35.428754] Data abort info:
[   35.428889]   ISV = 0, ISS = 0x00000045
[   35.429024]   CM = 0, WnR = 1
[   35.429230] user pgtable: 4k pages, 39-bit VAs, pgdp=00000000438e1000
[   35.429470] [0000000000000000] pgd=0000000000000000, p4d=0000000000000000, pud=0000000000000000
[   35.429922] Internal error: Oops: 96000045 [#1] PREEMPT SMP
[   35.430262] Modules linked in:
[   35.430591] CPU: 1 PID: 341 Comm: ps Not tainted 5.14.9-yocto-standard #1
[   35.430941] Hardware name: linux,dummy-virt (DT)
[   35.431386] pstate: 80000005 (Nzcv daif -PAN -UAO -TCO BTYPE=--)
[   35.431614] pc : uptime_proc_show+0xfc/0x130
[   35.431806] lr : uptime_proc_show+0xa4/0x130
[   35.431972] sp : ffffffc0110ebc10
[   35.432086] x29: ffffffc0110ebc10 x28: 0000000000000001 x27: 0000000000400cc0
[   35.432404] x26: 000000007ffff000 x25: ffffff8003953038 x24: ffffffc010d588b8
[   35.432629] x23: ffffff8003953000 x22: 0000000000000004 x21: ffffffc010d58bc8
[   35.432876] x20: ffffffc010c51908 x19: 000000053af57000 x18: 0000000000000000
[   35.433117] x17: 0000000000000000 x16: 0000000000000000 x15: 0000000000000000
[   35.433375] x14: 0000000000000000 x13: 0000000000000000 x12: 0000000000000000
[   35.433616] x11: 0000000000000000 x10: 0000000000000000 x9 : 000000003b9aca00
[   35.433889] x8 : 0000000000000000 x7 : d6bf94d5e57a42bd x6 : 0000000012c921f5
[   35.434134] x5 : 000000001ba81400 x4 : 0000000000000016 x3 : 000000001664eed0
[   35.434378] x2 : 0000000000000023 x1 : ffffffc010b0c440 x0 : ffffff8003953000
[   35.434735] Call trace:
[   35.434830]  uptime_proc_show+0xfc/0x130
[   35.434949]  seq_read_iter+0x1b4/0x4b0
[   35.435096]  proc_reg_read_iter+0x98/0xe0
[   35.435318]  new_sync_read+0xd4/0x150
...
```

KERNEL OOPS ANALYSIS ON YOCTO (1)

- Kernel space debugging is not that different from userspace debugging, but may require specific kernel configuration options enabled and additional tools.

- In particular, changing the kernel configuration to enable debug options might be required.

- For kernel debugging, you will probably need to (re)compile the kernel with debugging symbols (`CONFIG_DEBUG_INFO`).

KERNEL OOPS ANALYSIS ON YOCTO (2)

- Enabling debug symbols can be done by creating a specific defconfig for your kernel or a kernel config fragment (in case you are leveraging `kernel-yocto.bbclass`):

```
$ bitbake virtual/kernel -c kernel_configme -f
$ bitbake virtual/kernel -c menuconfig
$ bitbake virtual/kernel -c diffconfig
```

- After the build, the kernel ELF image with debugging symbols (`vmlinux`) will be in the build directory (`B`). Then you can use it for debugging and symbol resolution.

KERNEL OOPS ANALYSIS ON YOCTO (3)

- Now you can open a new terminal and set up the development environment:

```
$ . <SDK_INSTALL_DIR>/environment-setup-cortexa57-poky-linux
```

- Then find the kernel build directory and move to it:

```
$ bitbake virtual/kernel -e | grep ^B=
$ cd <KERNEL_BUILD_DIR>
```

KERNEL OOPS ANALYSIS ON YOCTO (4)

- To make it easier for GDB to find the kernel sources, you can set the source code directory on its configuration file:

```
$ bitbake virtual/kernel -e | grep ^S=
$ echo "set substitute-path /usr/src/kernel <KERNEL_SOURCE_DIR>" >> ~/.gdbinit
```

- Now you can run GDB to find the line of code causing the oops message:

```
$ aarch64-poky-linux-gdb -tui vmlinux
...
(gdb) list *(uptime_proc_show+0xfc)
```

HANDS-ON 4: KERNEL OOPS ANALYSIS

- Try to list `/proc/uptime` and the kernel will emit an oops message:

```
$ cat /proc/uptime
```

- In the host machine, use GDB to find the line of code in the kernel sources that is causing the issue.
- Obs.: You don't have to fix the issue, just find the root cause.

Embedded Labworks

Linux debugging

Memory leak

VALGRIND

- Valgrind is an instrumentation framework for building dynamic analysis tools.

  http://valgrind.org/

- Valgrind tools can automatically detect threading and memory management related bugs.

- Valgrind emulates the application binary on a synthetic CPU, and for this reason, it significantly impacts application runtimes.

VALGRIND TOOLS

- `memcheck`: detects memory management issues (memory leaks, invalid memory access, uninitialized memory, etc).
- `helgrind`: identifies synchronization problems in multithreaded applications (race conditions, deadlocks, etc).
- `cachegrind`: profile cache usage in applications.
- `massif`: profile heap usage in applications.

MEMORY LEAK ANALYSIS ON YOCTO (1)

- Let's say you want to debug an application called `memleak` that is leaking memory.
- To debug it with Valgring, and in order for Valgrind to resolve symbols in the target device, it is necessary to install in the image the application's **debug** and **source** packages (also, disabling security flags is recommended):

```
IMAGE_INSTALL:append = " memleak memleak-dbg memleak-src valgrind valgrind-dbg"
SECURITY_CFLAGS:pn-memleak = "${SECURITY_NOPIE_CFLAGS}"
SECURITY_CFLAGS_pn-valgrind = "${SECURITY_NOPIE_CFLAGS}"
```

- Then you can just run the application with Valgrind on the target device.

Embedded Labworks

MEMORY LEAK ANALYSIS ON YOCTO (2)

```
$ valgrind --tool=memcheck --leak-check=full ./memleak
==8331== Memcheck, a memory error detector
==8331== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==8331== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==8331== Command: ./memleak
==8331==
==8331==
==8331== HEAP SUMMARY:
==8331==     in use at exit: 30 bytes in 1 blocks
==8331==   total heap usage: 1 allocs, 0 frees, 30 bytes allocated
==8331==
==8331== 30 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8331==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8331==    by 0x4006C9: txData (main.c:18)
==8331==    by 0x4007AE: main (main.c:38)
==8331==
==8331== LEAK SUMMARY:
==8331==    definitely lost: 30 bytes in 1 blocks
==8331==    indirectly lost: 0 bytes in 0 blocks
==8331==      possibly lost: 0 bytes in 0 blocks
==8331==    still reachable: 0 bytes in 0 blocks
==8331==         suppressed: 0 bytes in 0 blocks
==8331==
==8331== For counts of detected and suppressed errors, rerun with: -v
==8331== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

HANDS-ON 5: MEMORY LEAK ANALYSIS (1)

- For this exercise, you will need two connections to QEMU (the console and one additional connection via SSH).

- First, start QEMU with more memory so Valgrind can run properly.

```
$ runqemu nographic qemux86 qemuparams="-m 512"
```

- Then create another connection to QEMU via SSH:

```
$ ssh root@192.168.7.2
```

HANDS-ON 5: MEMORY LEAK ANALYSIS (2)

- In the SSH connection, run the following command to monitor memory usage:

```
# while true; do clear; free; sleep 1; done
```

- In the QEMU console, run `fping`:

```
# fping -l 192.168.7.1
```

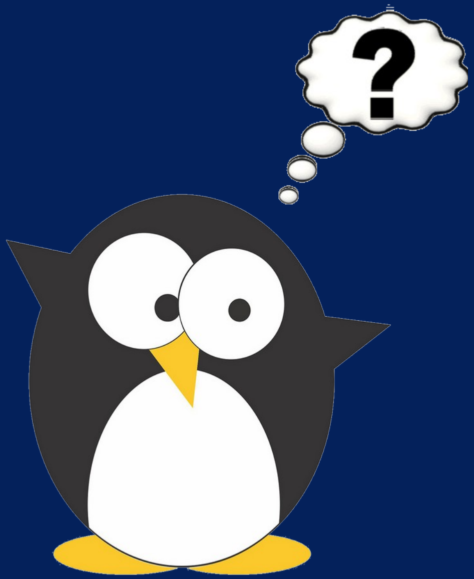
- Check the memory usage in the output of the SSH connection, and you will see that some memory is leaking during `fping` execution.

- Use Valgrind to find the line of code that is causing the leak.

Thanks!

E-mail	sergio.prado@e-labworks.com
Website	https://e-labworks.com/en
Linkedin	https://www.linkedin.com/in/sprado
Twitter	https://twitter.com/sergioprado
Blog	https://embeddedbits.org

Embedded Labworks



By Sergio Prado. São Paulo, September 2021