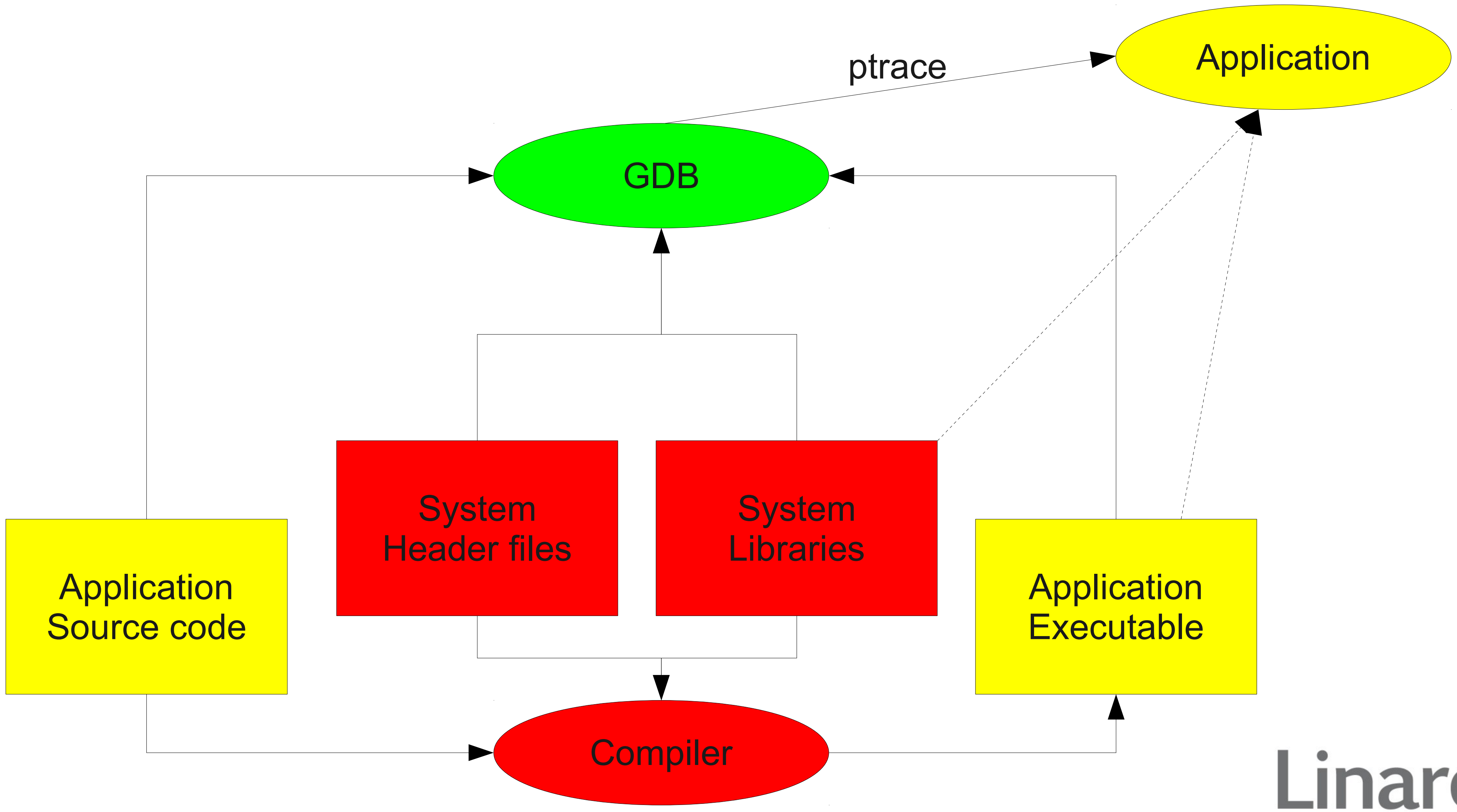
Debugging with GDB

Dr. Ulrich Weigand
<ulrich.weigand@linaro.org>
<ulrich.weigand@de.ibm.com>

Agenda

- Running a program under GDB
- Inspecting and modifying program state
- Automating debugging tasks
- Debugging optimized code
- Debugging multi-threaded programs
- Debugging several programs simultaneously
- Post-mortem debugging using core files
- Remote debugging
- Using tracepoints to debug non-intrusively

Running a program under GDB



Starting GDB

- Start up GDB to debug a program
  - “gdb executable”
  - “gdb –args executable arguments”
  - Or else just start “gdb” and then use commands
    - “file executable”, “set args arguments”
  - Then start up program via “run”
    - May want to set breakpoints first
- Attach GDB to an already running process
  - “gdb executable pid”
  - Or else just “gdb executable” and then “attach pid”
  - Process will be stopped wherever it happens to execute
  - Note: On Ubuntu, attaching is disabled by default. To enable:
    - echo 0 > /proc/sys/kernel/yama/ptrace_scope

Breakpoints

- “break location” will stop your program just before it executes any code associated with location.
  - A single source location may refer to multiple instruction start addresses (e.g. a function inlined into multiple callers). GDB will **automatically** set breakpoints on all those locations.
  - A single symbol name may refer to multiple source locations (e.g. a overloaded C++ function). GDB will **by default** set breakpoints on all those locations. This can by disabled via “set multiple-symbols ask”.
  - If location cannot currently be resolved, GDB asks whether to keep the breakpoint pending for future re-evaluation. This behavior can be modified via “set breakpoint pending on” or “set breakpoint pending off”.
  - All breakpoint locations are re-evaluated whenever a shared library is loaded or unloaded.
- “tbreak location” enables a breakpoint only for a single stop.
- “condition bnum expression” causes GDB to only stop at the breakpoint if the expression evalutes to non-zero.

Watchpoints

- “watch expression” will stop your program whenever the value of expression changes.
  - GDB will use hardware support to implement watchpoints efficiently if possible; otherwise GDB will continue silently single-stepping until the value of expression has changed.
  - The whole expression is constantly re-evaluated; for example “watch p->x” will trigger both if the value of the “x” member of structure “p” currently points changes, **and** if “p” is reassigned to point to another structure (if that structure's “x” member holds a different value).
  - Once a variable refered to by expression goes out of scope, the watchpoint is disabled.
  - Use “watch -location expression” to instead evaluate expression only once, determine its current address, and stop your program only if the value at this address changes.

ARM hardware watchpoints

- Feature set
  - Hardware watchpoints
    - Trap when a pre-defined memory locations is modified
    - Used to implement “watch” family of commands in GDB
  - Hardware breakpoints
    - Trap when execution reaches a specified address
    - Used to implement “hbreak” family of commands in GDB
    - Useful in particular to set breakpoints in non-modifyable code (e.g. ROM)
- Current status
  - Hardware breakpoint/watchpoint support added to Linux kernel 2.6.37
  - Support exploited by GDB 7.3
- Hardware pre-requisites
  - Cortex-A8: limited HW support, not currently exploited by Linux kernel
  - Cortex-A9: improved HW support, Linux kernel supports one single HW watchpoint
  - Cortex-A15: full HW support, Linux (3.2) supports multiple HW watchpoints

Catchpoints

- “catch throw” / “catch catch” will stop your program when a C++ exception is thrown or caught.
- “catch fork” / “catch vfork” / “catch exec” will stop your program when it forks or execs.
- “catch syscall [name]” will stop your program when it is about to perform a system call.
- “catch load [regexp]” / “catch unload [regexp]” will stop your program when a shared library is loaded or unloaded.
- Note that some of those commands may not be available on all platforms.

Continuing execution

- Continuing and stepping
  - “continue” resumes program execution.
  - “step” or “next” single-step to the next source line (stepping into/over function calls).
  - “finish” continues until the current function scope returns.
  - “until” continues until a location in the current function scope is reached (or it returns).
  - “advance” continues until a location is reached for the first time.
- Skipping over functions and files
  - “skip function” steps over any invocation of function, even when using “step”. (Useful for nested function calls.)
  - “skip filename” steps over all functions in the given file.

Inspecting program state

- Examining source files
  - “list” prints lines from a source file.
  - “search [regexp]” searches a source file.
  - “directory” specified directories to be searched for source files.
- Source and machine code
  - “info line linespec” shows which addresses correspond to a source line
  - “disassemble” shows machine code.
  - Use “set disassemble-next-line on” to automatically disassemble the current source line whenever GDB stops.

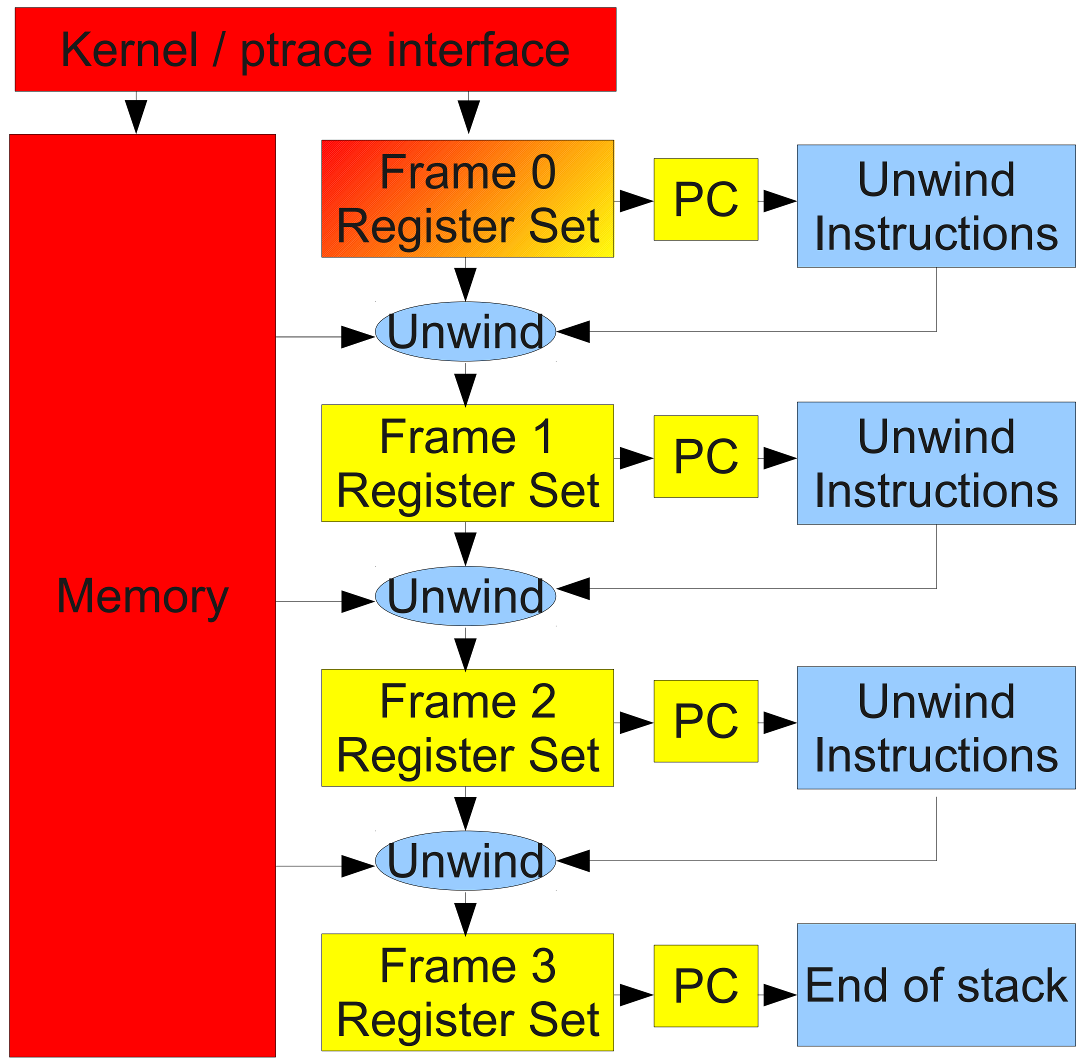
Inspecting program state

- Examining data
  - Use “print expression” to evaluate an expression in the source language and print its value.
  - Use “print/f expression” to use output format “f” to format the value (instead of its natural type).
    - Print as integer (various formats): “x”, “d”, “u”, “o”, “t”
    - Print as floating-point value: “f”
    - Print as address: “a”
    - Print as character or string: “c”, “s”
  - Use “x[/f] address” to print the value at address in the given format.
    - Additional format “i” to print disassembled machine instruction
  - Use “display[/f] expression” to automatically re-evaluate and print expression every time the program stops.
    - Always disassemble the current instruction via: display/i $pc

Inspecting program state

- Examining the stack
  - GDB will use current register values and memory contents to re-construct the “call stack” - the series of function invocations that led to the current location
  - “backtrace” show a backtrace of the entire call stack
  - “frame n” selects the n-th frame as “current” frame
  - “up [n]” / “down [n]” moves up or down the call stack
  - “info frame” describes the current frame
  - “info args” / “info locals” prints the function arguments and local variables for the current frame

Backtrace support: Background



- Basic algorithm
  - Start with initial register set (frame #0)
  - Extract PC from register set
  - Determine register unwind instructions at PC
    - "Restore PC from LR"
    - "Add 128 to SP"
    - "Restore R8 from memory at location (old) SP + 80"
    - "Register R10 is unchanged"
    - "Register R2 cannot be unwound; its prior value is lost"
  - Given old register set and memory contents, apply unwind instructions to construct register set at next frame (frame #1)
  - Repeat until uppermost frame is reached

Backtrace support on ARM

- How to determine unwind instructions at PC
  - Use DWARF-2 Call Frame Instructions (.debug_frame; on non-ARM also .eh_frame)
  - Use ARM exception table information (.ARM.exidx / .ARM.extbl)
  - Disassemble start of function containing PC and interpret prologue
  - Hard-coded special cases (e.g. signal return trampolines, kernel vector page stubs)
- Challenges on ARM
  - No .eh_frame section means no DWARF CFI in the absence of debug info
  - ARM exception tables were not supported in GDB
  - Glibc assembler code was not (always) annotated with ARM exception tables
  - Prologue parsing did not handle the Thumb-2 instruction set
    - Note that Thumb-2 is the default on current Ubuntu distributions
- Current status
  - Support for all missing features added
  - No GDB test case fails due to unwind problems
    - This is true even in the absence of system library debug info packages

Modifying program state

- Assignment to variables
  - Use “print var = expr” or “set variable var = expr” to store the value of expr into the program variable var
- Continuing at a different address
  - Use “jump linespec” or “jump *address” to continue execution elsewhere
- Returning from a function
  - Use “return” to cancel execution of the current function and immediately return to its caller
  - Use “return expr” to provide a return value
- Giving your program a signal
  - Use “signal { signr | signame }” to resume execution where your program stopped, but immediately deliver a signal

Calling program functions

- Use cases
  - Change program state (e.g. reset, initialize)
  - Pretty-print large data structures
  - Unit-test behavior of a single function
- Invocation
  - Use “print expr” or “call expr” where expr contains a function call
  - GDB will arrange for a stack frame to be allocated and argument values to be prepared, and then continue execution at the called function
  - Once execution completes normally, a temporary breakpoint will be hit; GDB resumes control and extracts the return value
- Caveats
  - If execution of the called function stops (due to a breakpoint, signal, or exception), the function invocation will remain on the stack
  - Use “set unwindonsignal on” and/or “set unwind-on-terminating-exception on” to have GDB instead unwind the stack

Automating debugging tasks

- GDB command language is powerful enough to “program” GDB to automate debugging tasks
- This includes features like:
  - Convenience variables
  - User-defined commands
  - Conditionals and loops
  - Command files
  - Breakpoint commands
  - Dynamic prints

Convenience variables

- Used to hold values that can be reused later
  - Exist only in GDB; their use does **not** affect the debuggee
- Related commands
  - Convenience variables idenfied by “$” prefix
  - Use “set $variable = expr” to set them
    - No predefined type; can hold values of any type
  - Use “show convenience” to display all variables
- Some predefined “magic” convenience variables
  - $_ / $__ automatically set to last address examined and its value
  - $_exitcode set to exit code when program terminates
  - $_siginfo set to signal info when program receives a signal

User-defined commands

- Named sequence of regular GDB commands
  - Useful e.g. to traverse long data structures
  - Up to 10 named arguments ($arg0 .. $arg9)
  - Can use convenience variables
  - Supports conditions and loops
    - if … else … end
    - while … loop_continue … loop_break … end

```
(gdb) define factorial
Type commands for definition
of “factorial”.
End with a line saying just
“end”.
>set $fact=1
>set $n=$arg0
>while $n>1
  >set $fact=$fact*$n
  >set $n=$n-1
  >end
>print $fact
>end

(gdb) factorial 5
$30 = 120
```

Other command sequences

- Command files
  - Format: sequence of GDB commands
  - Can be used via “source filename” command
- Automatically execute command file
  - gdb < filename
- Execute commands at GDB initialization
  - System-wide and private init command files (.gdbinit)
  - Command line options “-iex” or “-ix” execute single command or script **before** .gdbinit
  - Command line options “-ex” or “-x” execute single command or script **after** .gdbinit

Breakpoint actions

- Breakpoint commands
  - Command sequence associated with breakpoint
  - Use “commands [breakpoint-range] ... end”
  - Executed every time the breakpoint hits
  - May automatically continue execution
- Dynamic printf
  - Shortcut to combine a breakpoint with formatted printing of program data
  - Same effect as if you had inserted printf calls!
  - Use “dprintf location, template, expression, ...”
  - Example: dprintf test.c:25, “at line 25: glob=%d”, glob
    - Prints the value of “glob” every time line test.c:25 is hit
  - Default output to GDB console
    - May also call debuggee's printf (or related routine) instead

Debugging optimized code

- GDB **can** debug code compiled with optimizations enabled
  - However, the compiled code is no longer in 1:1 correspondence to your source code, which may cause various irritations
  - Still need to compile with -g to generate debug data
    - Note that use of -g **does not** disable any optimizations; generated code should be absolutely identical (with GCC)
    - Recent compilers generate much more detailed debug data to allow better debugging of optimized code
- Typical problems include
  - Assembler instruction sequence not in line with source code
  - Variables optimized away
  - Function inlining or tail-call optimizations

Variable locations

- What is the problem?
  - In optimized code, there is generally no fixed location (on the stack or in a register) where the current value of a local variable is stored
  - This makes it hard for a debugger to show the variable's value (or allow to modify it)
- How does GDB handle it?
  - Modern debug data allows GDB to find a variable at different locations throughout the execution of a function
  - If the variable is not stored **anywhere** debug data can still tell GDB how to synthesize the value it ought to hold at the current location
  - If none of this is the case, at least GDB should reliably recognize that the variable's value is currently not available
  - As a special case for function arguments: Even if the **current** value is unavailable, GDB may still be able to construct its “entry” value (i.e. the value the parameter held at the time the function was called) by inspecting the **caller's** stack frame
- Limitations include
  - Even with recent compilers, debug data still shows room for improvement ...

Function inlining

- What is the problem?
  - Instead of generating a call to a subroutine, the compiler places the subroutine body directly at its call site, subject to further optimizations
  - Source-level debugging shows lines of the subroutine intermixed with the caller
- How does GDB handle it?
  - GDB will pretend that the call site and the start of the inlined function are different instructions
  - Source-level stepping will first step onto the call site, and then to the first line of the inlined subroutine
  - Backtraces will show a synthesized frame representing the inlined subroutines including its arguments (if available)
- Limitations include
  - You cannot set a breakpoint on the call site; execution will stop showing the first line of the inlined subroutine instead
  - GDB is unable to show the “return value” when stepping out of an inlined subroutine using the “finish” command

Tail-call optimization

- What is the problem?
  - If the last action of a function B is a call to another routine C, the compiler may use a “jump” instruction instead of a “call” followed by “return”
  - If B was in turn called from function A, a backtrace will show A as C's caller; B is lost
- How does GDB handle it?
  - GDB tries to detect tail call situations by noticing that A has no call site calling C, but it does have a call site calling B, which in turn contains a **tail** call site invoking C
  - GDB will show a synthesized frame for B to represent the tail call situation
- Limitations include
  - In more complex situations, there is no unambigous call site path linking C to its (only possible) caller
  - In particular, tail **recursion** is always ambigous

Debugging multithreaded programs

- GDB will automatically detect all threads of the current process (in cooperation with the operating system and thread library)
  - On GNU/Linux, GDB will use
    - The libthread_db library to access state information held by the process' libpthread instance, to retrieve the initial thread list at attach time, as well as each threads' pthread_t value and thread-local storage block
    - The kernel's ptrace events mechanism to receive notification on thread creation and termination
- “info threads” shows existing thread
- “thread nr” selects current thread
  - Most GDB commands implicitly operate on this thread
- “thread apply nr|all command”
  - Invoke GDB command on another (set of) threads
- “break ... thread nr [if ...]”
  - Trigger breakpoint only while executing specified thread

Multithreaded execution: all-stop

- By default, GDB will stop **all** threads whenever the program stops
  - Advantage: While you work on the GDB prompt to inspect your program's state, it will not change due to actions of other threads
  - Disadvantage: Larger “intrusion” on the system, timing-sensitive operations may be disturbed more easily
- Conversely, whenever you restart your program **even just for single-stepping**, GDB will restart all threads
  - Advantage: No possibility of deadlocks introduced by GDB
  - Disadvantage: While single-stepping a thread, GDB may suddenly switch focus to another thread (e.g. because it hit a breakpoint)
- Use “set scheduler-locking on” to modify this behavior
  - GDB will then restart only the **current** thread, all others remain stopped
  - Use “set scheduler-locking step” to restart only current thread when using the “step” or “next” commands, but restart all threads when using “continue”

Multithreaded execution: non-stop

- In “non-stop” mode, you can freely select which threads should run and which should stop
  - When a thread runs into a breakpoint or some other stop condition, **only** this thread is halted
    - Note that it can still happen that more that one thread is stopped at the same time
  - Execution commands like “step”, “next”, or “continue” only restart the current thread
    - Note that “set scheduler-locking” has no effect in this mode
  - Use “continue -a” to restart **all** stopped threads
  - You may use “thread nr” to switch to a currently running thread, but execution control commands (and some others) will not work there
- Use “set non-stop on” to enable non-stop mode
  - Must be used **before** starting a program (or attaching to a program); has no effect on already started or attached programs

Background execution

- Use “set target async on” to enable asynchronous execution mode
- In asynchronous mode, GDB allows to run certain execution control commands “in the background”
  - To trigger background execution, use “&” at the end of the command line
  - GDB will then immediately provide another command prompt
  - Commands that support background execution:
    - Run, attach, step, stepi, next, nexti, continue, until, finish
- To interrupt a program executing in the background, use “interrupt”
  - In non-stop mode, “interrupt” only stops the current thread
    - Use “interrupt -a” to stop all threads
  - Note that Ctrl-C can be used to interrupt a program executing in the foreground
- Note that asynchronous mode and non-stop mode are independent
  - Still it is often useful to enable both simultaneously

Debugging several programs

- “Inferior”
  - Encapsulates state of execution of a program
  - Usually corresponds to a process
    - However, the inferior may be created before a process starts, and/or live on after a process has terminated
  - Most GDB commands (implicitly) operate on a designated “current” inferior
- Related GDB commands
  - “info inferiors” - list all inferiors managed by GDB
  - “inferior infno” - switch current inferior
  - “add-inferior [-exec filename]” - add new inferior
  - “clone-inferior” - create copy of current inferior
  - “remove inferior” - remove inferior (must have terminated)
    - May use “detach inferior infno” or kill inferior infno”

Multi-program execution

- All-stop mode
  - By default, only one inferior may be executing at any given time
    - “continue” etc. only affect the current inferior
  - Use “set schedule-multiple on” to allow multiple inferiors to be executing simultaneously
    - If any thread of any inferior stops, GDB will stop **all** threads of **all** inferiors
- Non-stop mode
  - Multiple inferiors may execute simultaneously
  - Threads of any inferior may be stopped or running independently of each other
  - In asynchronous mode, a new inferior may be started using “run &” while others are already running

Debugging across “fork”

- Debugging only one process
  - By default, GDB continues to debug parent process after it executed “fork”; child runs freely
  - Use “set follow-fork-mode child” to tell GDB to debug the child process; **parent** then runs freely
- Debugging **both** parent and child
  - Use “set detach-on-fork off” to keep GDB debugging both processes (in separate inferiors)
  - follow-fork-mode still determines whether parent or child is considered **current** inferior after fork
  - If multiple inferiors may run (either non-stop mode or all-stop mode with “set schedule-multiple on”), both parent and child will continue to run; otherwise only current inferior will do so

Debugging across “exec”

- By default, GDB will reset the current inferior to refer to the newly exec'd program
  - “run” will restart the **new** program, not the original
  - The symbol table of the old program is discarded and the symbol table of the new program is loaded
    - All breakpoints are re-evaluated against the new table
- Use “set follow-exec-mode new” as alternative
  - GDB will create a new inferior to track execution of the process after “exec”; this will behave as above
  - The original inferior remains unchanged
    - Use “run” in this context to restart the original program

Post-mortem debugging

- What is a core file?
  - Variant of the ELF file format
  - Contains a process' memory image
    - Unmodified memory-mapped file contents will be omitted
  - Also contains register contents
    - In multi-threaded applications, registers of all threads
- How does GDB use core files?
  - “gdb executable corefile” tells GDB to operate on a core file instead of a live process
  - GDB reads registers and memory from the core
  - All commands that examine debuggee state work just the same as when debugging a live process
  - Execution control (step, continue, ...) is disabled

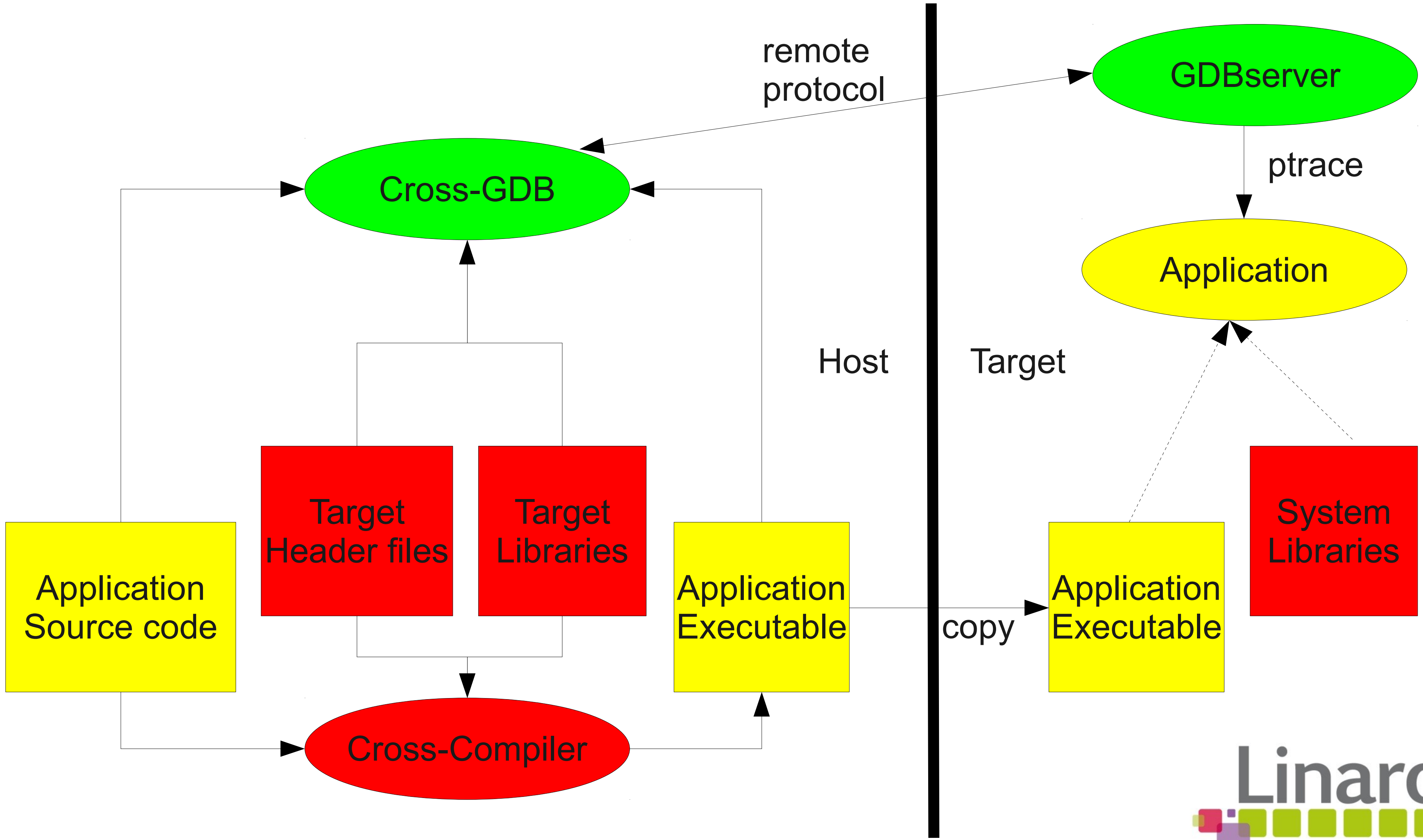
How to create core files

- When an application crashes
  - The kernel will automatically create a core file
  - Needs to be allowed via “ulimit -c unlimited”
  - File will be called “core” or “core.pid”
    - Can be configured via:
      - /proc/sys/kernel/core_pattern
      - /proc/sys/kernel/core_uses_pid
- On a live application
  - Use GDB to start the application or attach to it
  - Create core file via “generate-core-file filename”
  - Application will continue to run afterwards!
  - Utility “gcore” encapsulates this process

VFP/NEON register sets

- Floating-point / vector registers on ARM
  - Past architectures did not specify floating-point or vector registers; some implementations provided those via co-processor extensions
  - ARMv7 specifies VFPv3 and Advanced SIMD (“NEON”) extensions
    - VFPv3-D16: 16 64-bit registers / 32 32-bit registers
    - VFPv3-D32: 32 64-bit registers
    - NEON: VFPv3-D32 registers re-interpreted as 16 128-bit registers
- Current status
  - Access VFP/NEON register sets in native/remote debugging:
    Supported with Linux kernel 2.6.30 / GDB 7.0
  - Access VFP/NEON registers sets in core files:
    Supported with Linux kernel 3.0 / GDB 7.3

Remote debugging



Starting GDB for remote debugging

- On the target
  - "gdbserver :port executable arguments"
    - Starts up gdbserver and the program to be debugged (stopped at the first instruction)
  - "gdbserver –attach :port pid"
    - Starts up gdbserver and attaches to process PID (which will be stopped)
- On the host
  - Start via "gdb executable" as usual
  - Provide location of target libraries e.g. via "set sysroot"
  - Establish target connection via "target remote targethost:port"
    - GDB will now show program already running, but in stopped state
  - Start debugging e.g. via "break main" and "continue"
- Debugging multiple programs at the same time
  - Start up gdbserver in multi-program mode "gdbserver –multi :port"
  - On the host, connect to target via "target extended-remote targethost:port"
  - You can now start up a program as usual via "run" or "attach"

Remote debugging challenges

- GDB accesses application binary / target libraries on **host**
  - Assumes these are identical copies of files on target
    - Debugging will (silently) fail if that assumption is violated
  - Solution: Have gdbserver access files on **target**
    - Contents forwarded via remote protocol
  - Status: Implemented; enable via “set sysroot remote:”
- Native target and gdbserver target feature sets differ
  - Both implement similar functionality but do not share code
  - Some native features missing from remote debugging (and vice versa)
  - Long-term solution: Code re-factoring to allow re-use of identical code base
  - For now: Narrow gap by re-implementing missing gdbserver features
    - Support hardware break-/watchpoints
    - Disable address space randomization
    - Core file generation and “info proc”

Trace data collection

- Problem
  - Inspecting process state interactively takes time, while the process remains stopped
  - May cause trouble when debugging timing-sensitive applications
- Solution
  - Collect interesting pieces of process state on the fly, without disturbing timing much
  - Interactively inspect collected data later
- Basic mechanism
  - (Offline) Define tracepoints describing data to be collected
  - (Online) Run trace experiment, collecting data
  - (Offline) Find and inspect collected trace data
- Limitations
  - Only supported with remote debugging, and if supported by remote agent
  - Currently **not** supported by gdbserver on ARM (but on x86)

Defining tracepoints

- Use “trace” command to define tracepoints
  - Works just like “break” command
  - Tracepoints show up in breakpoint list, can be manipulated like breakpoints
  - Some targets support extra flavors
    - “ftrace” defines “fast” tracepoints
      - Collection performed in-process instead of via trap
    - “strace” enables “static” tracepoints
      - Pre-built into the target program (e.g. UST)
- Use “collect” command to define data to be collected
  - Used in tracepoint command list created via “actions”
    - Similar to breakpoint command list created via “commands”
  - “collect expr” collects value of expression
  - Special shortcuts “collect $regs”, “collect $args”, “collect $locals”
  - “while-stepping nr” single-steps several lines, repeatedly collecting data

Running trace experiements

- Use “tstart” to start trace experiment
- Use “tstop” to stop trace experiment
- Use “tstatus” to query status of experiment
- Use “set disconnected-tracing on” to keep trace experiment running even when GDB is disconnected from the target
- While the trace experiment runs, every time a tracepoint hits, a trace snapshot containing collected data is created

Using collected trace data

- Use “tfind ...” to focus GDB on a trace snapshot
  - As long as GDB is focused on a tracepoint, all GDB commands will behave as if we were currently debugging the program at the time the snapshot was taken
  - However, only such data as was collected will be available for inspection; GDB will report other data as “unavailable”
- Some variants of the tfind command include
  - “tfind n” - focus on n'th snapshot
  - “tfind tracepoint num” - next snapshot collected at tracepoint
  - “tfind linespec” - next snapshot collect at line/address
  - “tfind end” - leave trace snapshot mode

Questions?