LLDB 14 The New Debugger for FreeBSD

BY MICHAŁ GÓRNY AND KAMIL RYTAROWSKI

For the last decade, FreeBSD underwent a major effort of replacing the various components of its tool chain with more modern, permissively licensed programs. A part of that effort was replacing the aged GNU GDB debugger with LLDB, the debugger of LLVM project. Moritz Systems took part in that effort by taking up a few projects to modernize and improve the FreeBSD support in LLDB, as well as implement missing features.

LLDB 14 is a culmination of the work done so far. The March LLVM tool chain release includes a number of improvements that bring LLDB closer to a fully featured replacement for GDB. The debugger features FreeBSD support on amd64, arm, arm64, i386 and powerpc. It uses a client-server layout that provides uniform support for both local and remote debugging. In addition to the provided lldb-server, other protocol stubs are supported such as the ones provided by QEMU emulator or the FreeBSD kernel. Multithreaded programs are fully supported, as well as the most common multiprocessing scenarios, with more work underway. In addition to that, FreeBSD kernel debugging support akin to KGDB is provided.

This article details some of the more interesting aspects of LLDB's architecture and its features. However, prior to diving into the specific details, it probably makes sense to start by discussing some of the basic principles on how debuggers are implemented on Unix derivatives such as FreeBSD.

Debugging on Unix Derivatives

Debugging userspace processes on many Unix derivatives, including Linux, FreeBSD and other BSDs is implemented through a combination of ptrace(2) system call and signals. The former is generally used to control the traced process and obtain additional information about its state, while the latter is used to asynchronously report events to the debugger.

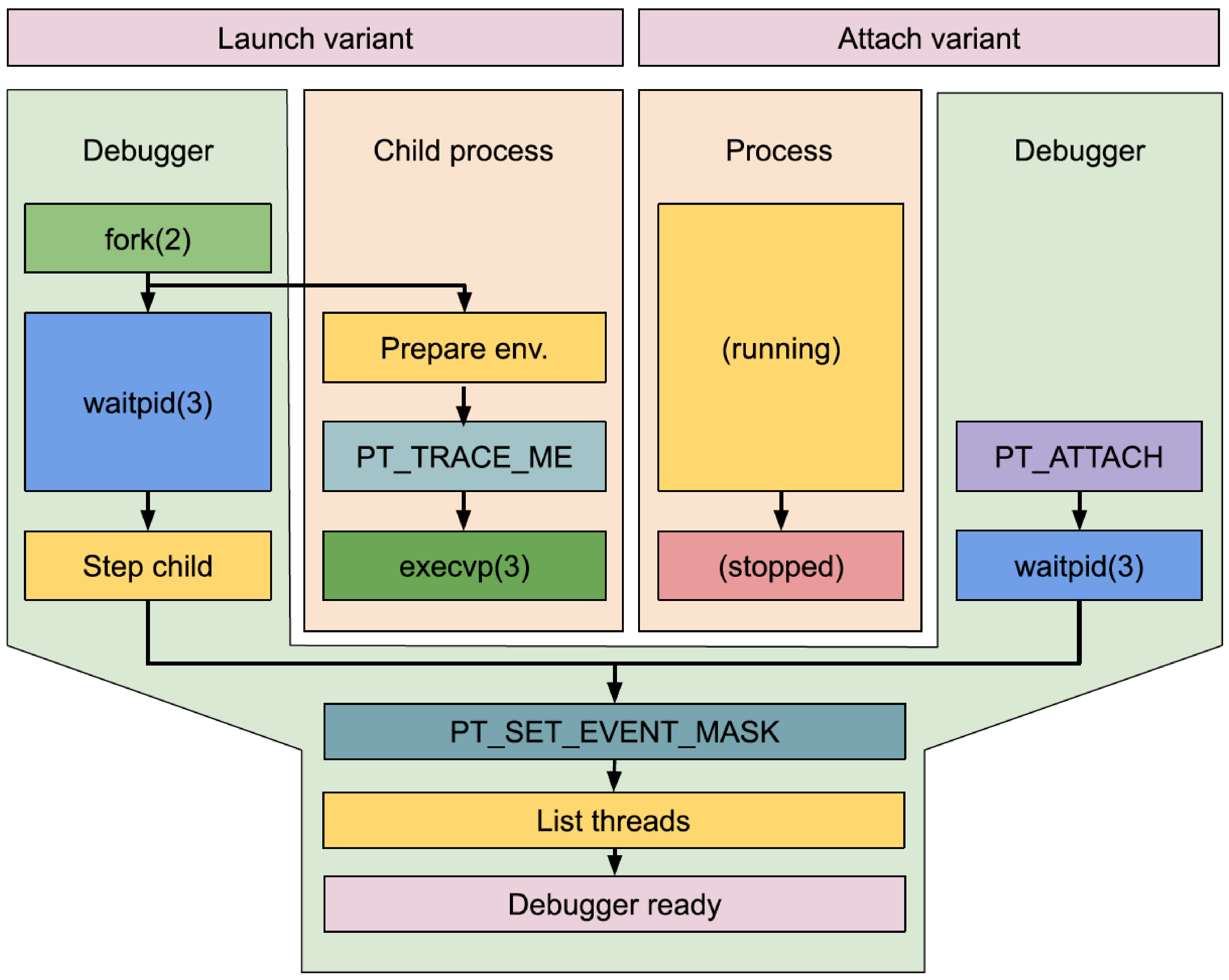

Fig. 1. The initial steps in a debugging session, initiated either via launching a new process or attaching to a running process.

The first step in a debugging session is for the tracer to either attach to a process that is already running, or launch a new program. In the former case, it issues a `PT_ATTACH` request. Once the request succeeds, the traced process is stopped and a signal is delivered to the debugger. In the latter case, the process is a bit more complex as presented on fig. 1.

The `ptrace(2)` API does not provide an explicit request to spawn a new program. Instead, the debugger needs to use the regular system API to do that, e.g. `fork(2) + exec*(2)`. However, just before executing the new program, the child process issues a `PT_TRACE_ME` request. This causes it to start being traced by its parent process (the debugger). At this point, the debugger steps the child process until the exec*(2) call finishes.

After attaching or launching, the traced process is stopped. The debugger performs additional setup, e.g. through setting the reported event mask or querying additional information about the debugged process. The remainder of the debugging session consists of the debugger issuing `ptrace(2)` requests to control the debugged process and query additional information about it, and the kernel delivering process-related events via signals. `SIGTRAP` has a special role here, as it is used to indicate the majority of events specific to the debugging process, e.g. breakpoint and watchpoint hits.

The Architecture of LLDB

LLDB utilizes plugins in order to abstract away large parts of its code base. At the moment of writing, there are 26 plugin categories existing in the LLDB source tree. The plugins provide support for different platforms, ABIs, programming languages, file formats and so on. While the plugin architecture is not considered complete yet (most notably, dynamic loading of plugins is not supported at the time of writing), it enforces the necessary encapsulation to prevent the code from becoming unmaintainable.

At the core of the plugin system, there is one category crucial to LLDB's debugging functionality: process plugins. These plugins implement all the routines needed to launch a process or attach to one already running, and debug it. In the modern versions of LLDB, the process plugin category holds two kinds of modules: the actual client plugins built on `Process` class, and lldb-server backends built on `NativeProcessProtocol` class.

Historically, every operating system supported by LLDB had its own client process plugin. Prior to LLVM 13, this was also the case for FreeBSD. When running on this platform, the LLDB client would load the respective process plugin and use it to control the debugged program. The debugger's UI and `ptrace(2)` invocations would both be done from a single process.

A more modern approach used by LLDB is to move the actual debugging process abstraction into the `lldb-server(1)` executable. The platform support is moved into a dedicated lldb-server backend. The client uses the `gdb-remote` plugin to either spawn a new lldb-server instance, or connect to another debug server using the GDB Remote Serial Protocol. This server does not have to be lldb-server — it could be the gdbserver from GDB or one of the implementations provided by e.g. QEMU, Valgrind or the FreeBSD kernel.

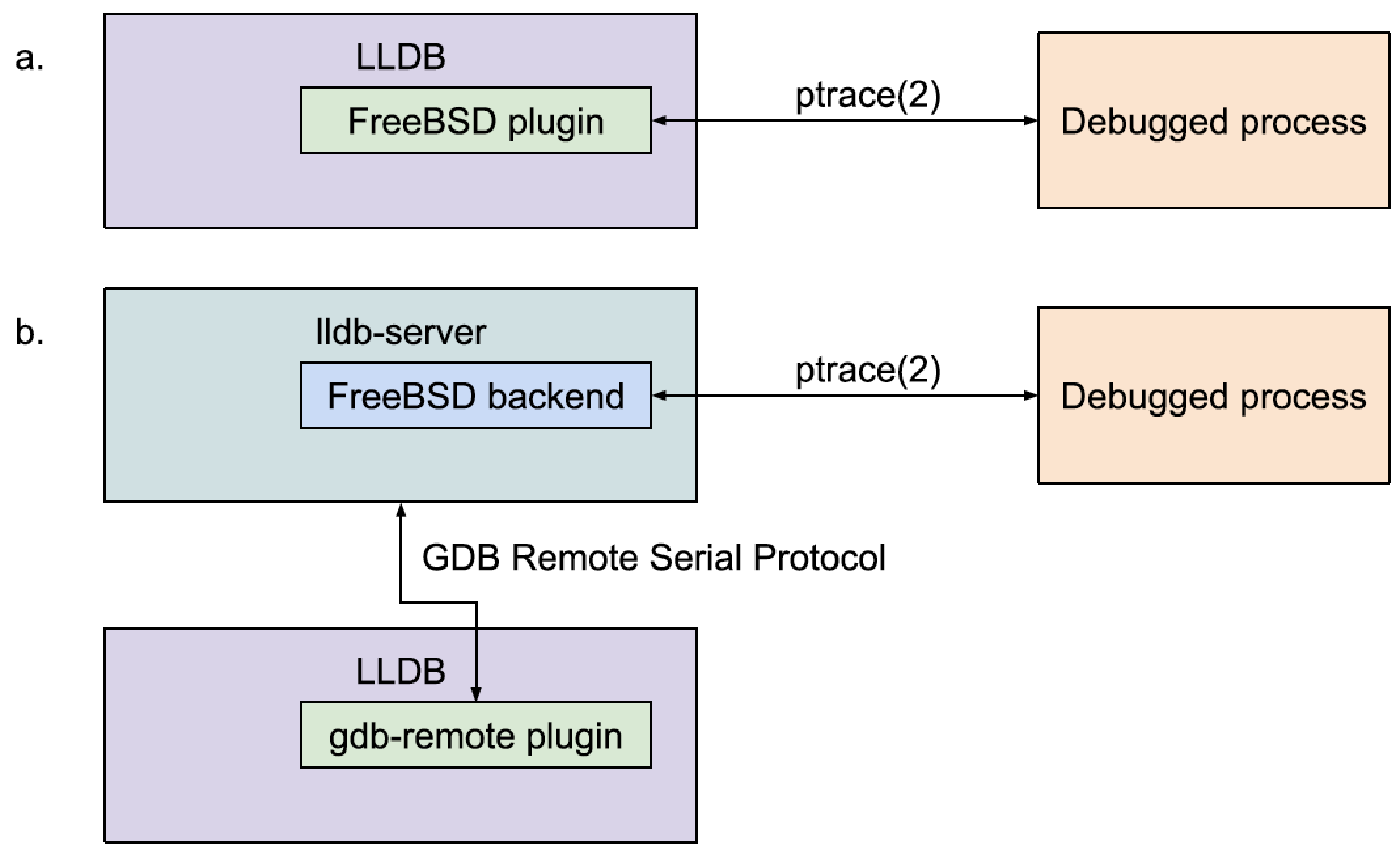

Fig. 2. Traditionally, the FreeBSD plugin was loaded into LLDB, and the debugged process was traced directly from the LLDB executable (subfig. a.). The more modern approach is to move this logic into lldb-server, and have LLDB communicate with it using the GDB Remote Serial Protocol (subfig. b.).

This change marks the evolution of LLDB from a standalone debugger to a client-server model that is capable of cross-platform remote debugging. This layout is also used when debugging locally, providing isolation between the debugger's UI (i.e. the LLDB client) and the server issuing the actual `ptrace(2)` calls.

As of LLDB 14, the vast majority of officially supported platforms use the client-server model and the `gdb-remote` process plugin. The other process plugins primarily implement support for a variety of core dumps formats.

Local and Remote Cross-platform Debugging

The LLVM toolchain is designed as a cross-compiler from ground up. This also applies to LLDB, as it features support for cross-debugging across different architectures and operating systems. However, since the majority of the debugging scenarios requires running the debugged program, LLDB needs to overcome the limitations of the operating system kernel.

A variety of operating systems feature the ability to run executables built for different ABIs that are compatible with the current CPU, most commonly 32-bit i386 executables on a 64-bit amd64 kernel. When this is the case, it is valuable for the kernel to support such programs via the `ptrace(2)` API and for a debugger to be able to use it correctly. The problem can be classified into three scenarios:

1. Native debugging — the kernel, the debugger and the debugged executable architectures are the same.

2. Debugging non-native programs — the kernel and the debugger architectures are the same but the executable architecture is different.
3. Running a non-native debugger — the kernel and the debugger architectures are different.

Table 1. Example mapping of architectures to the debugging scenarios.

Case	Kernel	Debugger	Executable
1	amd64	amd64	amd64
	i386	i386	i386
2	amd64	amd64	i386
3	amd64	i386	i386

The cases 1. and 3. are the same from the debugger's standpoint. Both the debugger and the traced executable are built for the same architecture. The debugger needs to explicitly feature support for the executables of this architecture, and its `ptrace(2)` API. Case 3. additionally requires the kernel support for non-native `ptrace(2)` API.

The second case is perhaps the most interesting. The debugger is built for the native system architecture, and therefore uses the native `ptrace(2)` API. However, this API needs to be adjusted for the non-native executable format. For example, if an i386 executable is run on amd64, the `PT_GETREGS` request uses 64-bit register dump format and it is desirable that the debugger translates between it and the format used natively on i386.

While local cross-debugging is limited by kernel features, remote debugging is much more powerful. In this scenario, lldb-server, gdbserver or any other server implementing a compatible protocol can be spawned on one machine, and the LLDB client can be used to connect to it from another. The two machines don't have to be running the same architecture or even the same operating system.

In fact, it gets even better. Remote debugging is not limited to tracing userspace applications. It can be used to connect to the GDB stub found in FreeBSD kernel over the serial port, and inspect the kernel's state. It can be used to connect to GDB stub implemented in QEMU in order to control the virtual machine's CPU and memory.

LLDB 14 features a more compatible implementation of the GDB Remote Serial Protocol than prior versions. Over the years, LLDB has evolved from using a custom variation of the protocol that was suitable only for communication between LLDB and lldb-server, to one that is compatible with many other debugging servers.

One interesting example of this evolution are register definitions. The earliest versions of lldb-server were transmitting them in JSON-based format that fitted the LLDB's internal layout best. Afterwards, XML-based format was added for compatibility with GDB. Finally, fallback definitions were added to the client for a variety of architectures in order to support stubs that did not transmit target definitions at all.

Debugging Multithreaded Processes

Every modern debugger needs to be able to handle multithreaded programs. In general, the support for multithreading consists of:

- receiving thread creation and termination events
- being able to distinguish other events applicable to a specific thread (e.g. per-thread signals, breakpoints)
- being able to control running and stopping individual threads

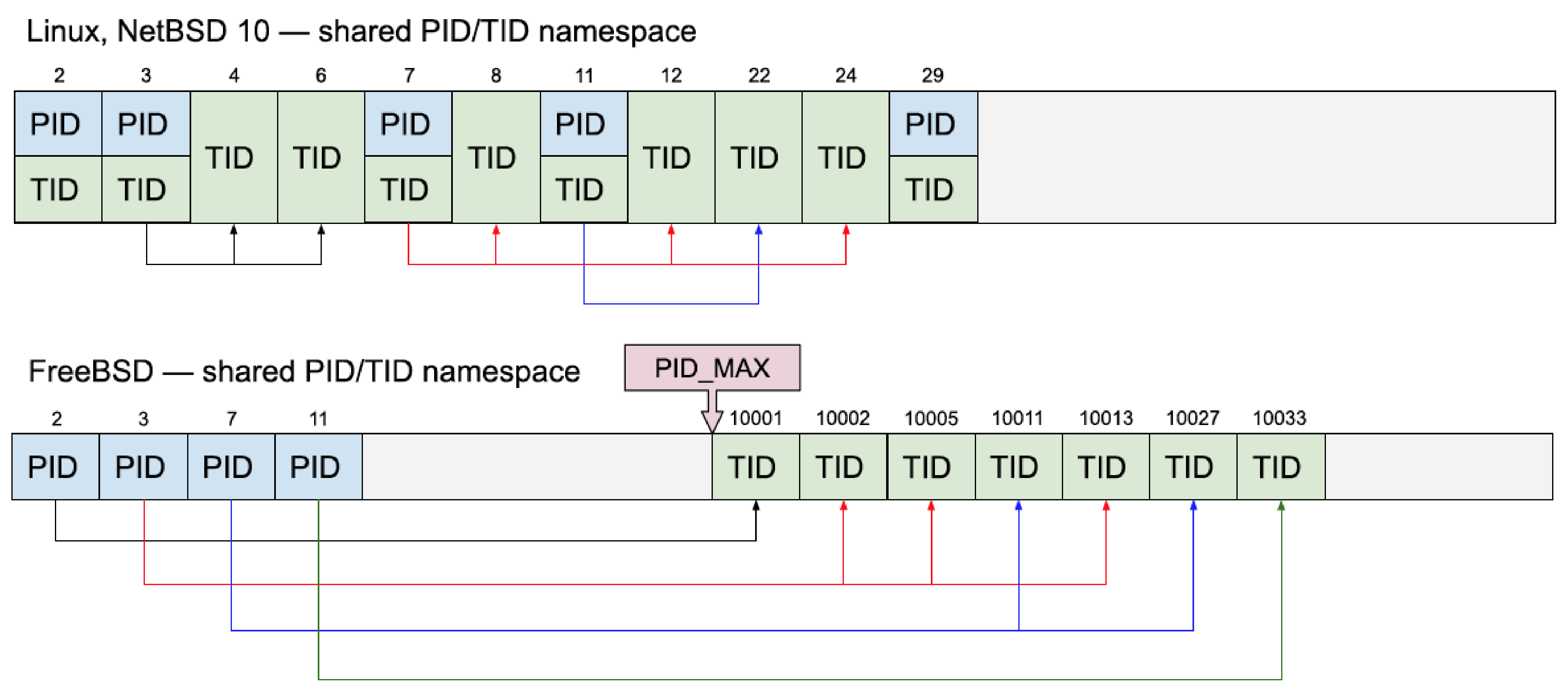
Linux, NetBSD 10 — shared PID/TID namespace
2 3 4 6 7 8 11 12 22 24 29
PID PID TID TID PID TID PID TID TID TID PID
TID TID TID TID TID
FreeBSD — shared PID/TID namespace
PID_MAX
2 3 7 11
PID PID PID PID
10001 10002 10005 10011 10013 10027 10033
TID TID TID TID TID TID TID

Fig. 3. On Linux and NetBSD 10, the primary thread shares identifier with the process identifier. On FreeBSD, process and thread identifiers use disjoint ranges.

The exact mechanism of handling multiple threads of a debugged program varies depending on the operating system. Modern kernels use a single namespace for process and thread identifiers. TIDs are globally unique. On Linux (and the future NetBSD 10 release) the primary thread has the same identifier as the corresponding process identifier. On FreeBSD process and threads identifiers use disjoint ranges.

This layout also implies how per-thread requests are handled via `ptrace(2)` API. On Linux and FreeBSD, TID can be passed in place of PID to perform a per-thread action. For historical reasons, on NetBSD the thread identifier needs to be passed separately along with the process identifier.

Thread list change events are generally enabled via setting an appropriate event mask. The kernel reports these events via issuing a `SIGTRAP` signal with appropriate data. However, it should be noted that the debugger needs to account for the events being received out of order, that is e.g. a breakpoint hit from a new thread arriving before the thread creation event.

On the current versions of FreeBSD and NetBSD, the tracing API could be called process scoped. When a thread-specific event occurs, the whole process is paused. The debugger receives a signal and needs to use `ptrace(2)` to obtain additional signal information, particularly the identifier of the corresponding thread. There are also requests to control whether a particular thread will remain paused, continue running or enter single-stepping when the process is resumed.

On the other hand, the Linux API treats threads in greater isolation. The debugger needs to trace every thread separately. Signals are reported for a specific thread. When a single thread stops, other threads of the process remain running.

Debugging Multiple Processes

The debugger's support for multiple processes can cover a variety of use cases, from programs forking themselves in order to run multiple operations simultaneously, to running external programs or complete pipelines. At the time of writing, LLDB has two features for debugging multiple processes: support for multiple targets and handling of fork events. Furthermore, there is an ongoing work to introduce full multiprocess support.

In LLDB terminology, a target represents a single debugged process. Appropriately, in order to be able to trace multiple processes simultaneously, LLDB needs to create and track multiple targets. In the most common case of using gdb-remote plugin to trace native processes, every process is traced by a separate lldb-server instance, and every target maintains a separate connection to its respective server. The limitation of this approach is that every target needs to be created separately, e.g. via launching the executable or attaching to a process that is already running.

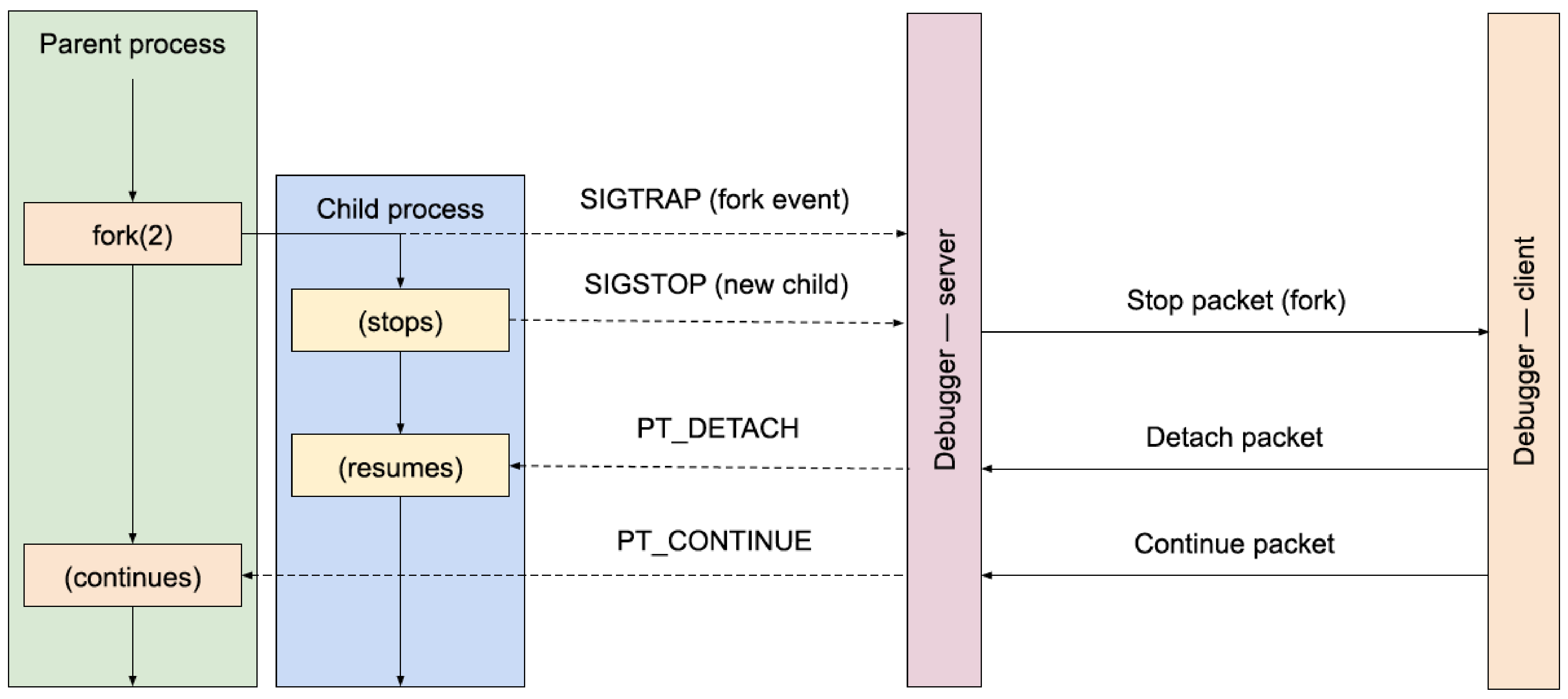

Fig. 4. When a process forks, the debugger receives `SIGTRAP` from the parent process and `SIGSTOP` from the child process. The server reports stop due to fork to the client, and the client requests detaching one of the processes (via `PT_DETACH`) and resuming the other one (via `PT_CONTINUE`).

The other feature is handling of `fork(2)`, `vfork(2)`, `posix_spawn(3)` and their equivalents. Similarly to the support for multiple threads, this is enabled via setting an appropriate event mask. When the process is forked, the debugger is signaled and starts tracing both the parent and the child processes. However, at this point LLDB does not feature full support for tracing both processes, and instead detaches one of them. LLDB can be configured to either continue tracing the parent process, or detach it and trace the newly forked child instead.

The purpose of the ongoing multiprocess effort is to combine both these features to provide the full support for debugging process trees. A key feature is to be able to start tracing the child process immediately without having to resume it or its parent. There are two primary possible implementations: using GDB-style multiprocess extensions to support multiplexing multiple traced processes within a single GDB Remote Serial Protocol connection, or using a separate connection for every new process.

Non-live Process Debugging Targets

While admittedly the primary use of LLDB on FreeBSD is to debug userspace processes, there are other kind of 'process' plugins: notably plugins handling core dumps and the FreeBSD kernel debugging. Similarly to the gdb-remote plugin, these modules are loaded directly into LLDB client and do not utilize the client-server architecture.

Core dumps on modern Unix derivatives are recorded using the ELF file format, much like executables on these platforms. Appropriately, they are handled by an elf-core plugin in LLDB. However, this plugin is not suitable for handling FreeBSD kernel core dumps since these dumps use physical memory layout rather than the virtual layout used by regular processes.

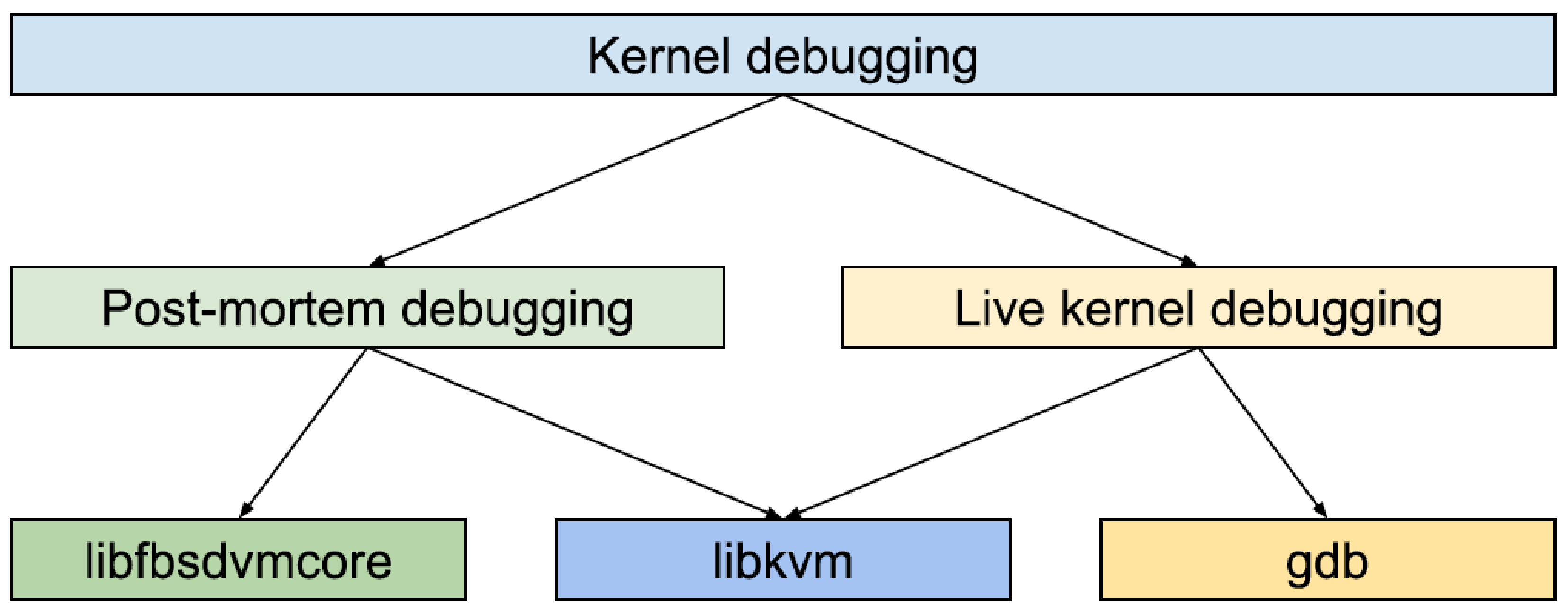

Fig. 5. Common methods of debugging the FreeBSD kernel

FreeBSD kernel debugging is a wider topic. There is a variety of methods provided for working either with the live kernel (i.e. the kernel of a running system) or with a kernel core dump. Firstly, the kernel itself features a built-in debugger, `kdb`. Secondly, it features a GDB stub that can be used to attach a remote debugger using the GDB Remote Serial Protocol, over a serial port. The recent versions of LLDB include improved serial port support for precisely this reason. Thirdly, it is possible to gain access to kernel memory through `/dev/mem` special device. Finally, it is possible to use a kernel core dump.

LLDB 14 introduced a new `FreeBSDKernel` plugin that provides support for the last two scenarios. It enables LLDB to open and process kernel core dump files correctly, both in the newer minidump format and in the older 'full memory dump' format. This is done with the help of one of the two libraries: either libkvm that is provided by the FreeBSD base system, or libfbsdvmcore that has been created as a portable alternative for cross-platform debugging. This makes it possible to use LLDB as a replacement for the KGDB tool.

Summary

Over the last years, LLDB has made major progress as a debugger. While it still does not feature 100% feature parity with GDB, it improves with every new release. Furthermore, it goes beyond aiming to be 'just' a replacement for GDB. Built on top of LLVM tool chain, it features extensive expression parsing support built on top of Clang, JIT suport, Python and Lua scripting support. Plugin-based design and good test coverage makes extending it a pleasure.

The progress made on the LLDB front has made it possible to finally retire the aged GDB from FreeBSD and replace it with LLDB that fits the project much better. Furthermore, thanks to fruitful cooperation between the projects, it was possible to integrate KGDB functionality directly into LLDB, removing the need for a separately maintained frontend. These achievements were possible in large part due to the support of the FreeBSD Foundation.

There is a lot of interesting work happening in LLDB all the time. A part of it is a recently started project on implementing full support for multiprocessing — effectively enabling users to efficiently debug complete trees of forked or spawned processes. Another interesting future project is RISC-V architecture support.

KAMIL RYTAROWSKI and **MICHAŁ GÓRNY** are open source enthusiasts and contributors for over 10 years. They founded Moritz Systems, a company focused on BSD development. They authored the modern LLDB plugin for NetBSD, and have been improving LLDB's support for FreeBSD since September 2020.