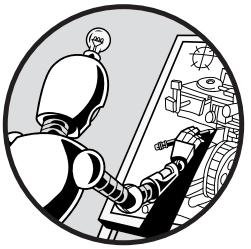
3

ATTACHING TO A PROCESS

We are all of us
tethered to our guide,
finding somewhere to begin.



What is a debugger without a process to debug? In this chapter, you'll start writing your debugger by creating a program that can attach itself to other processes, either ones that are already running or ones that it launches itself. Users will be able to interact with the debugger through a simple command line interface.

You'll put into practice some of the operating system fundamentals we talked about in Chapter 2 and get your first taste of the `ptrace` system call. You'll also begin to structure your debugger in a way that you can test more easily and use as a library rather than just on the command line.

Process Interaction

Before you write any code, you need to understand the facilities that Linux provides to spawn new processes and trace their execution.

fork and exec

On Linux systems, new processes are spawned using the `fork` and `exec` syscalls. The `fork` syscall divides the running process into two separate processes that are identical save for the return value of `fork` itself; the new, or *child*, process returns `0`, whereas the original, or *parent*, process returns the *process identifier (PID)* of the child. Every process running on your system has a unique PID that you can use in many syscalls to indicate the process on which to operate.

After forking, the child and parent are free to walk their own paths in life. The child process may choose to do something completely different from the parent by executing a different program. It can do so using the `exec*` family of syscalls, which replace the currently executing program with a new one.

Because all spawned tasks follow this common algorithm, no process is born out of nothing on Linux; the links between parents and children form a sort of family tree. The topmost process belongs to whichever tool your flavor of Linux uses as its initialization system, likely `init` or `systemd`, and has the PID `1`. You can visualize this tree using tools like `ps`. Here is some of the output of `ps ax --forest` on my WSL system:

```
  PID TTY      STAT   TIME COMMAND
    1 ?        Sl     0:00 /init
12427 ?        Ss     0:00 /init
12428 ?        S      0:00  \_ /init
12429 pts/0    Ss     0:00      \_ -bash
17351 pts/0    R+     0:00          \_ ps ax --forest
16972 ?        Ss     0:00 /init
16973 ?        S      0:00  \_ /init
16974 pts/2    Ss+    0:00      \_ -bash
```

Note that `ps ax --forest` is not a process born out of the ether; it was spawned by my bash shell, which was in turn spawned by `init`, which was itself spawned by a different `init` process.

ptrace

As mentioned in Chapter 2, `ptrace` is the main debugging interface provided by Linux, so you'll spend a lot of time with it while building your fully fledged debugger. This interface gives you a myriad of tools with which to communicate with a different process. We often refer to such a process as the *inferior process*, or simply the *inferior*.

Unfortunately, as it was first shipped in 1975 with V6 Unix, `ptrace` isn't exactly a shining example of modern API design. The Unix philosophy "do one thing and do it well" seems to have skipped over this particular function, which does rather a lot of things. (Generally 36 things, although its features may vary depending on the Linux kernel version you're using.)

The ptrace interface lives in the <*sys/ptrace.h*> header and looks like this:

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

The request parameter indicates the action you would like to perform, which could be anything from reading memory to setting up a process to be traced or sending a SIGKILL. The pid parameter is the PID of the process you'd like to operate on. The addr and data parameters vary in meaning depending on the value you pass for request.

You can examine the tool's manual page for an exhaustive list of available commands, as there are too many to reasonably list here. I'll introduce you to certain commands when you need them, but here are a few examples to give you a taste of what is available:

PTRACE_PEEKDATA Reads 8 bytes of memory at the given address

PTRACE_ATTACH Attaches to the existing process with a given PID

PTRACE_GETREGS Retrieves the current values of all CPU registers

PTRACE_CONT Continues the execution of a process that is currently halted

The return value from ptrace depends on the request, but generally speaking, it returns -1 and sets errno when an error occurs. (This is an example of poor API design; return values whose meanings differ based on the arguments supplied are confusing and difficult to handle correctly.)

While it's tempting to ignore the return value, you should always check if the call to ptrace returned -1 and, if so, report this error to the user. This could save you hours spent tracking down a heisenbug because you forgot to check for ptrace errors. Drink some water. Get enough sleep. Check your return codes.

Launching and Attaching to Processes

With the background out of the way, let's launch a program. We'll support two ways to attach the debugger to a process:

- Launching a named program ourselves and attaching to it by running sdb *<program name>*
- Attaching to an existing process by running sdb -p *<pid>*

To keep the focus on the process interaction code, our command line argument handling will be very basic.

The main Function

Begin by writing the main function in *sdb/tools/sdb.cpp*. A common pattern we'll use in this book is assuming that some function already exists, writing code that uses it, and then implementing that function. This workflow is called *top-down programming*.

Let's assume we have a function called `attach` that launches, attaches to the given program name or PID, and returns the PID of the inferior:

```
#include <iostream>
#include <unistd.h>

❶ namespace {
    pid_t attach(int argc, const char** argv);
}

int main(int argc, const char** argv) {
 ❷ if (argc == 1) {
        std::cerr << "No arguments given\n";
        return -1;
    }

    pid_t pid = attach(argc, argv);
}
```

We'll make a habit of putting symbols inside anonymous namespaces if they're used only in the implementation file in which they're defined. This practice will avoid name collisions if we happen to reuse a name across different files. In this case, for example, we put the `attach` function in an anonymous namespace ❶. Its return type, `pid_t`, is an integral type for storing a process ID.

In `main`, we call `attach` with the command line arguments provided to `sdb`. The first command line argument of a C++ program is always the path to the running executable itself, so the user should supply at least two arguments to specify the program to launch or the PID to attach to. If they've supplied only one argument ❷, we throw an error.

The attach Function

Now write the `attach` function in *sdb.cpp*. Although it's best practice to place this code in `libsdb`, we'll first implement the basic launching and attaching functionality in a single file, then refactor it at the end of the chapter to work in a decent error-handling story.

A high-level view of the structure of `attach` looks like this:

```
#include <string_view>
#include <sys/ptrace.h>

namespace {
    pid_t attach(int argc, const char** argv) {
        pid_t pid = 0;
        // Passing PID
     ❶ if (argc == 3 && argv[1] == std::string_view("-p")) {

        }
```

```
        // Passing program name
        else {

        }

    ❷ return pid;
    }
}
```

In `attach`, we check if the first command line argument passed was `-p` ❶. Note that simply writing `argv[1] == "-p"` would have done the wrong thing, as it would have compared only the pointer values rather than the string contents. Instead, we use C++17's `std::string_view` to avoid relying on old crusty C functions or dynamically allocating memory with `std::string`. Next, we define two branches inside the function, leaving them blank for now. Then, we return `pid` ❷, whose value we'll set inside of those blocks.

Let's implement the block that attaches to an existing process. For this, we use the `PTRACE_ATTACH` request:

```
--snip--
// Passing PID
if (argc == 3 && argv[1] == std::string_view("-p")) {
    pid = std::atoi(argv[2]);
    if (pid <= 0) {
        std::cerr << "Invalid pid\n";
        return -1;
    }
❶ if (ptrace(PTRACE_ATTACH, pid, /*addr=*/nullptr, /*data=*/nullptr) < 0) {
   ❷ std::perror("Could not attach");
        return -1;
    }
}
--snip--
```

We attach to the process by passing `PTRACE_ATTACH` and the process's ID to `ptrace` ❶. As a result, Linux will allow us to send other `ptrace` requests to this process. It will also send the process a `SIGSTOP` to pause its execution. Because we're good citizens, we check if `ptrace` returned an error. Then, we pass `nullptr` as the `addr` and `data` arguments, as they're unused in the `PTRACE_ATTACH` request.

If an error occurs, `ptrace` additionally sets the `errno` variable with an error code describing what went wrong. We use `std::perror` ❷ to print this description to `stderr`, along with a string that we pass to provide more context to the user.

Next, let's implement the launch-and-attach functionality using `fork` and `exec`:

```
--snip--
// Passing program name
else {
    const char* program_path = argv[1];
 ❶ if ((pid = fork()) < 0) {
        std::perror("fork failed");
        return -1;
    }

    if (pid == 0) {
        // In child process
        // Execute debugee
    }
}
--snip--
```

We call `fork` and then ensure that no error occurred ❶. Recall that `fork` returns `0` inside the child process, so we test for this case.

If we're in the child process, we should replace the currently executing program with the program we want to debug. However, before we call `exec`, we must set the process up to be traced using the `PTRACE_TRACEME` request, which will allow us to send more `ptrace` requests to this process in the future:

```
    --snip--
    if (pid == 0) {
        if (ptrace(PTRACE_TRACEME, 0, nullptr, nullptr) < 0) {
            std::perror("Tracing failed");
            return -1;
        }
     ❶ if (execlp(program_path, program_path, nullptr) < 0) {
            std::perror("Exec failed");
            return -1;
        }
    }
    --snip--
```

After enabling process tracing, we call `execlp` ❶, which is one of the flavors of `exec` I mentioned earlier in this chapter. The `l` in `execlp` means that arguments passed to the program should be supplied individually rather than as an array. The `p` means that the facility will search the `PATH` environment variable for the given program name if the supplied path doesn't contain a forward slash (`/`). Here is the signature for `execlp`:

```
int execlp(const char *file, const char *arg, ...);
```

The ... at the end of the argument list means that this function takes a variable number of arguments. Such functions are called *varargs* functions.

After we've attached to the process, we should wait until it has paused execution before we accept any user input. Linux helps us here by stopping the process on a call to `exec` if it's being traced using `ptrace`. We wait for this stop to occur using the `waitpid` function, whose signature is as follows:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

We can pass `waitpid` the PID of a child process to wait until a state change, which occurs when the child is either terminated or stopped by a signal. If the process has already changed state, the function will return immediately; otherwise, the parent process will block until a change occurs.

The `status` output parameter can give us information about the state change that happened. We can call various macros on the returned status to check its properties, such as `WIFSIGNALED(status)`, which checks if the child was terminated by a signal. Consult the manual page for `waitpid` for the complete list of macros.

The `options` parameter allows us to pass various flags to tune the wait, such as `WCONTINUED` to be notified if a `SIGCONT` resumes the child process. For now, we'll simply pass the PID and ignore the other parameters. Extend the `main` function as follows:

```
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, const char** argv) {
    if (argc == 1) {
        std::cerr << "No arguments given\n";
        return -1;
    }

    pid_t pid = attach(argc, argv);

    int wait_status;
    int options = 0;
 ❶ if (waitpid(pid, &wait_status, options) < 0) {
        std::perror("waitpid failed");
    }
}
```

We wait for the process to stop after we attach to it ❶. Now that we've attached to a process (which we may have also launched), we can start reading commands from the user.

Adding a User Interface

We want the user to interact with the debugger through a command line interface. We'll provide such an interface using `libedit`, which lets us

implement history, searching, and the kind of navigation you expect from a console-based debugger. (In fact, the LLDB debugger uses `libedit` as well.) Add basic support for `libedit` in *sdb/tools/sdb.cpp* (we'll shortly replace this code with something more complex; this is just to give you a feel for the process):

```
#include <editline/readline.h>
#include <string>

namespace {
    void handle_command(pid_t pid, std::string_view line);
}

int main(int argc, const char** argv) {
    --snip--
    char* line = nullptr;
 ❶ while ((line = readline("sdb> ")) != nullptr) {
     ❷ handle_command(pid, line);
     ❸ add_history(line);
     ❹ free(line);
    }
}
```

We loop in `main`, reading input from the user until there is nothing left to read ❶. Within the loop, the `readline` function from `libedit` takes a prompt and returns a `char*` representing the line it reads from the user. If it reads an end-of-file (EOF) marker (because the user has entered CTRL-D), it returns `nullptr`.

Next, we call a currently nonexistent function that handles the command ❷. We add the command to the searchable history using `add_history` ❸, which `libedit` provides. Finally, we clean up the memory that `readline` allocated for the line ❹; no one likes memory leaks.

Now we'll go one step further: if the user enters an empty line, we treat this as a shortcut to rerun the last command. Replace the code you just wrote with this:

```
    --snip--
    char* line = nullptr;
    while ((line = readline("sdb> ")) != nullptr) {
     ❶ std::string line_str;

     ❷ if (line == std::string_view("")) {
            free(line);
            if (history_length > 0) {
             ❸ line_str = history_list()[history_length - 1]->line;
            }
       }
```

```
    ❹ else {
          line_str = line;
          add_history(line);
          free(line);
      }

    ❺ if (!line_str.empty()) {
          handle_command(pid, line_str);
      }
  }
```

We add the `std::string` local variable ❶ for holding the command to be executed, regardless of whether it came straight from the user or from the `readline` history.

Next, we check whether the line is empty ❷. If so, we free the memory for it before trying to retrieve the last item in the `readline` history. `libedit` provides a `history_list` function for retrieving the history and a `history_length` global variable that tells us how many entries there are. Using these, we find the most recent line input by the user ❸.

If the line wasn't empty ❹, we save its contents into `line_str`, add it to our history, and free its memory. Finally, we handle the command if we received one ❺.

Handling User Input

Now that we can retrieve textual commands from the user, we need to interpret them and carry out the requested action. Our commands will follow a format similar to that of the GDB and LLDB debuggers. To continue the program, a user can enter `continue`, `cont`, or even just `c`. If they want to set a breakpoint on an address, they'll enter `break set` *`0xcafecafe`*, where *`0xcafecafe`* is the desired address in hexadecimal format.

Add support for continuing the program in *sdb/tools/sdb.cpp*:

```
#include <vector>

❶ namespace {
    std::vector<std::string> split(std::string_view str, char delimiter);
    bool is_prefix(std::string_view str, std::string_view of);
    void resume(pid_t pid);
    void wait_on_signal(pid_t pid);

  ❷ void handle_command(pid_t pid, std::string_view line) {
        auto args = split(line, ' ');
        auto command = args[0];
```

```
        if (is_prefix(command, "continue")) {
            resume(pid);
            wait_on_signal(pid);
        }
        else {
            std::cerr << "Unknown command\n";
        }
    }
}
```

We declare several functions in an anonymous namespace to carry out string handling and process manipulation tasks ❶. We'll implement these shortly. Note that we can't call the `resume` function `continue` because `continue` is a keyword in C++.

We implement a simple command handler ❷ by splitting the command on spaces in case the user provided arguments to the command. If the command is a prefix of `continue`, we continue the process and then wait for it to halt. If we don't recognize the command, we print an error message for the user.

AUTO

The command-handling function uses `auto`, a feature added in C++11 that specifies that the type of a variable should be deduced from its initializer. So, if we write `auto i = 0;`, then `i` will be of type `int`.

Importantly, `auto` doesn't deduce references. If your function returns a reference like `int& get();` and binds it to a variable like `auto i = var.get();`, then `i` will be an `int`, not an `int&`. In other words, it will be a copy of the return, not a reference to it. You need to explicitly ask for references with code like `auto& i = var.get();`.

I'll use `auto` throughout this book to make the code shorter or to save us from having to think about a variable's type if it's not very important.

Next, fill in `split` and `is_prefix`, a couple of small string manipulation helpers:

```
#include <algorithm>
#include <sstream>

namespace {
❶ std::vector<std::string> split(std::string_view str, char delimiter)  {
        std::vector<std::string> out{};
        std::stringstream ss {std::string{str}};
        std::string item;
```

```
        while (std::getline(ss, item, delimiter)) {
            out.push_back(item);
        }

        return out;
    }

❷ bool is_prefix(std::string_view str, std::string_view of) {
        if (str.size() > of.size()) return false;
        return std::equal(str.begin(), str.end(), of.begin());
    }
}
```

The `split` function ❶ uses `std::stringstream` and `std::getline` to read delimited text from the string we give it. The `std::getline` function will read a block of text from the given stream into `item` until it hits the given delimiter. We then collect all of these blocks into a `std::vector` and return it.

The `is_prefix` function ❷ is a small utility function that returns an indication of whether a string is either equal to or a prefix of another string. If you're using C++20, you can simplify this kind of string processing with ranges.

Finally, we use `ptrace` magic to make the inferior process continue:

```
namespace {
❶ void resume(pid_t pid) {
        if (ptrace(PTRACE_CONT, pid, nullptr, nullptr) < 0) {
            std::cerr << "Couldn't continue\n";
            std::exit(-1);
        }
    }

❷ void wait_on_signal(pid_t pid) {
        int wait_status;
        int options = 0;
        if (waitpid(pid, &wait_status, options) < 0) {
            std::perror("waitpid failed");
            std::exit(-1);
        }
    }
}
```

The `resume` function ❶ wraps a call to `ptrace` with the `PTRACE_CONT` request in some error handling. This request causes the operating system to resume the execution of the process. The `wait_on_signal` function ❷ similarly wraps a call to `waitpid`. Now we can evaluate the fruits of our efforts.

Manual Testing

Let's manually test the features we've just added. We'll automate these tests in the next chapter, after we've cleaned up the structure of the code.

We'll start with process launching. You should now be able to launch a process, have it stop once launched, and then resume its execution by entering `continue`. Give it a try by starting one of Linux's most useful programs, `yes`, which does nothing more than print out `y` over and over and over. Run it through *sdb* and continue it like so:

```
$ tools/sdb yes
sdb> continue
```

You should be immediately bombarded by an endless stream of `y`'s. This is good; it means the launching behavior is working.

Now we'll test process attaching. To attach the program to an existing process, you could try targeting a GUI application and noting that it halts execution when you start *sdb*. If you're using the terminal only, you *could* try running it with `yes` if you're brave enough, or you can run the following slightly less exciting test:

```
$ while sleep 5; do echo "I'm alive!"; done&
[1] 1247
$ tools/sdb -p 1247
```

The first command will print "I'm alive!" every five seconds. The ampersand at the end is important; it sends the command to the background.

The command should output the PID of the background process, which will likely be different on your machine. Pass this PID to *sdb* to attach to that process. When you do so, "I'm alive!" should stop printing. When you continue, it should print "I'm alive!" once and then return control back to the debugger because of the call to `waitpid`. Neat!

Depending on your Linux distribution, you may not be allowed to use `PTRACE_ATTACH` on processes that aren't children of *sdb*. This is due to the Yama Linux Security Module (LSM). There are two ways around this. First, you can globally allow attaching to non-child processes by setting LSM to "classic `ptrace` permissions" mode, like so:

```
$ echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

You may need to run this with `sudo`. Second, you can add capabilities to *sdb* to trace non-child processes with `setcap CAP_SYS_PTRACE=+eip sdb`.

Refactoring into a Library

We've written code the quick and dirty way to get going, but now it's time to refactor the project so it will serve us better as we scale up the debugger.

As mentioned in "The Directory Structure" on page 1, we'll place most of the debugger functionality in `libsdb` and treat *sdb* as a command line driver for the library. This will make testing much easier.

As such, `ptrace` shouldn't appear in *sdb*, which shouldn't even be aware of its existence. Let's create some data structures in `libsdb` that represent the components of the system and move the `ptrace` calls into them.

Creating a Process Type

We'll need a type that represents any running process we can launch, attach to, continue, and wait on signals for. We'll call it `sdb::process`. Like many types we'll make over the course of this book, `sdb::process` represents some unique resource. We shouldn't be able to copy an `sdb::process` object, because that would mean creating an entire new process on the system.

Therefore, users of the library will need to interact with `sdb::process` through pointers. To make our lives easier, we'll use *smart pointers*, which are wrappers for pointers that automatically manage the allocated memory, rather than requiring programmers to remember when to free it. Create a new file called *sdb/include/libsdb/process.hpp* with these contents:

```
#ifndef SDB_PROCESS_HPP
#define SDB_PROCESS_HPP

#include <filesystem>
#include <memory>
#include <sys/types.h>

namespace sdb {
    class process {
    public:
     ❶ static std::unique_ptr<process> launch(std::filesystem::path path);
        static std::unique_ptr<process> attach(pid_t pid);

     ❷ void resume();
        /*?*/ wait_on_signal();
        pid_t pid() const { return pid_; }

    private:
        pid_t pid_ = 0;
    };
}

#endif
```

We declare `launch` and `attach` member functions that create `sdb::process` objects ❶. The `launch` function takes the path to the program to launch, whereas `attach` takes the PID of the existing process to attach to.

A user should be able to resume a process that is currently halted, so we declare a member for that ❷. They should also be able to wait for the inferior to be signaled. We should return some information about what signal

was received, but we'll have to do some more thinking about what type to return, so for now, I've left this as an open question.

Finally, an `sdb::process` object needs to keep track of the PID for the process it is tracking and expose this to users, so we add a data member and a member function to retrieve it.

THE STD::UNIQUE_PTR SMART POINTER

Our process type uses the `std::unique_ptr` smart pointer. As an example of its use, if we allocate an `int` and store the pointer in a `std::unique_ptr`, we don't call `delete` when we're done with it because the smart pointer handles this for us:

```
{
    std::unique_ptr<int> i (new int(42));
}
```

Memory is allocated when `new` is called and automatically freed when the `std::unique_ptr` object is destroyed.

In some cases, it's safer to use the helper function `std::make_unique<T>`, as in `auto i = std::make_unique<int>(42);`, because it avoids some tricky problems related to exceptions and evaluation order. I'll prefer this when possible. See Chapter 4 of *Effective Modern C++* by Scott Meyers (O'Reilly, 2014) or the Stack Overflow question at *https://stackoverflow.com/questions/106508/what-is-a-smart-pointer-and-when-should-i-use-one* for more details.

We must make sure that users can't construct a `process` object without going through those static member functions and that they can't accidentally copy it, so we'll disable the default constructor and copy operations:

```
namespace sdb {
    class process {
        --snip--
     ❶ process() = delete;
     ❷ process(const process&) = delete;
        process& operator=(const process&) = delete;
        --snip--
    };
}
```

We delete the default constructor ❶ to force client code to use the static members and then delete the copy constructors ❷ to disable copy and move behavior.

We should clean up the inferior process if we launched it ourselves but leave it running otherwise. Let's add a destructor and a member to track whether we should terminate the process:

```
namespace sdb {
    class process {
    public:
     ❶ ~process();
        --snip--

    private:
        pid_t pid_ = 0;
     ❷ bool terminate_on_end_ = true;
    };
}
```

We declare the destructor as a public member ❶ and add a new private data member to track termination ❷. We should also keep track of the current running state of the process. We'll add an enum for this:

```
namespace sdb {
 ❶ enum class process_state {
        stopped,
        running,
        exited,
        terminated
    };

    class process {
    public:
        --snip--
     ❷ process_state state() const { return state_; }

    private:
        pid_t pid_ = 0;
        bool terminate_on_end_ = true;
     ❸ process_state state_ = process_state::stopped;
    };
}
```

The `process_state` enum ❶ represents the various situations in which a process may find itself. We represent it using a strongly typed enum (`enum class`), which is a C++11 feature that stops enumerator values from implicitly converting to and from integers and automatically qualifies the enumerator names with the name of the enum (like `process_state::stopped`). We add a member to track the state the process is in ❸ and expose this to users ❷. Finally, we need to provide a way for the static members to construct a `process` object. We do this with a private constructor:

```
namespace sdb {
    class process {
    --snip--
```

```
    private:
        process(pid_t pid, bool terminate_on_end)
            : pid_(pid), terminate_on_end_(terminate_on_end) {}
        --snip--
    };
}
```

We make this member private so that client code must use the static `launch` and `attach` functions to construct the `process` object.

Implementing launch and attach

Now we can implement the declared members, starting with `launch` and `attach`. For the most part, we can steal the code from *sdb/tools/sdb.cpp*. Implement them in *sdb/src/process.cpp*:

```
#include <libsdb/process.hpp>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

std::unique_ptr<sdb::process> sdb::process::launch(
      std::filesystem::path path) {
    pid_t pid;
    if ((pid = fork()) < 0) {
        // Error: fork failed ❶
    }

    if (pid == 0) {
        if (ptrace(PTRACE_TRACEME, 0, nullptr, nullptr) < 0) {
            // Error: Tracing failed
        }
        if (execlp(path.c_str(), path.c_str(), nullptr) < 0) {
            // Error: exec failed
        }
    }

    std::unique_ptr<process> proc (new process(pid, /*terminate_on_end=*/true));
    proc->wait_on_signal();

    return proc;
}

std::unique_ptr<sdb::process> sdb::process::attach(
      pid_t pid) {
    if (pid == 0) {
        // Error: Invalid PID
```

```
    }
    if (ptrace(PTRACE_ATTACH, pid, nullptr, nullptr) < 0) {
        // Error: Could not attach
    }

    std::unique_ptr<process> proc (new process(pid, /*terminate_on_end=*/false));
    proc->wait_on_signal();

    return proc;
}
```

First, `launch` carries out the `fork` and `exec` process you learned about at the start of the chapter; then it creates a new `sdb::process` object using the private constructor we just implemented and waits for the process to halt. Since we're launching the process, we pass `true` as the `terminate_on_end` argument. We'll address the comments about errors ❶ shortly.

Next, `attach` uses `PTRACE_ATTACH` to attach to the running process, then constructs the `sdb::process` and waits for the underlying process to halt. Since we're not launching the process in this case, we pass `false` as the `terminate_on_end` parameter.

You'll need to add this file to *sdb/src/CMakeLists.txt* to get it included in the build. While you're at it, you can remove the test *sdb/src/libsdb.cpp* and *sdb/include/libsdb/libsdb.hpp* files you created in Chapter 1. Replace the existing `add_library` call with this:

```
add_library(libsdb process.cpp)
```

The biggest change we made to the code ported over from *sdb/tools/sdb.cpp* was replacing the printing of error messages and the termination of the program with comments. Force-terminating the program from a library when the program may be able to recover is not good, but code comments aren't a great alternative. Unfortunately, it's time to think about errors.

Handling Errors

There are many different ways to handle errors in C++. The four main ones are ignoring them, using exceptions, using error codes, and relying on result types like `std::expected`.

The first option is unfortunately a common choice, but we can do better for this project. The others are all reasonable options. LLDB disables exceptions, so it uses custom result types and error codes. Projects written in C don't get many options, so they use error codes.

In *sdb*, we'll opt for exceptions. The debugger doesn't have the kind of memory footprint or predictability constraints that often cause projects to eschew exceptions, and exceptions will let us focus on the code's "happy path" while still being robust enough to help us diagnose errors if we make mistakes. We won't try to deal with every possible error case, but we will deal with many common ones so you don't have to spend hours diagnosing issues.

Let's begin by creating an *sdb*-specific exception type that we can use to differentiate our own errors from ones produced within the system, which we'll need to handle properly. We'll put the type in *sdb/include/libsdb/error.hpp*:

```
#ifndef SDB_ERROR_HPP
#define SDB_ERROR_HPP

#include <stdexcept>
#include <cstring>

namespace sdb {
    class error : ❶ public std::runtime_error {
    public:
        [[noreturn]]
        static void send(const std::string& what) { throw error(what); }
        [[noreturn]]
        static void send_errno(const std::string& prefix) {
            throw error(prefix + ": " + ❷ std::strerror(errno));
        }

    private:
        error(const std::string& what) : std::runtime_error(what) {}
    };
}

#endif
```

The `sdb::error` type inherits from `std::runtime_error` ❶, as it's a special kind of runtime error. We provide two ways to create one: `error::send`, which takes a message to use as the error description, and `error::send_errno`, which uses the contents of `errno` as the error description, adding the message we provide as a prefix. We declare both of these with the `[[noreturn]]` attribute, which indicates to the compiler that this function does not return control flow when it exits. This will prevent the compiler from issuing unnecessary warnings in some cases.

While `send` simply throws a new error with the given message, `send_errno` calls `std::strerror` ❷ to get a string representation of `errno`. This function is similar to `std::perror`, but it returns the message as a string rather than printing it to `stderr`.

Finally, we make a private constructor that forwards the error message on to the `std::runtime_error` constructor.

Let's put the new type into practice in the `attach` and `launch` functions in *sdb/src/process.cpp*:

```
#include <libsdb/error.hpp>

std::unique_ptr<sdb::process> sdb::process::launch(std::filesystem::path path) {
    pid_t pid;
    if ((pid = fork()) < 0) {
        error::send_errno("fork failed");
    }

    if (pid == 0) {
        if (ptrace(PTRACE_TRACEME, 0, nullptr, nullptr) < 0) {
            error::send_errno("Tracing failed");
        }
        if (execlp(path.c_str(), path.c_str(), nullptr) < 0) {
            error::send_errno("exec failed");
        }
    }

    std::unique_ptr<process> proc (new process(pid, /*terminate_on_end=*/true));
    proc->wait_on_signal();

    return proc;
}

std::unique_ptr<sdb::process> sdb::process::attach(pid_t pid) {
    if (pid == 0) {
        error::send("Invalid PID");
    }
    if (ptrace(PTRACE_ATTACH, pid, nullptr, nullptr) < 0) {
        error::send_errno("Could not attach");
    }

    std::unique_ptr<process> proc (new process(pid, /*terminate_on_end=*/false));
    proc->wait_on_signal();

    return proc;
}
```

I've replaced all the comments from the old version of the code with calls to either `error::send` or `error::send_errno`, depending on whether `errno` was set by the operation that failed.

You might still notice an issue with the error handling in `process::launch`; I briefly mentioned this when we implemented tracing. We'll fix it later, in Chapter 4. For now, let's move on to the destructor.

Destructing Processes

We can implement the destructor by calling `kill` on the child and waiting until it exits. (I wish the function were named something nicer, like `politely_ask_to_stop`, but oh well.) Here is the process-destructing code, which should go in *sdb/src/process.cpp*:

```
sdb::process::~process() {
    if (pid_ != 0) {
        int status;
        if (state_ == process_state::running) {
         ❶ kill(pid_, SIGSTOP);
            waitpid(pid_, &status, 0);
        }
     ❷ ptrace(PTRACE_DETACH, pid_, nullptr, nullptr);
        kill(pid_, SIGCONT);

        if (terminate_on_end_) {
         ❸ kill(pid_, SIGKILL);
            waitpid(pid_, &status, 0);
        }
    }
}
```

If we have a valid PID when the destructor runs, then we want to detach. For `PTRACE_DETACH` to work, the inferior must be stopped, so if it is currently running, we send it a `SIGSTOP` ❶ and wait for it to stop. We then detach from the process ❷ and let it continue. Finally, if we earlier determined that we should terminate the inferior when the managing `sdb::process` destructs, we send it a `SIGKILL` ❸ and wait for it to terminate.

Note that we don't handle any errors here or throw exceptions. Throwing exceptions from destructors is generally a no-no because if you're calling the destructor in the first place due to an exception being thrown up the stack, there's no way to throw an additional exception, so the program is terminated. We could log errors somewhere, but this will do for our purposes. You'll have to just believe in your destructor.

Next, we'll turn to the last two functions in `process`.

Resuming the Process

We want `sdb::process::resume` to force the process to resume and update its tracked running state. Implement it in *sdb/src/process.cpp*:

```
void sdb::process::resume() {
    if (ptrace(PTRACE_CONT, pid_, nullptr, nullptr) < 0) {
        error::send_errno("Could not resume");
    }
    state_ = process_state::running;
}
```

We simply issue a `PTRACE_CONT` command, check for errors, and update the state to be `running`.

Waiting on Signals

Now let's implement `wait_on_signal`. I said earlier that we should return some information about the signal that occurred. Let's make a type for this purpose in *sdb/include/libsdb/process.hpp*:

```
#include <cstdint>

namespace sdb {
    struct stop_reason {
        stop_reason(int wait_status);

     ❶ process_state reason;
     ❷ std::uint8_t info;
    };

    class process {
    public:
        --snip--
     ❸ stop_reason wait_on_signal();
        --snip--
    };
}
```

The `sdb::stop_reason` type holds the reason for a stop (whether the process exited, terminated, or just stopped) ❶ and some information about the stop (such as the return value of the exit or the signal that caused a stop or termination) ❷. This information will come from the `status` output parameter of `waitpid`, which we'll parse inside the `stop_reason` constructor. We also fill in that question mark from earlier in the return type of `wait_on_signal` with the new `stop_reason` type ❸.

Let's parse the `waitpid` status in the `stop_reason::stop_reason` implementation, found in *sdb/src/process.cpp*. We can use a series of macros to inspect the status that `waitpid` gives us:

```
sdb::stop_reason::stop_reason(int wait_status) {
    if (WIFEXITED(wait_status)) {
        reason = process_state::exited;
        info = WEXITSTATUS(wait_status);
    }
    else if (WIFSIGNALED(wait_status)) {
        reason = process_state::terminated;
        info = WTERMSIG(wait_status);
    }
```

```
    else if (WIFSTOPPED(wait_status)) {
        reason = process_state::stopped;
        info = WSTOPSIG(wait_status);
    }
}
```

First, `WIFEXITED` tells us if a given status represents an exit event; then, `WEXITSTATUS` extracts the exit code. We use `WIFSIGNALED` and `WIFSTOPPED` to figure out whether the stop was due to a termination or a stop and `WTERMSIG` and `WSTOPSIG` to extract the signal codes.

We can now use this type inside `wait_on_signal`:

```
sdb::stop_reason sdb::process::wait_on_signal() {
    int wait_status;
    int options = 0;
    if (waitpid(pid_, &wait_status, options) < 0) {
        error::send_errno("waitpid failed");
    }
    stop_reason reason(wait_status);
    state_ = reason.reason;
    return reason;
}
```

We call `waitpid`, update the state of the process based on the stop reason, and then return the reason to the caller. Now that `sdb::process` has all the `ptrace` functionality we relied on in *sdb/tools/sdb.cpp*, we can go back and substitute it in for the `ptrace` calls. As a result, `attach` becomes this:

```
#include <libsdb/process.hpp>

namespace {
    std::unique_ptr<sdb::process> attach(int argc, const char** argv) {
        // Passing PID
        if (argc == 3 && argv[1] == std::string_view("-p")) {
            pid_t pid = std::atoi(argv[2]);
         ❶ return sdb::process::attach(pid);
        }
        // Passing program name
        else {
            const char* program_path = argv[1];
         ❷ return sdb::process::launch(program_path);
        }
    }
}
```

We replace the `ptrace` calls that attached to the process with a call to `sdb::process::attach` ❶. We then do a similar replacement for the `launch` code ❷. We can get rid of the `resume` and `wait_on_signal` functions in *sdb/tools/sdb.cpp* and replace `handle_command` with this:

```
namespace {
    void handle_command(
          std::unique_ptr<sdb::process>& process,
       ❶ std::string_view line) {
        auto args = split(line, ' ');
        auto command = args[0];

        if (is_prefix(command, "continue")) {
         ❷ process->resume();
            process->wait_on_signal();
        }
        else {
            std::cerr << "Unknown command\n";
        }
    }
}
```

First, we update the signature to take a `std::unique_ptr<sdb::process>&` instead of a PID ❶. We then call our new version of `resume` that lives inside `sdb::process`. If the user issues a `continue` command, we call `resume` ❷ and `wait_on_signal` on the given process.

Even better, we can augment that call to `process->wait_on_signal()` to print out some details for the user:

```
namespace {
    void print_stop_reason(
          const sdb::process& process, sdb::stop_reason reason) {
        std::cout << "Process " << process.pid() << ' ';

        switch (reason.reason) {
        case sdb::process_state::exited:
            std::cout << "exited with status "
                      << static_cast<int>(reason.info);
            break;
        case sdb::process_state::terminated:
            std::cout << "terminated with signal "
                   ❶ <<  sigabbrev_np(reason.info);
            break;
        case sdb::process_state::stopped:
            std::cout << "stopped with signal " << sigabbrev_np(reason.info);
            break;
        }

        std::cout << std::endl;
    }

    void handle_command(
```

```
          std::unique_ptr<sdb::process>& process,
          std::string_view line) {
        --snip--
        if (is_prefix(command, "continue")) {
            process->resume();
            auto reason = process->wait_on_signal();
         ❷ print_stop_reason(*process, reason);
        }
        --snip--
    }
}
```

We introduce a function called `print_stop_reason` that, as you might expect, prints the stop reason. It starts by printing out the inferior's PID for the user, then prints out a message saying why that process stopped. If it exited, we print the exit status, and if it terminated or stopped due to a signal, we print the signal name. Fortunately, there is a function called `sigabbrev_np` ❶ that gets the signal abbreviation for a given signal code, so we use that here. If you're using a toolchain that doesn't supply the `sigabbrev_np` function, you can instead index the `sys_siglist` array, like `sys_siglist[reason.info]`. We also update the implementation of `handle_command` to call `print_stop_reason` ❷.

The last refactoring step is to update `main` to marshal the `sdb::process` around and report exceptions back to the user. We'll place the main loop of the debugger into a `main_loop` function that gets called from `main`:

```
#include <libsdb/error.hpp>

namespace {
    void main_loop(std::unique_ptr<sdb::process>& process) {
        char* line = nullptr;
        while ((line = readline("sdb> ")) != nullptr) {
            std::string line_str;

            if (line == std::string_view("")) {
                free(line);
                if (history_length > 0) {
                    line_str = history_list()[history_length - 1]->line;
                }
            }
            else {
                line_str = line;
                add_history(line);
                free(line);
            }

            if (!line_str.empty()) {
                try {
                 ❶ handle_command(process, line_str);
                }
```

```
            ❷ catch (const sdb::error& err) {
                  std::cout << err.what() << '\n';
              }
          }
      }
  }
}

int main(int argc, const char** argv) {
    if (argc == 1) {
        std::cerr << "No arguments given\n";
        return -1;
    }

    try {
     ❸ auto process = attach(argc, argv);
        main_loop(process);
    }
 ❹ catch (const sdb::error& err) {
        std::cout << err.what() << '\n';
    }
}
```

We extract the main loop in `main` into the new `main_loop` function, which takes the process as an argument. Then we pass that process through to `handle_command` ❶. If we encounter an error while handling a user command, we report the error and continue running so they can issue more commands. We achieve this with a catch handler ❷.

In `main`, we pass the process returned by `attach` ❸ to `main_loop`. If launching or attaching to the initial process fails, we just exit ❹.

I'm sure you'll agree that this code will be much easier to manipulate and test than the old version.

Summary

In this chapter, you built a library that can launch, attach to, and continue Linux processes. You also wrote a command line interface for it that exposes these facilities to the user.

In the next chapter, you'll learn how to use pipes to communicate between the debugger and the launched process, and you'll use them to implement automated tests for the code you just wrote.

Check Your Knowledge

1. What facilities are used on Linux to launch a new process?
2. What is the name of Linux's debug API?
3. How do we wait for a child process to be signaled on Linux?