Assignment 3

CS330: Operating Systems

1 Introduction

In this assignment, you will be implementing a simple debugger in gemOS. Please go through the following resources to understand how a debugger such as gdb works in Linux.

- http://www.alexonlinux.com/how-debugger-works
- https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1
- https://eli.thegreenplace.net/2011/01/27/how-debuggers-work-part-2-breakpoints
- https://eli.thegreenplace.net/2011/02/07/how-debuggers-work-part-3-debugging-information

2 Overview

Breakpoints are used to stop the execution of a process at a particular location so that debugger can analyse the state of the process being debugged at that point.

(A) INIT (present in `gemOS/src/user/init.c`) will act as a debugger.

(B) INIT will fork a new child process. This child process will act as the process to be debugged (debuggee). Thus the debugger and debuggee will share the same code segment.

(C) Debugger process will use system calls such as `set_breakpoint()`, `remove_breakpoint()` etc. to trace its child process.

We don't want our debuggee process to finish executing before debugger gets a chance to set breakpoints or query register values. This means that the debuggee process shouldn't start running as soon as it gets created. We've modified the fork implementation for this purpose. Refer: [`debugger_on_fork` in `gemOS/src/debug.c` and `do_fork` in `gemOS/src/entry.c`]

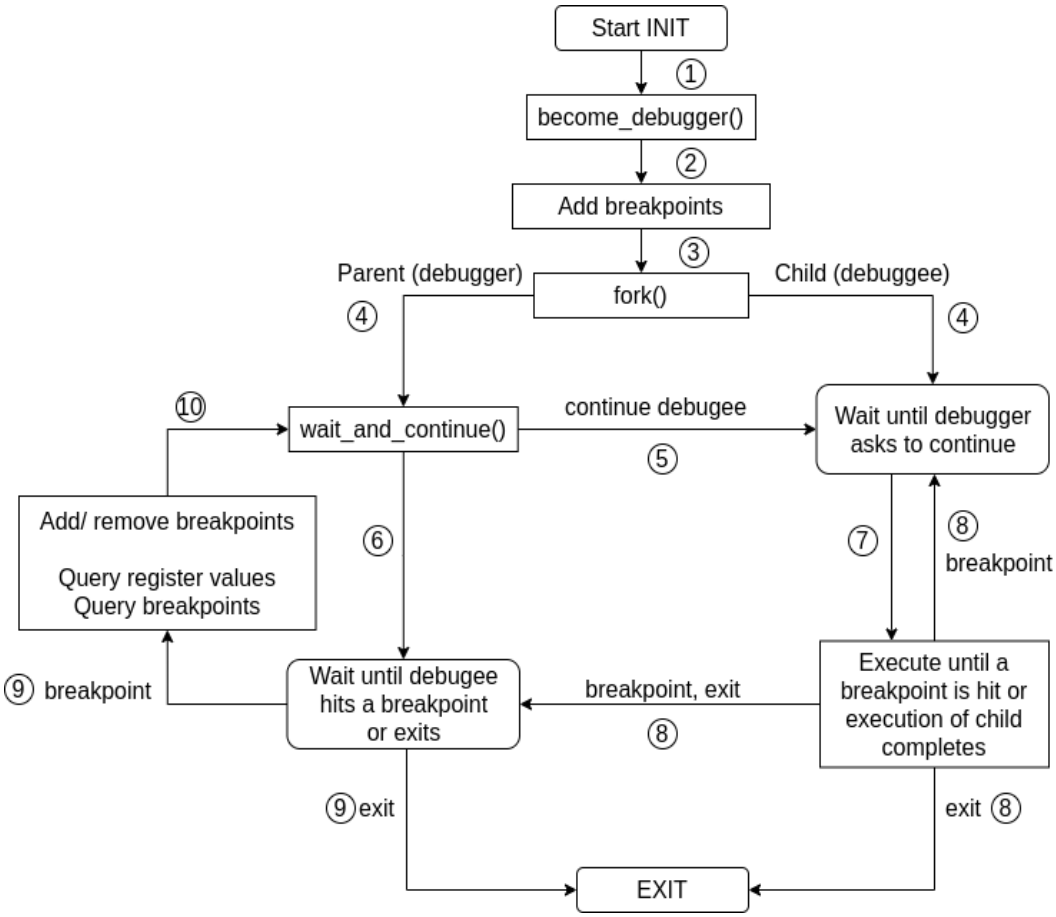

Figure 1: High level overview of the working of debugger

3 Given

(A) You have been given a sample user-space implementations of the debuggers in `gemOS/src/user/test/`. You can copy the contents from any of these to `gemOs/src/user/init.c` and test the debug functionalities. However, you need to implement the system calls that the debugger will need to work properly.

(B) GemOS has already been configured to call `int3_handler` when `INT3` (hex-code = `0xCC`) interrupt is triggered. You just need to fill in the definition of `int3_handler` in `gemOS/src/debug.c`.

(C) The file `gemOS/src/debug.c` contains all the templates for the system call implementations you need to complete.

4 Assumptions

(A) In gemOS, code segment is shared between parent process and child process. Therefore, when some code segment is written by parent or child, the changes in the code segment will be reflected both in the parent and the child.

(B) A debugger will fork only one debuggee process and this debuggee process will not fork any child of its own. Thus, there will be only two processes, debugger and debuggee.

(C) Debugger (parent) will not call any function on which breakpoint has been set.

(D) Debuggee will not call any debugger system calls.

(E) To schedule another process (by replacing the calling process), `schedule(P)` is invoked, where P is the `exec_context` of the process which is scheduled. Note that, any code written after the `schedule()` call will not be executed (even when the outgoing process is scheduled back). Consider the following system call snippet in gemOS.

```
int some_syscall(){
        struct exec_context *ctx = some_other;
        printk("Hello gemOS\n");
        schedule(some_other_ctx);
        printk("Bye gemOS\n");
}
```

If process P1 makes the above system call, "Bye gemOS" will not be printed. This is because, in gemOS, the process P1 will resume in user-space (after the system call invocation).

5 Task 1: Basic Debugger [80 marks]

In this task you will be implementing following system calls and functions to support debugging. Each sub-task is a functionality that you are expected to implement.

5.1 int become_debugger(void *addr)

Description of args: addr : The address of the *do_end_handler* function, which is present in `user/init.c`.

Description of functionality: Initialize the data structures related to the debugger inside this function. See the description for `set_breakpoint`. The *do_end_handler* function address is passed as the argument **addr**, which is to be

stored in the *end_handler* field in the parent context's `struct debug_info` (see `src/include/debug.h`). If a breakpointed function has *end_breakpoint_enable* flag set (see below), then control should pass from the end of that breakpointed function to this *do_end_handler* function. In this *do_end_handler* function, **INT3** interrupt should be raised. This will call the interrupt handler (see *int3_handler*) which should then schedule the parent. When child is scheduled back again later, it should continue from the next line in *do_end_handler* function. The *do_end_handler* function on returning, passes control to the place where the breakpointed function would have returned to.

Return value: 0 on success. -1 on failure.

5.2 int set_breakpoint(void *addr, int flag)

Description of args:
addr : The address of function where breakpoint is to be set
flag : Can be either 0 or 1. Flag indicates if the debugee function at the end should return normally or go to end_handler

Description of functionality: Set a breakpoint on debuggee process at address **addr**. You have to manipulate debuggee's address space so that whenever the instruction at **addr** is executed, `INT3` would be triggered.

If **flag** is set, then you must ensure that when the breakpointed function is about to return, control instead passes to the *do_end_handler* function, rather than returning. You should save the flag in the `end_breakpoint_enable` member inside `struct breakpoint_info` (see `src/include/debug.h`)

Save the relevant information about the breakpoint such as the address, end_breakpoint flag and breakpoint number in parent process's execution context.

Debugger process's `exec_context` (`gemOS/src/include/context.h`) contains a field `struct debug_info *dbg` and `struct debug_info` contains a field `struct breakpoint_info *head` as shown below.

```
struct exec_context{
    ...
    struct debug_info *dbg;
    ...
};

struct debug_info{
    struct breakpoint_info *head;   // head of breakpoint list
};

struct breakpoint_info{
        u32 num;                        // - breakpoint number
        u64 addr;                       // - address on which breakpoint is set
```

```
        struct breakpoint_info *next;   // - pointer to next node in breakpoint list
        int end_breakpoint_enable;      // - whether control should pass to end_handler
                                        // at breakpointed function return
};
```

You should store information about new breakpoint in new `struct breakpoint_info` object and insert it at the tail of the list pointed by *head*. Initially, *head* is NULL.

Example: Suppose, debugger makes following system calls. Assume `MAX_BREAKPOINT` is 2. Breakpoint numbers assigned are shown in brackets:
`set_breakpoint(addr1)`; (breakpoint number assigned: 1)
`set_breakpoint(addr2)`; (breakpoint number assigned: 2)
`remove_breakpoint(addr1)`; (breakpoint number 1 removed)
`set_breakpoint(addr1)`; (breakpoint number assigned: 3)
`set_breakpoint(addr1)`; (fails since `MAX_BREAKPOINT` is 2)
Note that your breakpoint implementation must support recursion i.e. If we set a breakpoint on `foo()`, it should trigger an interrupt every time `foo()` is called, till the breakpoint is disabled or removed.

Assumptions:

(A) Breakpoint is always set only on functions.

(B) **addr** is always a valid address i.e., the first instruction of a function.

(C) Maximum number of breakpoints set at any time (including both enabled as well as disabled) will be specified by `MAX_BREAKPOINTS` (present in `gemOS/src/include/debug.h`).

(D) If `MAX_BREAKPOINTS` number of breakpoints are already set and `set_breakpoint()` is called again, return -1 (error).

(E) If a breakpoint corresponding **addr** already exists, return 0 (success) and set its status to *enabled*.

(F) The breakpoint number is incremented if and only if set_breakpoint was successful.

Return Value: 0 on success, -1 on error

5.3 int remove_breakpoint(void *addr)

Description of args:
addr : The address corresponding to the breakpoint to be removed

Description of functionality:
Once a debugger is done with the process of analysing the state of debuggee

at a particular address, the debugger generally removes the breakpoint from that address so that, when debuggee's execution reaches this address again, no breakpoint is generated. In this system call, you have to implement this feature of removing any breakpoint.

You are required to find the breakpoint entry corresponding to the address specified by **addr** in list of breakpoints pointed to by *head* which is accessible from *dbg* pointer which is part of the `exec_context` structure. You have to completely remove the information about this breakpoint that you have maintained. Also, make sure that when in the future the child process's execution reaches **addr**, `INT3` shouldn't be generated anymore.
If no breakpoint corresponding `addr` exists, return -1 (error).
If `remove_breakpoint` is called on a function that is currently active on the call stack of debuggee process (function has been called but not returned yet), AND its end_breakpoint_enable flag is set to the value 1, then also `remove_breakpoint` should return -1 (error).
For example, suppose f1 has a breakpoint with end_breakpoint_enable set to 1, and f2 is another function with a breakpoint. Assume we are at the breakpoint of f2(). Let the call stack be main() → f1() → f2(). Note that f1() is still on the call stack. So here if user tries to do `remove_breakpoint(f1)`, it should fail.
Note that this rule does not apply for functions with end_breakpoint_enable value 0.

Assumptions:

(A) Address specified by **addr** can be any address.

Return Value: 0 on success, -1 on error

5.4 s64 wait_and_continue()

Description of functionality:
Stop execution of the debugger process and continue execution of the debugee process. The debugger should go to the WAITING state and the debugee should go to the READY state and should be scheduled now.

The Debugger process's execution will be resumed in the following scenarios:

(A) `INT3` interrupt has occurred because of the debugee process hitting a breakpoint.

(B) Debugee process has exited.

Return Value:

(A) If the debuggee process has reached a breakpoint, return the address of the breakpoint.

(B) If debuggee has exited, return `CHILD_EXIT` (defined in `gemOS/src/include/debug.h`)

(C) If an error occurred, return -1.

5.5 void int3_handler()

Description of functionality:
Whenever an `INT3` instruction (Byte value: `0xCC`) gets executed, interrupt number 3 is generated and control goes to the interrupt 3 handler. We have already done the configuration so that `int3_handler()` is called when an `INT3` instruction gets executed. You just have to fill the definition of this function.
If you have setup breakpoint correctly, control will reach `int3_handler` when breakpoint is encountered in the debugee process. In `int3_handler`, you have to stop the execution of the debugee process and start execution of the debugger process, so that the debugger can analyse the current state of the debuggee.

If the *end_breakpoint_enable* flag of the breakpointed function is set, then the aforementioned control transfer should also happen when returning from the breakpointed function. In `int3_handler`, you should figure out where the control came from (from beginning of function or from *end_handler*) and handle accordingly.

Return Value: 0 on success, -1 on error

5.6 void debugger_on_exit(struct exec_context *ctx)

Note that, this isn't a system call but a function called from the process exit handler (`do_exit` in `gemOS/src/entry.c`) executed whenever a process exits.

Description of functionality:

When a debugger exits, you have to make to sure that memory allocated by debugger is freed. e.g, Memory allocated for storing information about breakpoints. When a debuggee exits, debugger should be scheduled and informed that debuggee has exited (See the description of `wait_and_continue`).

Assumption: Debugger will never exit before debuggee.

5.7 int info_breakpoints(struct breakpoint *bp)

Description of args: bp is an array of `struct breakpoint` defined in `gemOS/src/include/debug.h`. Size of this array is `MAX_BREAKPOINTS`

Description of functionality: `info_breakpoints` is used by debugger to get information about all breakpoints currently set. For each breakpoint, following information is provided.

(A) Break point address

(B) End-Breakpoint flag i.e. whether breakpoint at end-of-function is enabled (1) or disabled (0).

(C) Break point number

Array of `struct breakpoint` i.e. **bp** will be passed from the user space as the argument to `info_breakpoints()`. You have to fill information about the breakpoints in this array of structure and return the number of set breakpoints. Entries in `bp` should be stored as per the increasing values of breakpoint number i.e. breakpoint with lowest breakpoint number should be stored in `bp[0]`, the second lowest in `bp[1]` and so on.

Return: Number of breakpoints set (0 to `MAX_BREAKPOINTS`) in case of success. -1 in case of error.

5.8 int info_registers(struct registers *reg)

Description of args: Address of an element of type `struct registers` defined in `gemOS/src/user/ulib.h` amd `gemOS/src/include/debug.h`

Description of functionality: `info_registers` is used to find the values present in registers just before a breakpoint occurs during the execution of debuggee. Address of a `struct register` type variable will be passed from user space as the argument of `info_registers`. You are required to fill the register values of child process just before the `INT3` instruction gets executed (i.e. just before breakpoint occurred) into this structure passed as argument.

Return: 0 on success, -1 on error

Testing: While testing the correctness of register state obtained using info_registers, we will only check the values corresponding the following registers:

1. RIP, RSP, RBP, RAX
2. Registers used to store arguments passed to functions i.e. RDI, RSI, RDX, RCX, R8, and R9

6 Task 2: Adding stack back-trace functionality [20 marks]

In this question you will be implementing the following system call.

6.1 int backtrace(unsigned long *bt)

Description of args:
bt points to an array of `u64` elements (i.e., each element = 8bytes) of size `MAX_BACKTRACE` (`gemOS/src/include/debug.h`). Result of backtrace will be loaded in this array.

Description of functionality: Backtrace allows a debugger to analyse the call stack of the debuggee process. In this question you will be implementing a simple version of backtrace. This simple version of backtrace will only report the return addresses pushed on to the stack as one function calls another function. All the return addresses (starting from the function where breakpoint is set) till the `main` function should be filled in the **bt** array.

Example: Suppose `main()` calls `fnA` and `fnA` calls `fnB`. Suppose breakpoint is set on `fnB` (with end_breakpoint_enable's value equal to 0). In this case, bt[0] should contain the address of first instruction of function on which breakpoint occurred i.e. `fnB` in this case. Return address in `fnA` (saved in stack when `fnB` was called) should be loaded in bt[1]. Return address in `main` (saved in stack when `fnA` was called) should be loaded in bt[2]. Return address stored in main's stack frame is `END_ADDR` (defined in `gemOS/src/include/debug.h`). When this address is encountered, the back tracing should stop. `END_ADDR` should not be included in **bt**. So, for the above example, 3 values will be stored in bt[0], bt[1], bt[2] and the `backtrace` system call should return 3.
Note that, if end_breakpoint_enable on fnB had the value 1, and breakpoint hit at the end of fnB, then the backtrace will contain only 2 entries, i.e., the function address of fnB will not be in the backtrace, since fnB has exited.

Return: On success, return the number of return addresses that are part of the backtrace. In case of error, return -1.

Assumptions: There will be at most `MAX_BACKTRACE` number of return addresses as part of backtrace testing.

7 Helper API's:

These are some of the scheduling and memory related API's provided by gemOS that maybe helpful to you.

(A) `struct exec_context *get_ctx_by_pid(u32 pid);` - Given a pid, it returns a pointer to the `exec_context` struct associated with it.

(B) `struct exect_context *get_current_ctx(void);` - Returns the pointer to `exec_context` struct associated with the current running process.

(C) `struct exect_context *pick_next_context(struct exec_context *ctx);` - Returns pointer to the `exec_context` struct of the next process that can be scheduled. Pointer to `exec_context` struct of the current running process is passed as the argument.

(D) void `schedule(struct exec_context *ctx);` - Schedules the process corresponding to the passed `exec_context` argument.

(E) `void *os_alloc(u32 size);` - Allocates a memory region of `size` bytes. Note that you cannot use this function to allocate regions of size greater than 2048 bytes.

(F) `void os_free(void *ptr, u32 size);` - De-allocates the memory region allocated by `os_alloc`.

8 Modifications allowed

(A) Among non-header files, you can only modify `gemOS/src/debug.c`

(B) You can't modify any header file apart from `gemOS/src/include/debug.h`

(C) You can't remove anything from `gemOS/src/include/debug.h`

(D) You can add new structures (and new members in existing structures) in `gemOS/src/include/debug.h`

(E) `struct debug_info` provided by us contains a field `struct breakpoint_info *head`. Don't remove/rename it.

9 Testing

(A) Please make sure that you are compiling and testing your code inside the provided docker environment.

(B) In the GemOS terminal (accessed using the telnet command), you can type `init` to execute the user space process (debugger).

(C) A sample debugger code is available in `gemOS/src/user/init.c`

(D) You need to write your test cases in `init.c` to validate your implementation.

(E) The sample test-cases (in `gemOS/src/user/test`) can be copied into `init.c` to use them.

(F) Test your implementation by running these tests (and also your own test-cases) and ensuring that the expected output is observed.

(G) The user and kernel code are compiled into a single binary file, i.e., `gemOS.kernel` when built using 'make' from the src directory.

10 Submission

You have to submit a zip file containing following source files.

(A) `gemOS/src/include/debug.h`

(B) `gemOS/src/debug.c`

Name of the zip file must be <your rollno>.zip

11 Hints and Clarifications

- To see the assembly code of a function, you can use the `objdump` utility. For example, write a C file (say `sample.c`) with some functions in it. Compile it (say `sample.out` is the binary created), and run `objdump -D sample.out` on the terminal.
  You can use this to view the assembly code of the object files provided. For more information on `objdump`, see `man objdump`.
- Before scheduling a process using the `schedule` function, make sure that the process which is being scheduled is in **READY** state.
- Once the process jumps to do_end_handler, the caller function is assumed to have exited. So, at the end-breakpoint of a function func(), system calls like `remove_breakpoint(func)` should succeed. Similarly, `backtrace` should not include the caller function in the return address stack that is filled.
- The debugger process can make system calls irrespective of how it gained control - whether from breakpoint at start of function or at end. The calls' success or failure may depend on it though.