

# 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]`

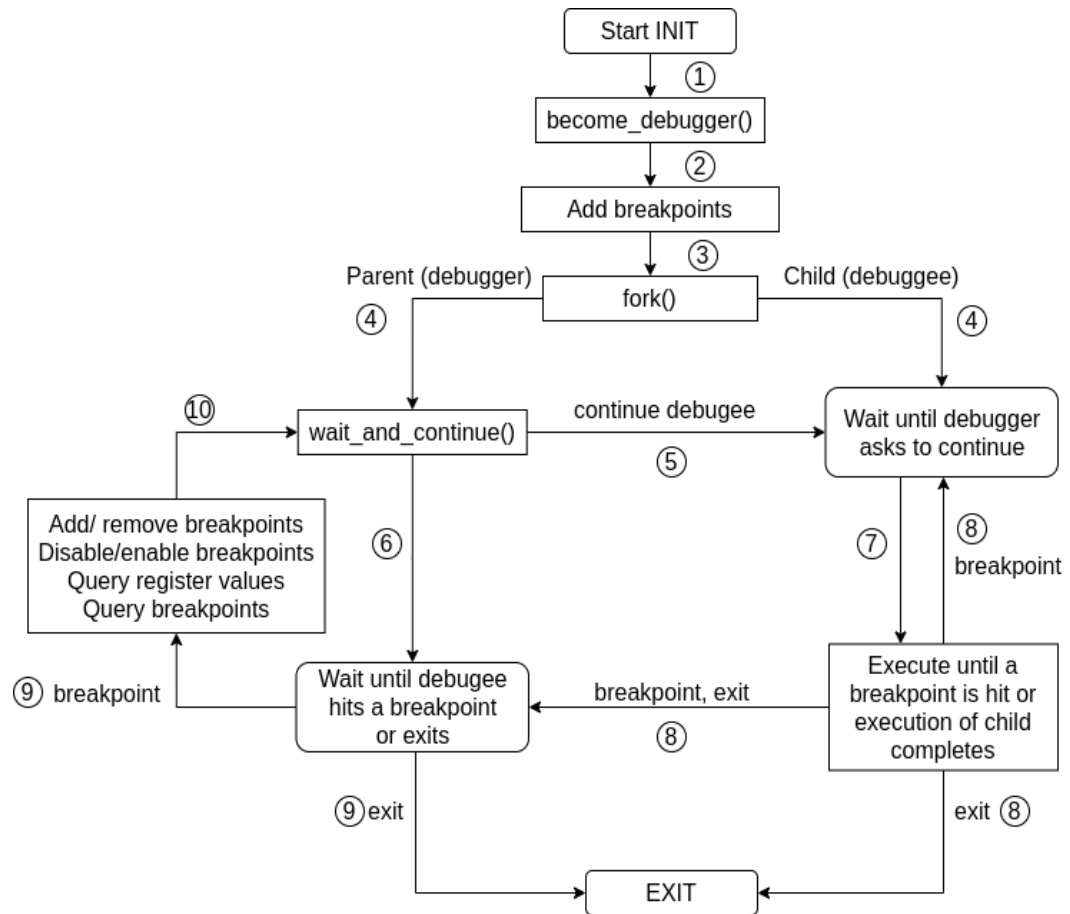


Figure 1: High level overview of the working of debugger

### 3 Given

- (A) You have been given a sample user-space implementation of the debugger in `gemOS/src/user/init.c`. 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 [60 marks]

In this question you will be implementing following system calls and functions to support debugging.

### 5.1 `int become_debugger()`

**Description of functionality:** Initialize the data structures related to the debugger inside this function. See the description for `set_breakpoint`.

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

## 5.2 int set\_breakpoint(void \*addr)

### Description of args:

**addr** : The address of function where breakpoint is to be set

**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. Save the relevant information about the breakpoint such as the address, status (i.e. whether the breakpoint is enabled or disabled) 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
    u32 status;                     // breakpoint status
    u64 addr;                       // address on which breakpoint is set
    struct breakpoint_info *next;    // pointer to next node in breakpoint list
};
```

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

Newly created breakpoint using `set_breakpoint()` will have status *enabled* ( status = 1 means *enabled* and status = 0 means *disabled*) and breakpoint number will be assigned an increasing integer value starting from 1.

**Example:** Suppose, debugger makes following system calls. Breakpoint numbers assigned are shown in brackets:

```
set_breakpoint(addr1); (breakpoint number assigned: 1)
set_breakpoint(addr2); (breakpoint number assigned: 2)
remove_breakpoint(addr1);
set_breakpoint(addr1); (breakpoint number assigned: 3)
```

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*.

**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).

**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 debugee process has reached a breakpoint, return the address of the breakpoint.
- (B) If debugee 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 debugee.

**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 debugee exits, debugger should be scheduled and informed that debugee has exited (See the description of `wait_and_continue`).

**Assumption:** Debugger will never exit before debuggee.

## 6 Task 2: Adding additional functionality to the Basic Debugger [20 marks]

In this question you will be implementing following system calls to support debugging

### 6.1 `int disable_breakpoint(void *addr)`

**Description of args:**

**addr:** The address corresponding to which the breakpoint is to be disabled.

**Description of functionality:**

Disabling a breakpoint removes the impact of this breakpoint on the execution of debuggee i.e. debuggee's execution won't stop at the address corresponding to this breakpoint in future. However, the state of this breakpoint isn't removed as the breakpoint may be re-enabled in future.

In this system call you have to find the breakpoint corresponding to the specified address **addr**. Disable this break point i.e., modify the code such that, when debuggee executes this address in future, breakpoint shouldn't occur and instead the actual instruction should get executed.

Don't delete any information kept by debugger about this breakpoint. Just change the status of this breakpoint from enabled (1) to disabled (0). If no breakpoint exists at address **addr**, return -1 (error). If breakpoint corresponding to **addr** exists and is already disabled, return 0 (success).

**Return Value:** 0 on success, -1 on error

### 6.2 `int enable_breakpoint(void *addr)`

**Description of args:**

**addr:** The address corresponding to which the breakpoint is to be enabled

**Description of functionality:**

Enable breakpoint is used to cancel the effect of the disable breakpoint. In this system call you have to find a breakpoint corresponding to the specified address **addr**. Enable this break point i.e. when execution of debuggee reaches this address in future, breakpoint should occur. Also, change the status of this breakpoint from disabled (0) to enabled (1). If no breakpoint corresponding to **addr** exists, return -1 (error). If breakpoint corresponding **addr** exists and is already enabled, return 0 (success).

**Return Value:** 0 on success, -1 on error

### 6.3 `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) Status of breakpoint i.e. whether the breakpoint 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 (irrespective of enabled or disabled) 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.

### 6.4 `int info_registers(struct registers *reg)`

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



## 7 Task 3: Adding stack back-trace functionality [20 marks]

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

### 7.1 `int backtrace(void *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`. 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.

**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.

## 8 Helper API's:

These are some of the scheduling and memory related API's provided by `gemOS` that may be 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 exec_context *get_current_ctx(void);` - Returns the pointer to `exec_context` struct associated with the current running process.

- (C) `struct exec_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`.

## 9 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 in `gemOS/src/include/debug.h`
- (E) Among the existing structures present in `gemOS/src/include/debug.h`, you can only modify `struct debug_info`.
- (F) `struct debug_info` provided by us contains a field `struct breakpoint_info *head`. Don't remove/rename it.

## 10 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_cases`) can be copied into `init.c` to use them.
- (F) If your implementation is correct, the output of executing test cases should match the expected output provided in `gemOS/src/user/test_cases`.

- (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.

## 11 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`