Debugging Kernel and User Space Synchronously based on GDB

Zhiyang Chen, Jingbang Wu*, Yaqi Yang, Yifan Zhang
School of Computer and Artificial Intelligence
Beijing Technology and Business University
Beijing, China
* Corresponding author: wujingbang@btbu.edu.cn

Hongyuan Wang
Faculty of Information Technology
Beijing University of Technology
Beijing, China

***Abstract*—Debugging operating systems running on multiple privilege levels is challenging due to symbol table inconsistencies caused by context switching. Conventional debuggers fail to manage breakpoints and symbol tables when the OS transitions between kernel and multiple user processes. To address this, we propose a debugging method based on breakpoint groups and their switching automation. By grouping and caching breakpoints and dynamically switching to the working breakpoint group, our method ensures consistent debugging across different execution environments. Experiments on RISC-V hardware and QEMU confirm the effectiveness of the method in debugging OSes like xv6 (written in C) and Starry (written in Rust and C). This work offers a practical and portable solution for cross-privilege-level and multi-process debugging in both academic and real-world OS development.**

Keywords-Operating System Debugging; GDB;

I. INTRODUCTION

In operating system (OS) debugging via external interfaces like QEMU's GDBStub [1], existing debuggers can only debug the kernel or one user program at a time. They cannot debug the kernel and multiple user programs simultaneously. Specifically, users cannot set breakpoints in a user program while the OS is paused in the kernel, or set breakpoints in the kernel or another user program while the OS is paused in a specific user program. This limitation arises because OSes enforce isolation to protect the kernel from user programs and optimize hardware usage. Two key isolation measures are: (1) the kernel and user programs run at different privilege levels, and (2) The kernel and each user program have independent contexts, especially separate address spaces. When switching between the kernel and user programs via system calls or trap returns, both privilege level and the page table change.

However, this page table switching leads to the following problem: users usually set source-level breakpoints. To set a source-level breakpoint, the debugger must load the symbol table corresponding to the source code of the breakpoint before setting it. The symbol table maps the symbols in the program's source code to virtual addresses in the program's address space. However, as mentioned earlier, the second isolation measure means that the kernel and each user program have independent symbol tables. This implies that, when setting a breakpoint, the address space of the debugged OS must match the address space of the symbol table corresponding to the user-defined breakpoint for it to function correctly. Conversely, if the user attempts to load the symbol tables of the kernel and multiple user programs simultaneously and set breakpoints in non-current isolated environments, these breakpoints are likely to fail. This will be elaborated on in Chapter III.

Since the switching of isolated environments in OSes renders symbol tables invalid and prevents setting breakpoints across isolated environments, we propose grouping symbol tables according to their respective isolated environments and grouping breakpoints based on the symbol tables they depend on. The grouped breakpoints are referred to as **breakpoint groups** [2]. Only one Breakpoint group and its corresponding symbol tables are active at a time and will be switched to another when the isolated environment changes. This ensures that the debugger only loads kernel breakpoints and symbol tables when the debugged OS is in kernel mode and only loads the breakpoints and symbol tables of a specific user-mode program when the debugged OS is in that user-mode program, thus preventing the aforementioned breakpoint setting errors.

We implement this idea through two core methods. First, we cache breakpoints and symbol tables from different isolated environments into breakpoint groups using two user-defined mapping rules. The first maps the breakpoint group name to the path of the symbol table file. When the debugger switches to a breakpoint group, it uses this rule to locate the symbol table file corresponding to the breakpoint group and switches the symbol tables. The second maps source code files to breakpoint group names. When a user sets a breakpoint, the debugger receives the filename and line number of the breakpoint. By inputting the filename into this mapping rule, the debugger determines the breakpoint group to which the breakpoint belongs and caches the breakpoint within it. If this breakpoint group aligns with the current isolated environment, the debugger not only stores the breakpoint information in the breakpoint group but also immediately sets the breakpoint.

Second, we detect isolated environment switches using breakpoints with special functionalities, enabling the switching of breakpoint groups and symbol tables during isolated environment transitions. As mentioned earlier, the debugger cannot detect isolated environment switches, causing the debugger to fail to change breakpoint groups when the debugged OS switches isolated environments, thereby failing to set and trigger breakpoints from multiple isolated environments. To address this, we designed and implemented two types of special breakpoints. The first is **hook breakpoints**, which execute user-defined commands after being triggered.

This work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62402019 and Beijing Natural Science Foundation under Grant No.4244072.

These commands read and save the context of the debugged OS at the time the hook breakpoint is triggered. The second is **exit breakpoints**, whose triggers signify an imminent isolated environment switch. For example, the system call functions of user-mode programs can serve as exit breakpoints. Before the debugging session begins, users must manually analyze the OS's source code and provide the filenames and line numbers of hook breakpoints and exit breakpoints, as well as the procedure for collecting and handling context in a configuration file. When the debugger starts, it automatically sets breakpoint groups, hook breakpoints, and exit breakpoints for the current isolated environment according to the configuration file. When a hook breakpoint is triggered, the debugger collects the context and determines which isolated environment the debugged OS will switch to after leaving the current isolated environment, thereby determining the next breakpoint group. When an exit breakpoint is triggered, the debugger switches to the next breakpoint group by executing the breakpoint group switching procedure, including consecutive single-stepping to bypass beforementioned breakpoint setting limitations (This procedure will be explained thoroughly in Chapter III).

We implemented this cross-privilege-level, multi-process OS debugging mechanism capable of functioning with real hardware and QEMU emulator based on a VSCode plugin. The implementation details can be found in our repository [3].

II. Related Work

A. Breakpoints and Single-Stepping Machenism

In the implementation of debuggers (e.g., GDB [4], LLDB), setting breakpoints is typically achieved through instruction replacement. The debugger overwrites the instruction at the target address with a special interrupt instruction (e.g., INT3 on x86, EBREAK on RISC-V). Executing it triggers an exception or trap, transferring control back to the debugger. This allows the code to pause at the specified location, enabling the debugger to inspect registers, memory, stack, and other states. However, if the breakpoint address cannot be replaced or accessed (e.g., due to differences in page table mappings), the breakpoint might fail to be set or triggered.

Single-stepping differs by not relying on instruction replacement. Instead, it uses either the processor's built-in single-step mode or the debugger's software-based single-step simulation. On certain architectures, setting the processor's single-step flag or privileged register allows an exception to be triggered automatically after every instruction, pausing execution. This approach enables the target to pause at instructions where the breakpoint setting fails to work.

B. Symbol Tables

Source-level debugging relies on debugging information files (e.g., the DWARF section in ELF [5]), which contain the symbol table. A symbol table maps symbols in the source code (e.g., function names, variable names, line numbers) to virtual addresses in the address space of the executable program. When a user sets a breakpoint at a specific line in a source code file, the debugger uses the symbol table to locate the corresponding virtual address and replaces the instruction at that address with an interrupt instruction.

III. Problem Description

This chapter details the problems in setting cross-privilege-level, multi-process breakpoints caused by page table switching during transitions between kernel and user programs. These issues motivate the Breakpoint Group Management Module and related mechanisms described in Chapter IV.

Assume a developer is debugging an OS on real hardware or a QEMU virtual machine [6]. He/she connects to the target environment's debugging interface via GDB. On real hardware, the debugging interface is the JTAG protocol-based OpenOCD interface [7], while on QEMU, the debugging interface is the built-in GDBStub. The OS is currently paused in the kernel code, and no user-mode processes have been executed yet. The developer intends to debug the first process that the OS will enter, specifically the initial process (process 0). So, he/she loads the symbol table for process 0 and enters a command in the GDB terminal to set a breakpoint in the source code of process 0. Let us call this process P_1 and the breakpoint B_1.

When GDB executes this command, it performs 2 steps. In step 1, it deduces the virtual memory address of the breakpoint from the loaded symbol table (let us call this memory address A_1). In step 2 , it modifies the instruction at the virtual memory address A_1 to a special interrupt instruction. While executing the step 2, if the kernel and user-mode program share the same page table, the virtual address A_1 points to the instruction corresponding to breakpoint B_1, regardless of whether the OS is running in kernel mode or user mode. As a result, breakpoint B_1 can be successfully set and triggered. However, if the kernel and user-mode program use different page tables, two unusual scenarios can occur: (1) The virtual memory address A_1 is only included in the page table of process 0 and not in the kernel's page table. In this case, in kernel mode, the virtual memory address A_1 is inaccessible. As a result, GDB determines that the breakpoint cannot be set at this time and marks the breakpoint as PENDING. GDB will repeatedly attempt to set this breakpoint whenever the debugged OS stops due to single-stepping or another breakpoint. (2) The virtual memory address A_1 is included in both the page table of process 0 and the kernel's page table. However, in the kernel's page table, the virtual address A_1 does not point to the instruction corresponding to breakpoint B_1, but instead points to unrelated content. Consequently, in kernel mode, A_1 does not correspond to the instruction for breakpoint B_1, and GDB modifies unrelated instructions (or possibly unrelated data) into interrupt instructions, setting the breakpoint at an incorrect physical address.

These scenarios show that since GDB does not support cross-page-table breakpoint setting, and the OS switches page tables during privilege-level transitions, GDB cannot set cross-privilege-level breakpoints. In other words, if kernel and user programs use different page tables, GDB cannot set breakpoints in both kernel and user programs simultaneously.

If the user wants to use breakpoints to debug multiple user-mode processes in the debugged OS simultaneously, GDB cannot correctly set breakpoints in different user-mode

processes, regardless of whether the kernel and user-mode program share a page table. For instance, suppose that after triggering breakpoint B_1, the user instructs GDB to set a breakpoint B_4 in another process P_4. GDB resolves the memory address A_4 corresponding to breakpoint B_4 based on the given symbol table and modifies the instruction at address A_4. However, due to overlapping address spaces among different user-mode programs, at the moment the user sets breakpoint B_4, the memory address A_4 corresponds to some instruction in the currently running process P_1, not process P_4.

In summary, when debugging an OS, GDB cannot correctly set breakpoints in memory regions managed by non-current page tables. Therefore, as the OS frequently switches page tables, GDB alone is insufficient for debugging OSes using breakpoints. The next chapter presents our solution: an external module for grouping and caching breakpoints. By detecting kernel context and isolated environment transitioning events via breakpoints with special functionalities, the module dynamically loads and unloads different groups of breakpoints and symbol tables, ensuring the effectiveness of all breakpoints set in multiple isolated environments.

IV. DESIGN

We previously analyzed the challenges of setting breakpoints across isolated environments. To address these issues, we designed a **Breakpoint Group Management Module (BGMM)** that operates between the debug GUI and GDB, intercepting users' breakpoint setting requests. It categorizes these breakpoints by their corresponding page tables, ensuring that GDB only sets the group of breakpoints corresponding to the current page table (referred to as the "current breakpoint group"). Breakpoints associated with non-current page tables are cached in the module and remain unknown to GDB.

We will then describe the breakpoint caching strategy, the method for determining the next breakpoint group, and the process of switching breakpoint groups. These procedures currently rely on user-submitted mapping rules that define the correspondence from source files to breakpoint groups to symbol table files, as well as additional breakpoints that detect transitions between isolated environments or fetch local variables that identify the next process to be executed. While this design provides the flexibility to adapt to various OSes and architectures by avoiding reliance on specialized debugging facilities, it adds configuration overhead. Future enhancements might reduce it via automated configuration mechanisms.

A. Caching Breakpoints into Breakpoint Groups

As described earlier, the BGMM can intercept user commands for setting breakpoints. Therefore, as long as the module can (1) determine which breakpoint group a breakpoint belongs to from the user's breakpoint setting request and (2) if a breakpoint belongs to the current breakpoint group, the module immediately instruct GDB to set the breakpoint, the module will have the desired breakpoint caching functionality.

Typically, user commands for setting breakpoints include the source file name and line number of the breakpoint. Hence, we require users to provide a configuration file specifying a mapping rule that associates source file names with breakpoint group names. For example, if all source files in the "initproc" folder belong to the page table of the "initproc" process, the user can map all source files under "initproc" folder to a breakpoint group named "initproc_breakpoints" in the configuration file. With such a mapping, each breakpoint can be classified into its corresponding breakpoint group.

Since the core function of this module is to ensure that only the breakpoints in the current page table are set, and we already have a mechanism to classify breakpoint setting commands into different breakpoint groups, the next step is to determine which breakpoint group is the current one.

Some ISAs (like RISC-V) do not expose a register that directly shows the current privilege level [8]. Therefore, in certain situations, we cannot directly fetch the current privilege level to determine the current breakpoint group. To address this, we add exit breakpoints to each breakpoint group. The purpose of an exit breakpoint is to indicate when the OS is about to leave the current isolated environment. For example, the exit breakpoint for the kernel breakpoint group is set in the kernel context switching code, and the exit breakpoints for a user-mode program's breakpoint group are set at the system call functions. At the beginning of a debugging session, the current breakpoint group in the BGMM defaults to the kernel's breakpoint group. Once its exit breakpoint is triggered, the module switches the current breakpoint group to the next breakpoint group. Since OSes typically run both the kernel and multiple user programs concurrently, the BGMM often contains more than two breakpoint groups. The next subsection will explain how to determine which breakpoint group the module should switch to next.

B. Determining the Next Breakpoint Group

To determine which breakpoint group is the next one, the BGMM maintains two properties: "next breakpoint group" and "next-next breakpoint group." The initial values of these properties are specified by the user in the configuration file. Typically, the user should configure the "next breakpoint group" as the breakpoint group corresponding to the initial process (process 0) and the "next-next breakpoint group" as the kernel breakpoint group (because the OS returns to the kernel after leaving process 0). For instance, when the OS is running in the kernel, the "next breakpoint group" is set to the process 0's breakpoint group. After the OS switches from the kernel to process 0, the module swaps the values of "next breakpoint group" and "next-next breakpoint group," so that before the OS returns from process 0 to the kernel, the "next breakpoint group" becomes the kernel's breakpoint group. When the OS switches back to the kernel from process 0, the values of "next breakpoint group" and "next-next breakpoint group" are swapped again, making the "next breakpoint group" the process 0's breakpoint group and the "next-next breakpoint group" the kernel's breakpoint group.

At this point, the value of "next breakpoint group" is incorrect. To address this, we use hook breakpoint during kernel code execution to retrieve the correct name of the next breakpoint group and assign it to the "next breakpoint group" variable. When the hook breakpoint is triggered, GDB

automatically executes the hook breakpoint's user-defined actions. These actions capture the correct value of the next breakpoint group. For example, the user can set a hook breakpoint in the kernel's scheduling code and define its action as fetching a string variable in the kernel's context. This string can then be processed according to user-defined rules to generate the correct value for the "next breakpoint group" variable. Therefore, after the OS returns from process 0 to the kernel, the hook breakpoint is then triggered, capturing the name of the next process. Subsequently, the "next breakpoint group" variable in the BGMM is updated from the process 0's breakpoint group to the name of the breakpoint group corresponding to the next process to run.

C. *Breakpoint Group Switching*

When an exit breakpoint is triggered, it is necessary not only to update the "current breakpoint group" variable but also to switch the breakpoint groups themselves. Since GDB cannot set breakpoints while the debugged system is running, we need to pause execution after each page table switch to change the breakpoint group and the corresponding symbol table. A seemingly straightforward approach is to set a breakpoint (temporarily referred to as B_2) at an instruction following the page table switch. When B_2 is triggered, it would indicate that the page table switch has finished, allowing the breakpoint group to be switched. However, this approach is logically infeasible because setting breakpoint B_2 would encounter the same issue as setting breakpoint B_1.

Similarly, setting a breakpoint (referred to as B_3) at the page table switch instruction or a few instructions before it is also infeasible. Although B_3 can be correctly set and triggered because its address aligns with the page table referred during its setting, triggering B_3 occurs before the page table switch. At this moment, switching to the next breakpoint group would lead to issues where setting the breakpoints in the new group would encounter the same problem as setting breakpoint B_1.

Thus, setting a single new breakpoint is insufficient to facilitate breakpoint group switching. However, it is worth noting that breakpoints B_3 and B_2 can individually complete half of the switching process: (1) if the OS pauses at the instruction associated with B_3 (as described earlier, B_3 can be successfully set and triggered because its corresponding page table matches the enabled page table when B_3 is being set), GDB can successfully unload the breakpoints of the old breakpoint group. At this moment, the enabled page table during the unloading process matches the old breakpoint group's corresponding page table, so breakpoints in the old breakpoint group can be successfully unloaded; (2) if the OS pauses at the instruction associated with B_2 (although we cannot implement this by pre-setting B_2, it can be achieved through repeated single-stepping after B_3 is triggered), GDB can successfully set the breakpoints of the new breakpoint group. At this moment, the enabled page table during the breakpoint setting process matches the new breakpoint group's page table, so breakpoints in the new breakpoint group can be successfully set and triggered.

Therefore, GDB only needs to set breakpoint B_3. When B_3 is triggered, repeated automatic single-stepping can bring the execution to the instruction corresponding to B_2. The combined pauses at these two points allow for the complete switching of breakpoint groups, avoiding the inability to pre-set B_2. When the OS is running under a specific page table, breakpoint B_3 (referred to as the "exit breakpoint" since its triggering indicates that the OS is about to leave the current page table) is set. After B_3 is triggered, GDB unloads the breakpoints of the old breakpoint group and performs continuous single-stepping until it pauses at the instruction corresponding to B_2. Then, GDB sets the breakpoints of the new breakpoint group.

Since all breakpoints stored in the BGMM are source-level, their corresponding symbol tables must already be loaded before setting them. Therefore, before setting the breakpoints of the new breakpoint group, the BGMM must locate and load the symbol tables required by the new breakpoint group. Similarly, after unloading the breakpoints of the old breakpoint group, the symbol tables corresponding to the old breakpoint group must be unloaded to avoid conflicts with the new breakpoint group. Users must specify the mapping relationships between breakpoint groups and their corresponding symbol table files in the configuration file.

V. IMPLEMENTATION

This section describes our modifications to a VSCode extension and demonstrates the implementation of the cross-privilege-level, multi-process debugging mechanism. We extended Native Debug [9], an extension originally aimed at debugging user-mode programs. Based on this extension, we added the BGMM, the hook breakpoints handling logic, the exit breakpoints and the automated breakpoint group switching. With these addons, users can set breakpoints in the kernel and multiple user processes at the same time.

A. *Overall Framework and Breakpoint Group Management Module*

Fig. 1 shows the extended VSCode extension continues to serve as a VSCode Debug Adapter [10], bridging GDB and the VSCode debug UI. Our critical modification is that the original procedure of directly forwarding user requests such as "set breakpoint" to GDB is intercepted by a new layer, the BGMM. This module uses user-defined mapping rules ("file name → breakpoint group name" and "breakpoint group name → symbol table file list") to decide whether to forward breakpoint commands to GDB and the correlation between breakpoint groups and their corresponding symbol table files.

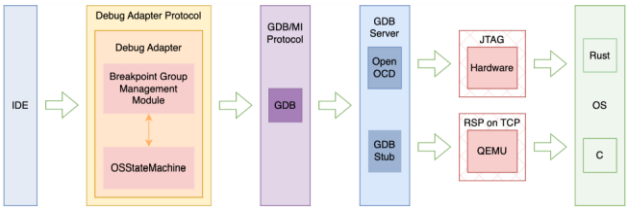

Figure 1. Overall Framework of our Implementation.

In the module, we implemented a class named Breakpoint Groups to store data of all breakpoint groups and names of the current, next, and next-next breakpoint groups. Breakpoint groups are stored in a dictionary where each group name is

associated with its corresponding list of breakpoints (and the paths to their associated symbol tables). If the user adds/deletes breakpoints during debugging, the module dynamically updates the breakpoints in the corresponding group and decides, based on the current breakpoint group, whether to immediately instruct GDB to set/delete the breakpoint.

The switching of breakpoint groups and the updating of current/next/next-next breakpoint group names rely on events triggered by exit breakpoints and hook breakpoints. To support this procedure and subsequent single-stepping after exit breakpoints, we implemented a finite state machine to manage transitions between kernel and user mode.

B. *State Machine for Debugging Context Transitions*

To do breakpoint group switching, we abstract the current isolated environment as discrete states "kernel", "user", and transitional states (e.g., intermediate states during single-stepping into user/kernel). After a breakpoint or a single-stepping-induced stop occurs, the state machine executes a series of actions based on the current isolated environment state and the occurred event. Actions include "unloading the current breakpoint group", "loading the next breakpoint group" and "doing a single-step." Fig. 2 summarizes the main logic for state enumeration, event enumeration, and state transitions:

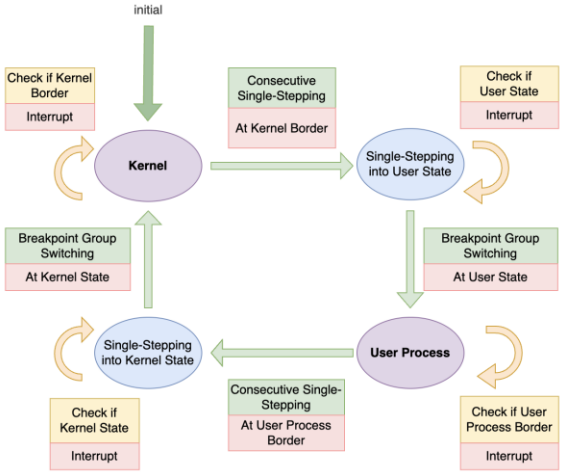

Figure 2. State Transitions and Actions for Managing Breakpoints Across Isolated Environments.

For transitions from kernel to user processes, hook breakpoints are set during kernel or "kernel single step to user" states to update the "next breakpoint group" variable. This ensures that when the OS switches to a user process, the correct breakpoint group is loaded.

C. *Examples and Experiments*

We verified the feasibility of our cross-kernel/user-mode, multi-process debugging tool in various OSes on QEMU and real hardware. Table 1 lists some of the scenarios currently tested, including educational OSes like rCore-Tutorial [11], xv6 [12], and Linux-compatible OSes like Starry [13]. To port our tool to other architectures or OSes, it is sufficient to modify exit breakpoints, hook breakpoints, "file name → breakpoint group name" and "breakpoint group name → symbol table file list" mappings in the configuration file.

TABLE I. TESTED OSES AND DEBUGGING ENVIRONMENTS

OS	Debug Interface	Architecture
rCore-Tutorial (Rust)	QEMU GDBStub	RISC-V
xv6 (C)	QEMU GDBStub / VisionFive2 with JTAG and OpenOCD	RISC-V
Starry (Rust/C Hybrid)	QEMU GDBStub	RISC-V

VI. CONCLUSION

We propose a cross-privilege-level, multi-process OS debugging method. It introduces a Breakpoint Group Management Module to group and cache breakpoints and dynamically switches to the correct breakpoint group when the debugged OS transitions between isolated environments (e.g. kernel and process). This approach resolves conflicts in setting breakpoints at different privilege levels and multiple processes. Additionally, we set up exit breakpoints to enable the module to detect isolated environment transitions in the debugged OS and perform automatic breakpoint group switching. Finally, hook breakpoints were used to retrieve identifiers of the next process to be executed, facilitating simultaneous debugging of multiple user processes. Using this method, we successfully set and triggered breakpoints in both the kernel and multiple user programs in various OSes on QEMU and real hardware, demonstrating its portability.

REFERENCES

[1] H. Li, Y. Xu, F. Wu, and C. Yin, "Research of 'Stub' remote debugging technique," in Proc. 2009 4th Int. Conf. Computer Science & Education, pp. 990–994, 2009.

[2] Z. Chen, Y. Yu, Z. Li, and J. Wu, "An online debugging tool for Rust-based operating systems," 2022.

[3] Z. Chen, "Code Debug." [Online]. Available: https://github.com/chenzhiy2001/code-debug

[4] R. Stallman, R. Pesch, S. Shebs, and others, "Debugging with GDB," Free Software Foundation, vol. 675, 1988.

[5] DWARF Debugging Information Format Committee, "DWARF debugging information format version 4," Tech. Rep., June 2010.

[6] F. Bellard, "QEMU, a fast and portable dynamic translator," in Proc. USENIX Annu. Tech. Conf., FREENIX Track, vol. 41, no. 46, pp. 10–5555, 2005.

[7] H. Högl and D. Rath, "Open on-chip debugger—openocd," Fakultat fur Informatik, Tech. Rep., 2006. [Online].

[8] Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual volume II: Privileged architecture version 1.7," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-49, 2015.

[9] J. Jurzitza, "Native Debug." [Online]. Available: https://github.com/WebFreak001/code-debug

[10] Microsoft, "Debug Adapter Protocol." [Online]. Available: https://microsoft.github.io/debug-adapter-protocol

[11] Y. Wu, "rCore-Tutorial-v3." [Online]. Available: https://github.com/rcore-os/rCore-Tutorial-v3

[12] R. Cox, M. F. Kaashoek, and R. Morris, "Xv6, a simple Unix-like teaching operating system," Sept. 5, 2013. [Online]. Available: http://pdos.csail.mit.edu/6.828/2012/xv6.html

[13] Y. Zheng, "Starry-OS." [Online]. Available: https://github.com/Starry-OS/Starry