Design and Implementation of a Remote Debugger for Concurrent Debugging of Multiple Processes in Embedded Linux Systems *

Jung-hee Kim[1], Hyun-chul Sim[1], Yong-hyeog Kang[2], and Young Ik Eom[1]

[1] School of Information and Communication Eng., Sungkyunkwan University
300 cheoncheon-dong, Jangan-gu, Suwon, Gyeonggi-do 440-746, Korea
{kimjh, jlmaj, yieom}@ece.skku.ac.kr
[2] School of Business Administration, Far East University
5 San Wangjang, Gamgok, Eumseong, Chungbuk 369-851, Korea
yhkang@mail.kdu.ac.kr

Abstract. In the embedded software development environments, developers can concurrently debug a running process and its child processes only by using multiple gdbs and gdbservers. But it needs additional coding and messy works of activating additional gdb and gdbserver for each created process. In this paper, we propose an efficient mechanism for concurrent debugging of multiple remote processes in the embedded system environments by using the library wrapping mechanism without Linux kernel modification. Through the experimentation of debugging two processes communicating by an unnamed pipe in the target system, we show that our proposed debugging mechanism is easier and more efficient than preexisting mechanisms.

1 Introduction

Currently, the gdb has been popularly used as a remote debugging tool in the embedded Linux software developments. By running the gdb in the host system and the gdbserver in the target system, developers can debug a remote process running in the target system [1][2]. However, developers must insert a "sleep" function into the debugged program in order to concurrently debug a newly created child process of the current debugged process. Developers also need additional gdbserver in the target system and connect it to the blocked child process. In the host system, additional gdb is required to connect to the new gdbserver in the target system. Therefore, developers must have the same number of gdbs in the host system and gdbservers in the target system as the number of the debugged processes.

A gdbserver in the target system provides developers with the ability of debugging a process by using ptrace system call in the Linux systems. But the ptrace system call needs the parent-child relationship between a gdbserver and

* This work was supported by Korea Research Foundation Grant (KRF-2003-041-D20420).

a debugged process [3]. When a debugged process creates a new process, the parent-child relationship is not established between a gdbserver and the newly created child process. Developers need to insert the sleep code into a newly created child process code. When the newly created process is blocked by the sleep code in the target system, developers run a new gdbserver in the target system and connect it to the blocked process. Developers also run a new gdb in the host system and connect it to the new gdbserver in the target system. When two connections are established, developers can debug the newly created process in the target system by using the gdb in the host system and the gdbserver in the target system.

2 Our Proposed Mechanism

In this paper, we propose a new debugging mechanism that supports concurrent debugging of multiple remote processes by using the mgdb library and the mgdbserver. Fig. 1 shows the overview of our proposed mechanism that supports the concurrent debugging of multiple remote processes. The mgdbserver in the target system communicates with the gdb in the host system. Developers can concurrently debug multiple remote processes by selecting the process intended to debug at desired time by using the mgdbserver. Whenever a debugged process creates a new child process, the mgdbserver runs a new gdbserver in the target system and connects it to the newly created child process automatically in order to support the concurrent debugging of the newly created child process.

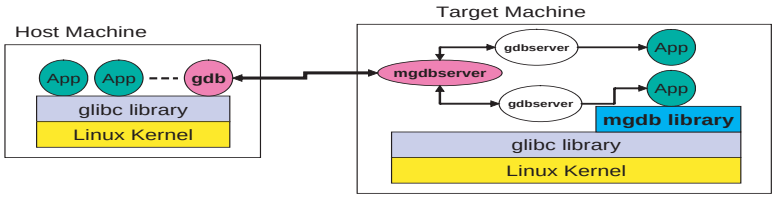

Fig. 1. Overview of our concurrent and remote debugging mechanism

In order to support concurrent debugging of multiple remote processes, the mgdbserver must know when the current debugged process invokes fork system call. In this paper, we use the mechanism of wrapping the glibc library in order to intercept the system call that a currently debugged process invokes. When the currently debugged process calls the function in the glibc library, the library wrapping scheme intercepts the function call and calls the same name function in our mgdb library. The called function in our mgdb library executes the code that is needed for debugging of multiple processes before calling the function in the glibc library that is intended to be called originally. In order to intercept system call, we use the interposition mechanism of Linux dynamic linker [4] by preloading our mgdb library before the glibc library.

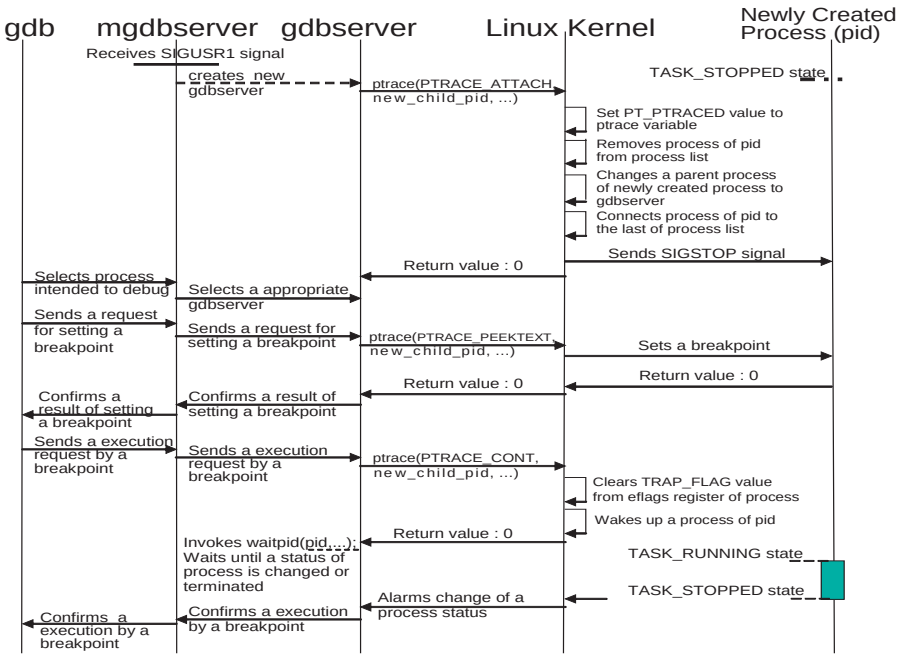

Fig. 2. Flow of debugging newly created process after mgdbserver receives signal

When the currently debugged process creates a new child process, our mgdb library blocks the newly created process in order to prevent the process from terminating. It also informs the mgdbserver that the currently debugged process creates a new child process by the signal. The mgdbserver runs a new gdbserver and connects it to the newly created process. As shown in Fig. 2, when the mgdbserver receives the signal from our mgdb library, it creates a new gdbserver.

In order to become the parent process of the newly created process, the new gdbserver sets the PT_PTRACED value to a ptrace variable of the newly created process by invoking ptrace system call. When developers want to change the currently debugged process, they can select the process to debug by using a gdb in the host system. When the new debugged process is selected by a gdb, the mgdbserver passes debugging request from the gdb to the gdbserver.

3 Experiment and Performance Analysis

The scenario used in the experiment is as follows. The parent process creates a child process and sends a string through the unnamed pipe shared by the child process. After the child process receives the string from the parent process, it rearranges the string in reverse order and sends the string to the parent process through the unnamed pipe. After the parent process receives the string, it prints the string sent by the child process.

Developers can see the debugged status information of all processes created by the currently debugged process by typing "show-remoted-debugee" at the "gdb" prompt in the host system. By selecting the process identifier number, they also can change a specific process for debugging through "change-remote-debugee" command with "pid" argument.

Table 1. Comparison of TotalView with our mgdb library and mgdbserver

	ETNUS's TotalView	Our proposed debugger
Test program image size	53658 bytes	23973 bytes
Library linking mechanism	staic	dynamic
Remote debugging	no supporting	supporting

In this experiment, we focus on the ability of our debugger tool to support concurrent and remote debugging of the parent process and the newly created child process by selecting the process intended to debug using only one gdb in the host system. As shown in Table 1, we compare our proposed scheme with ETNUS TotalView program that supports debugging of multiple processes [5]. However, the library linking mechanism in ETNUS TotalView supports only static linking, therefore the size of the debugged program in ETNUS TotalView is larger than that in our proposed scheme. ETNUS TotalView also cannot support remote debugging.

4 Conclusion

In this paper, we presented a new concurrent debugging mechanism for remote processes through the design and implementation of the mgdb library and the mgdbserver. In our proposed scheme, developers can debug all debugged processes in the target system by selecting the debugged process among them through one gdb in the host system. Compared with the preexisting mechanism, our proposed scheme provides easier and more efficient concurrent debugging for multiple remote processes in the target system.

References

1. Daniel Jacobowitz, Remoting Debugging with GDB, http://www.kegel.com/linux/gdbserver.html, 2002.
2. Richard M. Stallman, Debugging with GDB, 4th ed., Cygnus Support, 1996.
3. Uresh Vahalia, Unix Internals, Prentice Hall, 1996.
4. Sun Microsystems Inc., Linker and Libraries Guide, October, 1998.
5. Etnus, Totalview Getting Started, http://www.etnus.com/Products/TotalView/started/getting_started2.html, 2001.