ON REDUCING OF CORE DUMP FILE SIZE

Sergey S. Grekhov[1], Jaehoon Jeong[2], Mikhail P. Levin[1], *Member IEEE*

Abstract: **In this paper a solution to minimize the size of core dump in computing machinery with limited resources is proposed. The proposed technique describes how to reduce some sections in core dump and yet still keeps the ability to debug with this reduced information on embedded systems.**

Index Terms**: Core dump file, Embedded Systems, Debuggers, Computing Machinery with Limited Resources**

I. INTRODUCTION

This paper is dedicated to storing information of program crash reasons on systems with restricted resources.

During debugging and testing, software programs can crash. In this case the kernel of operating system could create a so- called core dump file – a snapshot of a program state at the moment of program crash [1]. This information usually helps the developer to identify the source of the crash problem. The core dump feature is designated primarily for servers and workstations equipped with abundant amounts of memory and data storage. In contrast, embedded systems usually have a small volume of memory and often do not have additional data storage for storing core dump files. Therefore the kernel of operating system can't make a core file. But the presence of the core dump feature is as well necessary for the developers of embedded systems since it makes the debugging process more efficient and rapid.

To solve this problem we need to reduce the size of core dump to make it usable under limited resources in embedded systems. A special care is necessary to preserve compatibility with the well-known Open Source GDB debugger's [2] "back trace" feature as a main tool to investigate the reason of the crash. Another solution is to redirect normal core dump to some external storage such as USB-flash. Such solution was proposed in [3]. Unfortunately in this case embedded systems should be equipped with such an additional external storage and special tools to store dump files on it. This increases the price of embedded systems and increases the complexity of systems. Therefore in this paper we only consider the first solution.

In [4], the system and the method for creating of core dump files of minimal volume were proposed. In this approach, the system creates a complete crash dump at the moment of crash which is, typically, a dump of all physical memory mapped to the application in the machine. After that the stand alone extraction tool can be used for obtaining a mini crash dump that consists of references to real memory regions in physical memory, thread data structure, process data structure, stack data structure (processor registers of kernel), loaded module list data structure, and memory management information block. Therefore implicitly the size of core dump remains the same: one part of original core dump file is stored in the new core dump file and another in physical memory. Obviously, this method is not acceptable for embedded systems with restricted resources because it needs a lot of memory to store the initial dump file and, thus, the mini crash dump file can not be created. Also it should be mentioned that usual debuggers can not read core dump files modified by [4]. Therefore in this case debuggers also should be modified to read such core dump files.

In contrast with [4], the proposed method in this paper decreases the size of core dump because as will be shown below the decreased core dump file contains only sections mostly important for post mortem debugging. For embedded systems with restricted resources this method is adjustable because it also allows debuggers to read decreased core dump files and provide debugging on embedded systems without any modifications of debuggers.

II. GENERAL DESCRIPTION OF IDEA

The system and method of creating crash (core) dump files under conditions of systems with restricted resources allow storing the reason of program crash. The kernel of operating system that supervises the embedded system should save only the most important information describing the crashed program and reject storing extra memory regions. This approach allows to give the developers ability to debug the crashed program without increasing necessary memory in embedded systems.

To solve the above mentioned problem, the following method of creating of core (crash) dump files is proposed. The kernel of operating system in the case of program crash should save the following information about the state of the crashed program:

1) the images of kernel variables, used for

[1] Advanced Software Group, Samsung Research Center in Moscow

[2] OS Group, Samsung Advanced Institute of Technologies

execution of crashed program;

2) the reference information about memory region that includes the contents of the stack of the crashed program;
3) the memory region that includes the contents of the stack of the crashed program.

Now let us specify some detailed conditions of our investigation for the considering problem. We consider Linux kernel as the example of kernel. As an example of the embedded system the ARM-processor hardware system [5-6] is taken and as an example of crash core dump file the Linux core dump file is taken. The well-known Open Source GDB-debugger [2] is taken into consideration as an example of debugger.

Usually running application uses only the user space. But in the crash case the core dump generation unit uses data from both user and kernel spaces. For the kernel space, the snapshots of the operating system variables (used for managing the application process) and the CPU state are created. For the user space, the snapshots of memory areas used by application are created. All these snapshots are included into the core dump file in the crash case. Here we should take into account that process state data in the crash moment includes the content of CPU registers and the stack data. This classification allows suggesting the method to create a mini core dump file. Since we want to minimize the volume of core dump file, we must determine in the initial core dump file which information should be omitted. Our main goal in creating of mini core dump file is to keep back-trace properties in debugging. Therefore the CPU register data and back-trace data must be stored in the mini core dump file but the data from user space can be omitted. By this manner the mini core dump file is generated.

Extraction of data included into the mini core dump file is only the first stage of the minimization problem solution. On the second stage let us consider another minimization problem, namely: how to minimize volume of mini core dump file. To solve this minimization problem we suggest reduce the size of the memory area containing stack back-trace of the crashed application. To do this it needs at first to provide the alignment of the memory area containing stack back-trace required by the operating system. At second it needs to find the pointer to the beginning of the stack back-trace of the crashed application, using the CPU state snapshot. At third it needs to decrease the memory area with the alignment value, by moving the start address of the memory area to the direction of the stack pointer as many times as possible.

III. ANALYSIS OF STRUCTURE OF CORE DUMP FILES

To allow developers to use core dump in debugging of embedded systems we propose to generate the decreased core dump files which, at same time, keep the ability to use it for bug tracking. In this paper, we suggest the method of generation of mini core dump file that allows to use in debugging. This method holds the format of core dump file and also allows examining the stack of crashed application in debugger.

The core dump file has a standard ELF format (see [7-9] for details of ELF format). The structure of core dump file in ELF format is shown by the diagram presented on Fig.1.

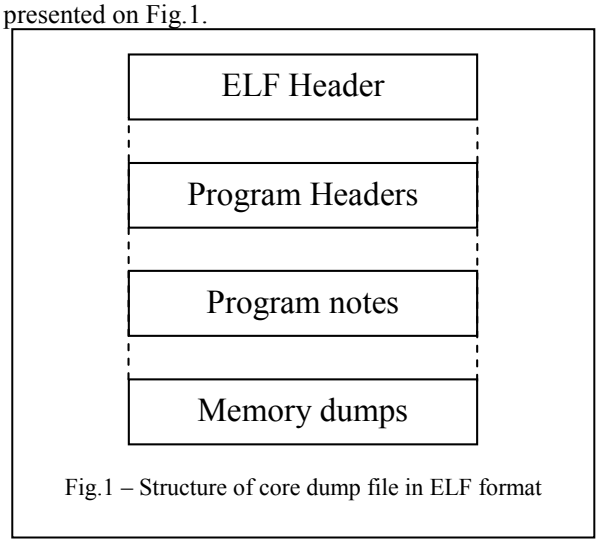


Fig.1 – Structure of core dump file in ELF format

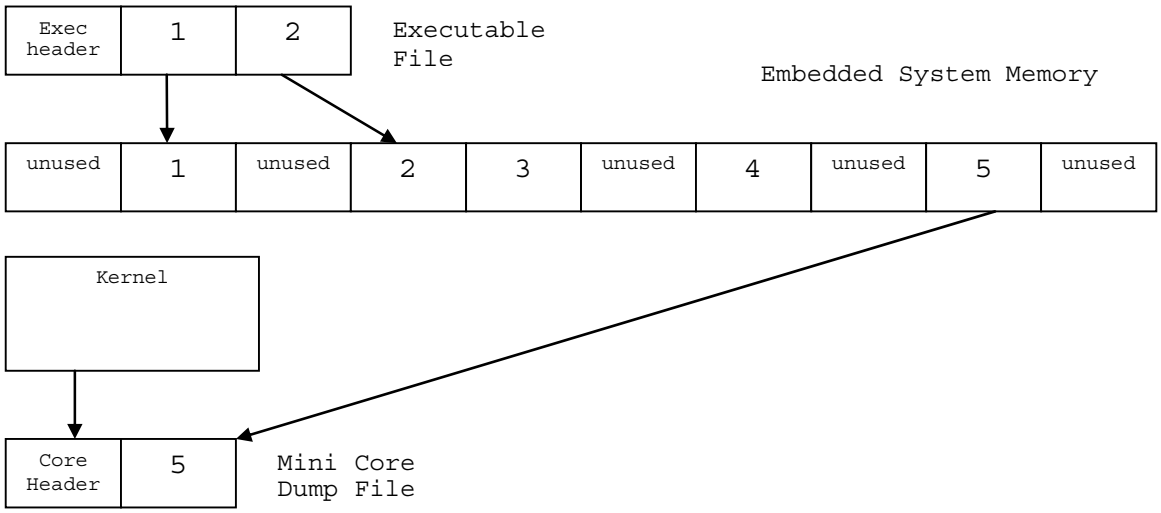
It has the following sections:

- ELF Header describes the common format parameters and basic info about contents (number of various sections, etc.);
- Program headers describe the parameters of the Program notes and Memory dumps;
- Program notes contain the images of the system variables that control the program execution in kernel;
- Memory dumps contain images of stack and data sections with local variables of the crashed program.

If an application has only one thread, then the biggest part of the core dump file is usually the memory dumps and, particularly, the areas with local variables (here local variables mean variables that were declared in the function body, but not the function input parameter which is stored in the stack).

In case of a multithread application, the situation is more complex. The standard initial stack size for each additional thread is about 2 Mbytes. Thus, if local variables are relatively small – the biggest part of the memory section will be the thread stacks.

The following diagram shown on Fig.2 explains the idea of how to reduce the size of core dump file.



Section number	VMA (example)	VMA size (example)	Description (example)
1	0040 0000	1000h	.text
2	1000 0000	9000h	.data
3	1000 9000	26000h	.bss, heap
4	2AAA 8000	10000h	extra data
5	7FFE 3000	15000h	stack

Fig.2. - Scheme of idea for mini core dump file generation

Each executable file as shown on the diagram below consists at least of the header section, the text section (1), and the data section (2). Typical locations of these sections in the address space are defined by virtual memory address (VMA) and the appropriate size (VMA size) as it is shown in the table presented on this diagram. The executable file runs on the embedded system that is supervised by a kernel. When a process crashes, the kernel creates snapshots of kernel variables that were used for controlling the process (see Embedded System Memory diagram on Fig.2). These kernel variables are stored in Core Header section of mini core dump file (see Fig.2). Also in this file the data from stack section (5) are stored. Sections (2-4) are omitted and don't include in the mini core dump file.

IV. THE METHOD OF CORE DUMP FILE SIZE REDUCING

To decrease core dump file we suggest storing Core Header section in mini core dump file. In the original core dump file after the header section usually follows sections in which snap shots of memory areas used by considering program. Among these sections there is a special section where the stack content is stored (see section (5) in Embedded System Memory structure on Fig.2). We suggest keep only this section in the decreased core dump file as shown in Mini Core Dump file structure on Fig.2. Thus, all other memory sections except (5) are discarded in the decreased Mini Core Dump file. This discarding operation and the process of the mini core dump file creation are provided by a special modification of kernel.

The main idea of reduction is to cut out some section from the 'Memory dumps', particularly, to leave only the stack back-trace of the crashed thread (if a program has several threads).

Let us assume that the program local variables are relatively big or the program has several threads. Then let us consider Algorithm #1 allowing us to reduce the size of core dump file. It is as follows:

Algorithm#1

1. Extract the stack pointer register value from the 'Program notes' section for the crashed thread;
2. Find the header of the memory region that holds this stack;
3. Save ELF header, header for 'Program notes' and header for the found memory region;
4. Save memory region with stack.

With Algorithm#1, the GDB-debugger will be able to execute the "back trace" command and print out the stack of the called functions, but will not be able to print the local variables except functions' input parameters.

The more complex situation is when the stack of the crashed thread is the biggest part of the 'Memory dumps'. Here it could be possible to cut some extra memory from the stack region: for example, the thread uses about a half of the memory area and the other part is unused. Then let us formulate Algorithm2 as an improved version of Algorithm #1 as follows:

Algorithm#2:

1. Extract the stack pointer register value from the 'Program notes' section for the crashed thread;
2. Find the header of the memory region that hold this stack;
3. Reduce the size of memory area (if possible) in the header by moving the start address of the memory region to the stack pointer by step equals to the size of the memory region alignment;
4. Save ELF header, header for 'Program notes' and header for the found memory region;
5. Save reduced memory region with stack.

Our experiments showed that the decreasing of storage volume can reach up to 80% (the percentage of cut out data from original core dump). Usually about 75% of reducing is obtained by discarding of memory areas and about 5% of reducing is obtained by decreasing the memory region volume containing the call stack. These estimations are based on suggestion that usual codes are using the most volume of mapped memory for storing of local variables. The typical application for embedded systems, which has about 100 threads and require about 100 Mbytes of RAM, produces the "minimal" core dump file with 80-82% of decrease. The special programs used in testing were designed, according to the structure of reduced core dump file, to check different ways of generating huge core dump files.

Descriptions of tests programs used in our tests are presented in Table 1.

Table 1. Description of test programs

Program	Description
Test program #1	The code of this test application contains a lot of local variables, particularly, several arrays of integers.
Test program #2	The code of this test application includes a lot of recursive calls of local functions.
Test program #3	The code of this application includes generating several independent threads , each thread allocates a huge array of data.

In result of discarding operation, the size of core dump file is significantly decreased and can satisfy the rigid memory restrictions of embedded systems and simultaneously keeps ability to use this file in debugging. Although the usage of such files has some certain restrictions, the developer is able to observe only call stack tree and variables that were saved in stack. Nevertheless, it allows find the source of program crash by more effective manner than without using of core dump features in debugging. Moreover it is not necessary to increase the memory of embedded systems to keep the ability to provide debugging with core dump features on these systems.

All modifications of Linux kernel code for creation of reduced core dump files were implemented as a special patch. After applying of this patch to Linux kernel it should be recompiled with a special option which allows create core dump files of decreased size. If in the embedded system there is not enough disk storage for storing core dump file, the modified kernel writes as much data as possible on the disk and breaks the storing process if/when the limit is reached. The modified kernel also provides ability to create so-called "reliable" core dump files. In these files the memory regions containing stacks are stored at the beginning of the dumped memory areas list. This allows reduce the probability of loosing the call stack tree in case when there is not enough disk storage to save full core dump file.

V. ACKNOWLEDGEMENT

Authors would like to thank Mrs. Jamee Lee, Leader of OS Group, Samsung Advanced Institute of Technologies, and Mr. Sang Bae Lee, Senior Engineer of Samsung Advanced Institute of Technologies, for fruitful and extensive discussion on the proposed approach and help.

REFERENCES

[1]. Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel*, 3rd Edition, O'Reilly, 2005.
[2]. Richard M. Stallman, Roland H. Pesch and Stan Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, 9th Edition, Free Software Foundation, 2002.
[3]. Masahiro Kiyoi, Hirofumi Nagasuka, Masaya Ichikawa, Akira Otsuji, *Method and apparatus for getting dump of a computer system*, Patent US7010725, March 7, 2006.
[4]. Andre F. Vachon, *Creation of mini dump files from full dump files*, Patent US6681348, January 20, 2004.
[5]. *ARM Architecture Reference Manual*, David Seal, ed., Addison-Wesley, 2000.
[6]. Andrew Sloss, Dominic Symes and Chris Wright, *ARM System Developer's Guide: Designing and Optimizing System Software*, Morgan Kaufmann, 2004.
[7]. Eric Youngdale, *The ELF Object File Format: Introduction*, Linux Journal, Issue 12, April 1995. (http://www.linuxjournal.com/article/ 1059)
[8]. Eric Youngdale, *The ELF Object File Format by Dissection*, Linux Journal, Issue 13, April 1995. (http://www.linuxjournal.com/article/ 1060)
[9]. *ARM ELF File Format*, ARM DUI 00101-A, ARM Limited, 1998. (http://www.arm.com/pdfs/ DUI0101A_Elf.pdf)

Sergey S. Grekhov received MS degree in Computer Sciences from Novosibirsk State University in 2004. He joined to Samsung Research Center in Moscow in 2005. His research interests include design and analysis of efficient algorithms, embedded systems, Linux Kernel development.

Jaehoon Jeong received MS degree in Electronic engineering from Hanyang University, Seoul Korea, in 2001. He joined to Samsung Software Lab. in 2001. His research interests include System Software design and development for embedded systems, Linux Kernel BSP and new kernel feature development for CE Product.

Mikhail P. Levin received MS degree in Mechanics with Honor from Moscow Power-Engineering Institute (State Technical University) in 1978, PhD degree in Physics and Mathematics from Computing Centre of USSR Academy of Science in 1984, and Senior Research Scientist (Associate Professor) degree from Computing Centre of Russian Academy of Sciences in 1993. From 1980 till 1994 he jointed to Computing Centre of Russian (USSR) Academy of Sciences as a research staff. In 1994-2000 he was a principal software developer at German-Russian Joint Venture EuroSoft GmbH. He jointed to Korea Advanced Institute of Science and Technology (KAIST) from 2000 till 2001 as a professor at the Departments of Mathematics and Aerospace Engineering. In 2001-2003 he jointed to deCODE Genetics, Inc., Reykjavik, Iceland as a research staff. In 2003-2005 he jointed to General Energy Technologies Ltd. as a principal researcher in mathematical modelling. In 2005-2006 he jointed to Kraftway Corporation PLC as a leading staff member in High Performance Computing. He jointed to Samsung Research Center in Moscow in 2006. His research interests include design and analysis of efficient algorithms, embedded systems, Linux Kernel development, numerical methods and optimization of systems with distributed parameters, parallel algorithms and high performance computing. Dr. Levin has published more than 80 research papers and books on computer sciences, applied mathematics and numerical methods.

.