An Improved RTEMS Supporting Real-Time Detection of Stack Overflow

Rui Zhang[1,3(✉)], Yan Du[2,3], Tao Zhang[1,3], Qi Qiu[1,3], Liang Mao[1,3], and Jiaxiang Niu[1,3]

[1] Beijing Institute of Spacecraft System Engineering, Beijing, China
299542@qq.com
[2] Institute 706, Second Academy of China
Aerospace Science and Industry Corporation, Beijing, China
[3] Science and Technology on Communication Networks Laboratory, Shijiazhuang, China

Abstract. Aiming at the common problem of stack overflow in satellite software, this paper improves the RTEMS operating system which is supporting real-time stack use depth and overflow detection. Taking the on-board software based on TSC690F processor as an example, the accessible area and unaccessible area are set for each thread stack by using the memory access protection mechanism provided by the processor. The improved RTEMS shared the access protection mechanism among threads through context switching. A trap handler is designed to take over write protection error traps, calculate stack usage depth, and monitor stack overflow in real time. The core module performance test and stack detection instance verification show that the improved RTEMS has little effect on the software performance, so that the software can detect the stack depth online and real-time. By using this method, the software is still manageable in case of stack overflow, rather than runaway crash, and the reliability of the software is improved.

Keywords: Improved RTEMS · On-board software · Stack used depth · Stack overflow

1 Introduction

RTEMS (Real Time Executive for Multiprocessor System) is a multi-processor embedded real-time operating system which was developed in the 1980s [1]. It was a free open source software supported by the U.S. military and first applied to the missile control system. RTEMS has the characteristic of good real-time and stability, and it is widely used in the field of aerospace. On-board software is a typical embedded software. With the support of RTEMS operating system, on-board software runs stably on space-borne equipment. The Aerospace Orbit Control System (AOCS) software and On-Board Data Handling (OBDH) software are the two typical on-board embedded software. The OBDH software is mainly responsible for the data management, telecommand and telemetry service for the satellite [2]. The AOCS software mainly

M. Jia et al. (Eds.): WiSATS 2019, LNICST 280, pp. 283–293, 2019.
https://doi.org/10.1007/978-3-030-19153-5_29

controls the attitude of satellites. This two software is classified as the key level software, and its function is very critical to the satellite.

The stack is one of the most important resources in the software running process. It not only stores the parameters and return address of the calling function, but also stores the local variables of the function. The usage of the stack depends on the running software which is dynamic, then it is not easily for the verification whether the allocation of the stack is enough for the software running. Of course, the overwhelming allocation will result in the waste of on-board computer resources.

For stack depth detection, static test method and dynamic test method are usually used [3, 4]. Static testing methods are generally supported by specialized tools, such as Stack Analyzer tools developed by AbsInt. This tool is not widely used, because of the cost, and the analysis process does not support function pointers, recursive logic analysis and so on. Dynamic testing method is to test the software stack during system running. For example, the RTInsight tool developed by Shanghai Chuangjing Company can be used for dynamic testing of the stack after matching the interface of the target system. Dynamic testing methods has higher requirement for the design of test case. If the deepest function call path on the stack cannot be analyzed, even if the program runs for a long time, the maximum use depth of the stack cannot be obtained, and the stack overflow problem cannot be avoided during the running of the software. StackGuard and StackSheild are widely used in stack overflow protection technology [5]. The principle is to add some code to each function to create a "guard" when it is called, and compare when it is returned. Because each function call and return must run these codes, the efficiency of software operation is greatly affected. It is difficult to satisfy the real-time requirement of on-board software.

In summary, the use of on-board software stack has the following problems. ① Designers unable to obtain the actual use of stacks and margins, and always create the stack space by experience. Generally, it is a margin insufficiency or waste. ② Stack depth detection is difficult. Static testing is more dependent on tools, and dynamic testing is more dependent on test case design. Both of them are unsatisfactory. ③ software cannot be prevented before stack overflow, and it cannot provide alert alarm or protection measures. ④ After the stack overflow, the running position of the program when the software crashes are generally not near the overflow code, and the hysteresis characteristic of the fault makes it difficult to solve the problem [6, 7]. ⑤ The behavior after software collapse is unpredictable, and there is a risk of secondary failures. In addition, it is also difficult to design targeted protective measures.

In view of the above problems, we have improved the RTEMS. The improved RTEMS has the ability of real-time stack overflow detection. It mainly supports SPARC series processors. This paper introduces the implementation principle of RTEMS which supports real-time stack overflow detection, taking the TSC695F processors commonly used in spaceborne equipment as an example. The principle is also applicable to the SPARC series processors such as TSC697 and BM3803.

2 Introduction of TSC695F Processor

2.1 Access Protection Mechanism

The Rad Hard 32-bit SPARC Embedded Processor (TSC695F), ERC32 Single-chip, is a highly integrated, high-performance 32-bit RISC embedded processor implementing the SPARC architecture V7 specification. It has been developed with the support of the ESA (European Space Agency), and offers a full development environment for embedded space applications.

TSC695F processor can be programmed to detect and mask write accesses in any part of the RAM [8]. The programmable write access protection is segment based. A segment defines an area where write cycles are allowed. Any write cycle outside a segment is trapped and does not change the memory contents. Two segments are implemented. Each segment is implemented with two registers: The Segment Base Register and the Segment End Register. The segment base register contains the start address of the segment, and enabling bits for supervisor/user mode (SE/UE). The segment end register contains the first address outside the segment, i.e. last address of segment plus one word. The start address of segment is a low memory address than the end address. Only word aligned addresses are supported. The segments are only active during RAM access, i.e. they can only be mapped to the RAM area. There are two modes of the access protection. In normal mode, a memory exception is generated only if both segments indicated a write protection error. In block protect mode, a memory exception is generated if any of the segments indicate a write protection error. Fig. 1 is shown difference of two mode. The memory exception will force the IU to vector to a data access exception, then the IU enters the corresponding trap (trap type 9).

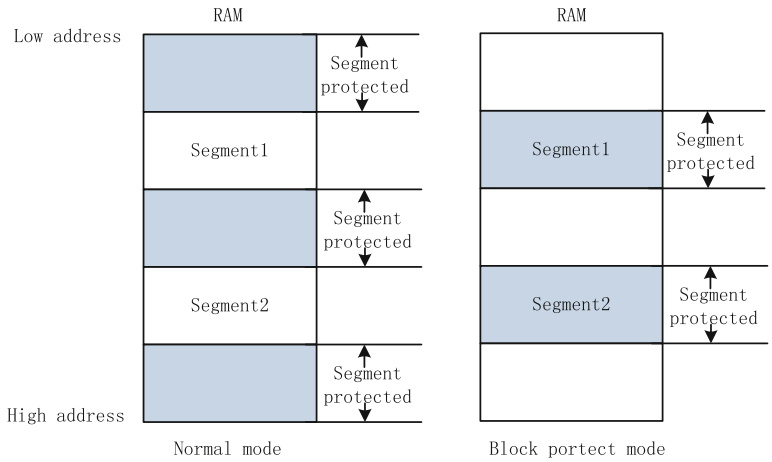

Fig. 1. Access protection mechanism

2.2 Stack Introduction

When satellite onboard software is running, the code and data are divided into text section, data section and bss section. The remaining space in memory is allocated to the heap space and stack space, which are generally uniformly allocated and managed by the operating system. Figure 2 is a schematic diagram of the memory allocation of the satellite onboard software.

TSC695F is widely used in spacecraft because of its EDAC (Error Detection and Correction) protection mechanism. The EDAC detects any two bits error and corrects any single bit data error on the 40-bit bus. The processor has eight sets of window registers, each of them consists of eight input registers (i_0–i_7), eight local registers and eight output registers (o_0–o_7). The input and output registers are mainly used to pass parameters to the function and receive the return value of the function. The output register o6 saves the stack pointer (sp), which points to the top of the current stack. Therefore, the output register can pass 6 parameters at most, and more parameters shall be passed through the stack. The input register i6 keeps the heap pointer (fp), which points to the bottom of the current stack. The stack consists of heap pointer and stack pointer, and the stack grows from high address to low address direction. The stack space occupied by each function is called a stack frame. For example, function A calls function B, function B then calls function C, and their stacksare shown in Fig. 3.

With the support of the operating system, the functions of on-board software are usually divided into several threads. The stack of each thread is allocated from the stack space in Fig. 2. In this paper, the start address of stack is a high memory address than the end address of stack, because stack grows direction shown in Fig. 2. When the stack of one thread overflows, it might overwrite the stack of another thread which will cause an error when another thread ready and runs.

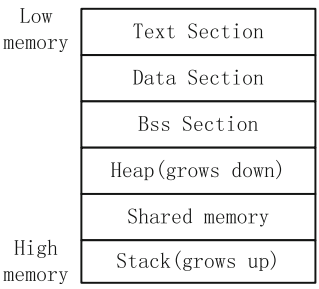

Fig. 2. Memory map of on-board software

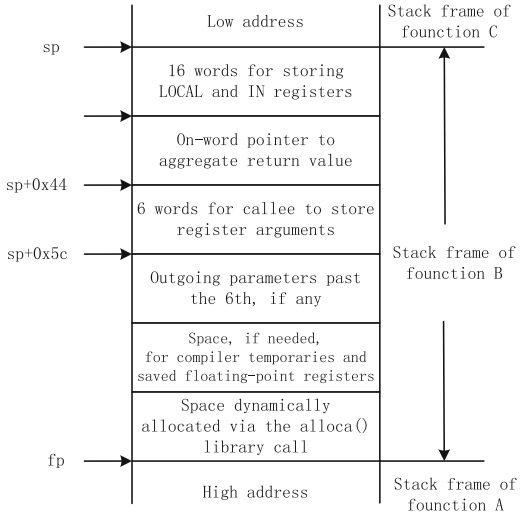

Fig. 3. Stack of TSC695F

3 Improvement and Implementation

3.1 Principle Introduction

Considering the problems discussed above, this paper proposes a method which protects the stack of different threads and provide the detection of the stack depth and overflows. This method bases on the memory access protection mechanism of processor. The improved RTEMS use the block protect mode of memory access protection mechanism.

The detection principle is: adding the definition of access protection register as variable in RTEMS thread control block (TCB), that is, expanding the definition of thread context, so that a single thread can exclusively use the access protection register of the processor when it is running. Different threads share the access protection mechanism of the processor by switching the thread context. The thread stack is divided into the initial accessible and unaccessible areas during initialization. With the growth of the stack, when the stack exceeds the initial accessible area and attempts to write data to the unaccessible area, the processor will produce a trap (type 0x9) immediately. By hooking the corresponding trap handler, the software resets the thread unaccessible stack area to extend the accessible area. This method can meet the stack growth and make sure the software keeps running normally. The software can calculate the depth of the current stack according to the expansion of the accessible area. As the stack grows, when the software triggers the trap and finds that the unaccessible area is less than or equal to 1 K bytes, it is assumed that the thread stack is about to overflow. At this time, the software can design specific recovery measures from stack overflow, such as thread restart, software reset, switch to backup computer work, etc. Figure 4 shows an example of the growth of thread stack changes using this RTEMS.

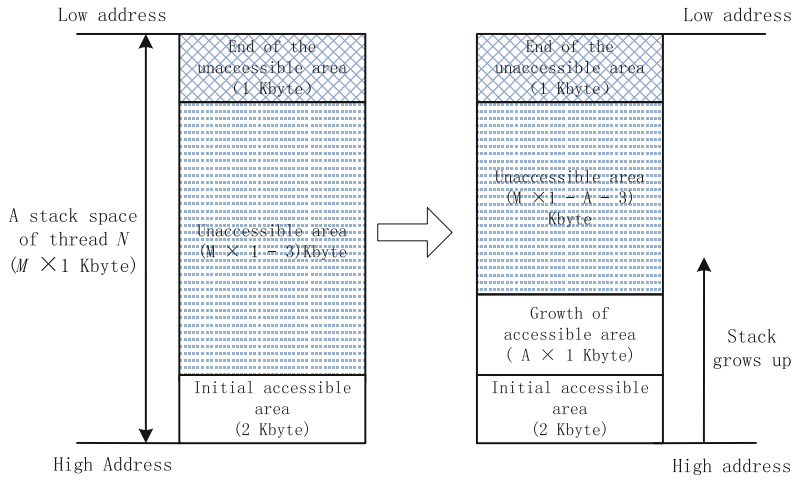

Fig. 4. An instance for growth of stack

The above detection principle requires the following improvements to RTEMS:

(a) Modify the definition of the thread control block (TCB) and add access protection registers as variables in thread context.
(b) Modify the stack initialization module of RTEMS (function _Stack Initialize). After the thread stack is created, the initial accessible area and unaccessible area of the thread stack are calculated and set. The start and end addresses of the unaccessible area are saved as thread context in the thread control block.
(c) Modify the thread context switch module of RTEMS (function _Thread Dispatch) to switch the write protection area of the stack when different threads are running. When context switching occurs, the value of the current access protection register is stored in the suspended thread control block, and the value recorded in the TCB of the thread to be run is written to the access protection register.
(d) Add the corresponding trap processing module to achieve the re-setup of the unaccessible area of the stack, and detect the stack depth and process the overflow in real time.

3.2 Stack Initialization Module

When a thread is created, the size of its stack is generally need to be specified. After RTEMS allocate the stack to the thread, the initial start address and depth of the stack are recorded in TCB. It can also be assigned with other unit lengths. This paper takes 1 Kbyte as an example to illustrate.

The stack space of a thread is divided into the following 2 parts, as shown in the left half of Fig. 4.

(a) Accessible area: the initial size is 2 Kbyte. This area is a stack area allowed for normal operation by software. With the growth of the stack, the scope of the area can be increased gradually. The maximum accessible space equals the stack allocation space minus 1 K bytes. The start address of this area is equal to the start address of stack. the end address is a low memory address than the start address.
(b) Unaccessible area: the scope of this area is located at the end of the accessible area to the end of the stack. The end address of the stack is used as the start address of the unaccessible area which would be set to the segment base register when thread is running. And the end address of the accessible area is used as the end address of the unaccessible area which would be set to the segment end register. As the stack grows, the area becomes smaller and smaller. When the unaccessible area has been minimized (1 K bytes), a write protection trap occurs and the stack is thought to be overflowing. The processing of traps is carried out according to stack overflow.

When the stack initializes, the start address and end address of the unaccessible area are recorded to TCB.

3.3 Context Switch Module

When the software runs, the CPU switching to another thread needs to save the state of the current thread and restore the state of another thread, which is called context switching. Context switch includes saving the running environment of the current thread and restoring the running environment of the thread that will run. In RTEMS, the running environment of the thread includes window register, floating point register, PC counter and so on. In improved RTEMS, segment base register and segment end register are added to the context definition of threads.

The module has very little modification, only a small amount of assembler code needs to be added. When context switching occurs, the value of the current access protection register is stored in the suspended thread control block, and the value recorded in the TCB of the thread to be run is written to the access protection register. In this way, different threads can share processor access protection mechanism.

3.4 Trap Handler

After all threads have been created, the software starts running normally. As the stack grows gradually, and its use depth extends to low addresses. When it exceeds the accessible area (initial 2 Kbyte), it enters the unaccessible area. Due to the existence of processor access protection mechanism, the program running on the unaccessible area cannot operate write operations. Any write operation will immediately trigger processor exceptions, resulting in a 0x9 trap. The flow of trap handler is shown as Fig. 5.

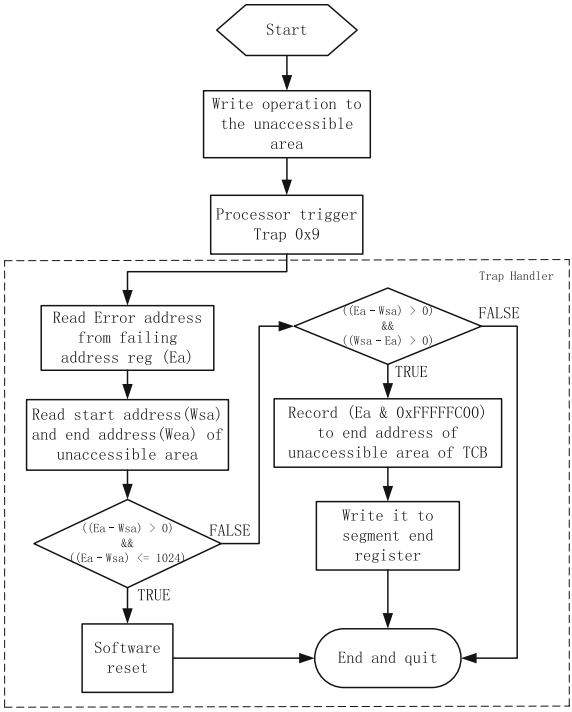

Fig. 5. Flow of trap processing

After the software enters the trap handler, it gets the error address of the trap by reading the failing address register. Comparing the error address with the access protection address range set by the current processor determines whether it is in a thread stack area. If the error address falls within the scope of a thread stack unaccessible area, the error address is compared with the start address of the unaccessible area for the corresponding thread. If the error address minus the starting address of the unaccessible area is less than 1 Kbyte, it is considered the thread stack will overflow soon. If it is greater than 1 Kbyte, the stack is considered to grow normally. The following operations are performed for this error:

① Logical AND operation is performed for the error address and 0xFFFFFC00, and the result is used as the end address of unaccessible Area and saved to TCB, which enlarges the accessible area and makes the wrong address in the accessible area.
② Write the end address of the unaccessible area to the processor write protection end register.

3.5 The Use Depth of Stack

While the software runs, the depth of use of each thread stack can be calculated by accessing the unaccessible protection area end address of each thread (The stack depth equals the stack start address minus the unaccessible protection area end address, accurate to 1 Kbyte). Stack occupancy equals the stack depth divided by the allocation depth, multiplied by 100%.

4 Improvement and Implementation

The performance and accuracy of the improved RTEMS are tested and verified on the target hardware platform. The target system processor uses TSC695F platform and the main frequency is set to 10 MHz. The performance of initialization module, context switching module and trap processing module are tested respectively. The test results are shown in Table 1. From the performance test results, it can be seen that the delay of initialization module and context switching module modified by this method almost does not increase. The processing delay of the new trap processing module is also very small, so the processing time of 30us has little effect on the system performance.

Table 1. Module performance test

Module		Performance(us)
Initial stack	RTEMS	4.89
	Improved RTEMS	9.12
Context switch	RTEMS	72.32
	Improved RTEMS	76.27
Trap handler	New	30.15

Taking an on-board software of the integrated electronic system as the test object, the stack usage of eight threads created by the improved RTEMS is tested, and the test results are compared with those using dynamic testing tools. The test results are shown in Table 2.

Table 2. An instance for stack used depth detection

Thread id	Start address of stack	Stack space (Kbyte)	Dynamic test tools		Improved RTEMS	
			Use depth (Kbyte)	Margin (%)	Use depth (Kbyte)	Margin (%)
1	0x023eb858	32	8.564	73.2	9.000	71.8
2	0x023e1708	10	1.886	81.1	2.000	80.0
3	0x0243ea30	20	4.004	80.0	5.000	80.0
4	0x0243bf98	10	1.742	82.6	2.000	80.0
5	0x023e45a0	6	0.704	88.3	2.000	66.7
6	0x023f6b00	8	1.124	86.0	2.000	80.0
7	0x02444450	8	2.450	69.4	3.000	62.5
8	0x0244ff90	12	3.670	69.4	4.000	66.7

In the above test case, thread 5th is selected to illustrate how thread stack overflow can be detected, through calling the following C function.

```
void foo(void)
{
inttest_array[1200];
test_array[1199] = 0;
return;
}
```

The trap of the test program may suspend the stack overflow thread and print out the memory address of the trap, program pointer by the Serial port. The results of the above test case are shown in Fig. 6. The program pointer address is 0x02015488 which points to the function foo. The trap was trigger at memory address 0x023E3020, which is the address is the last 1 Kbyte of the thread 5th and cannot be accessed by the thread. As soon as the trap was triggered, the thread 5th will be suspended.

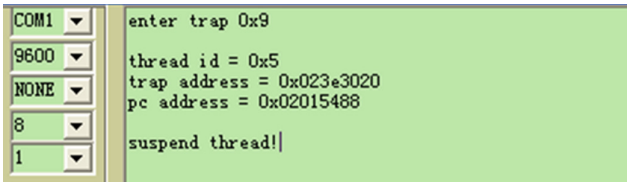

Fig. 6. Output of test result

The test results show that the stack depth can be obtained by using the improved RTEMS and accuracy is 1 Kbyte. The accuracy is not as high as that measured by dynamic testing tools, but it can meet the requirements for stack margin observation. In addition, this method is the same as most dynamic testing methods, and measuring the maximum depth of stack usage depends on the test case. But different from this method, the stack usage depth can be obtained online and real-time, and the key information (including the overflow address, thread ID and program counter) can be recorded before the stack overflow, which is very important for the subsequent fault detection.

5 Conclusion

Satellite On-board software is difficult to sense the growth process of the stack during its operation. It is not sure whether the usage margin is sufficient, and also it is difficult to allocate the stack overflow fault. This paper improves the RTEMS by using the processor memory access protection mechanism, and proposes a real-time stack overflow detection method. Taking the on-board software running on the TSC695F processor as an example, the software is set by setting the access protection area in the stack, and it has ability to sense stack dynamic growth and calculate stack size in real time. By checking the minimum scope of the access protected area, the improved RTEMS have the ability to forecast the stack overflow in advance. The improved RTEMS has the advantage that operation system can obtain the stack depth of every thread (accuracy to 1 Kbyte) in real time; the principle is simple and easy to implement, and the software can detect online and in real time after the implementation, and the performance of improved RTEMS is almost unchanged. With the RTEMS improved in the paper, once the stack overflow happens, the trap handler will take over the following up task, which effectively eliminate the uncertainty of software behavior after stack crash, greatly simplify the design of system fault handling countermeasures and improve software reliability.

References

1. Li, H., Yin, C.: Analysis and improvement of RTEMS memory management. In: CONFERENCE 2009 First International Workshop on Education Technology and Computer Science (2009)
2. He, X., Sun, Y.: Engineering realization of software in central terminal unit of satellite data management system. J. Spacecr. Eng. **16**(5), 47–53 (2007)
3. Kuperman, B.A., Brodley, C.E.: Detection and prevention of stack buffer overflow attacks. J. Commun. ACM **48**(11), 51–56 (2005)
4. Dong, Z., Hou, C., Guo, J., et al.: Dynamic detection method of spacecraft software process stack used depth. J. Spacecr. Eng. **26**(1), 85–90 (2017)
5. Cao, Y., Wang, Y.: An overview of the stack protection techniques in the GCC compiler. J. Inf. Technol. (7), 23–25(2017)

6. Pan, Q., Wang, C., Yang, Y.: Analysis and prevention of the stack overflow attacking. J. Shanghai Jiaotong Univ. **36**(9), 1346–1350 (2002)
7. Sun, H., Xu, L., Yang, H.: The principle and detection of buffer overflow attack. J. Comput. Eng. **27**(1), 127–128 (2001)
8. ATMEL Corporation: TSC695FSPARC 32-bit Space Processor User Manual