Maciej BORZĘCKI, Bartłomiej ŚWIERCZ, Andrzej NAPIERALSKI
DEPARTMENT OF MICROELECTRONICS AND COMPUTER SCIENCE, TECHNICAL UNIVERSITY OF ŁÓDŹ

SEU – zjawisko pomijane przez współczesne systemy operacyjne

mgr inż. Maciej BORZĘCKI

W 2006 ukończył studia magisterskie na specjalnosci Telecommunications and Computer Science prowadzonej przez Politechnikę Łódzką. Obronił z wyróżnieniem pracę magisterską pt. „Timing Definition Language for POSIX environment". Aktualnie jest dotorantem prof. Andrzeja Napieralskiego w Katedrze Mikroelektroniki i Technik Informatycznych Politechniki Łódzkiej, gdzie wspólnie z dr Bartłomiejem Świerczem zajmuję się odpornością na błędy w środowisku wieloprocesorowym

e-mail: mborzecki@dmcs.pl

dr inż. Bartłomiej Świercz

Adiunkt w Katedrze Mikroelektroniki i Technik Informatycznych Politechniki Łódzkiej oraz specjalista od oprogramowania mobilnego i wbudowanego w Teleca Poland. W 2008 roku obronił z wyróżnieniem doktorat ze specjalności systemy operacyjne czasu rzeczywistego. Zainteresowania naukowe to systemy operacyjne, systemy czasu rzeczywistego oraz platformy mobilne i wbudowane.

e-mail: swierczu@dmcs.pl

prof. Andrzej NAPIERALSKI

Kierownik Katedry Mikroelektroniki i Technik Informatycznych na Politechnice Łódzkiej, którą kieruje od momentu jej powstania. Dwukrotnie wybierany na prorektora PŁ. Łączny dorobek naukowy to ponad 820 pozycji w tym 4 książki. Jest promotorem 36 zakończonych rozpraw doktorskich. Wieloetni członek Komitetu Elektroniki i Telekomunikacji PAN oraz przewodniczący Sekcji Mikroelektroniki KEiT PAN.

e-mail: napier@dmcs.pl

Streszczenie

W zależności od przeznaczenia systemu operacyjnego, projektanci skupiają się na jednej z kluczowych cech: wydajności, determinizmie czasowym lub niezawodności. Często projektanci systemów operacyjnych uwzględniają kilka z przytoczonych cech jednocześnie, lecz zazwyczaj zapominają o niekorzystnym wpływie otoczenia na sprzęt elektroniczny. Celem artykułu jest omówienie wpływu jednego z czynników zewnętrznych, jakimi są promieniowanie neutronowe lub kosmiczne, na pracę systemów operacyjnych. Zamiarem autorów artykułu jest przedstawienie programowego algroytmu ochrony systemów przed błędami oraz omówienie możliwości implementacji algorytmu na przykładzie jądra systemu Linux.

Słowa kluczowe: SEU, odporność na błędy, systemy operacyjne, zarządzanie pamięcią

SEU - the phenomenon omitted by modern operating systems

Abstract

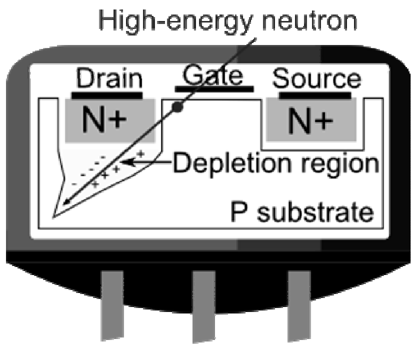
Modern operating systems are expected to provide one of the key features: performance, meeting time constraints or reliability. Sometimes, the operating systems designers may embed a mix of the listed features, but very few of them are aware of the adverse influence of the environment. In this paper, neutron radiation and cosmic rays are considered as the external factors. A software method of countering the environment induced errors is presented, together with a discussion of the implementation possibilities based on the Linux kernel.

Keywords: SEU, fault tolerance, operating system, memory management

1. Introduction

Interaction of neutron radiation on the electronics systems has been known since 1954 when atomic weapon tests were performed. The same influence on electronics is observed in cosmic space due to cosmic radiation. Moreover, cosmic rays are able to affect electronic devices on the ground level, what is the key of given paper. The interaction phenomena, called Single Event Upset (SEU) [1] is shown on the Fig. 1. A neutron crossing through Metal-Oxide-Semiconductor (MOS) transistor generates pairs of electron-holes due to indirect ionization process [2]. If the neutron looses enough energy, such that critical charge can be accumulated in proximity to drain, transistor is switched on and as consequence, element such as memory cell based on MOS transistors changes its state to reverse one.

Described occurrence is observed logically, as a bit-flip in memory and depending on the memory cell address, the effects can be numerous: erroneous calculation result, incorrect jump operation or even a critical error which may render the system unusable.



Rys. 1. Struktura tranzystora MOS z zaznaczonym wpływem promieniowania neutronowego
Fig. 1. The structure of MOS transistor affected by the neutron radiation

Radiation sensitivity of the semiconductors devices (such as MOS transistors) is increasing as a result of technology progress measured by physical transistor dimension. The increased susceptibility of modern integrated circuits to radiation effects has been noticed by industry leaders such as Cisco, who claim:

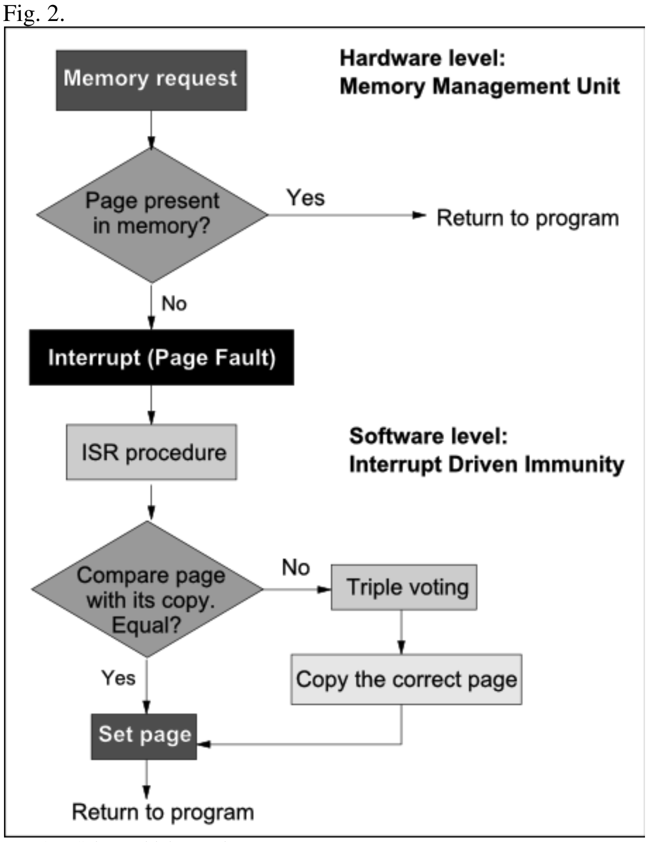
"All future designs that require highest availability must counter unavoidable SEUs."

The paper is divided as follows. Section 2 provides introduction to SEU tolerance algorithm implemented in software at the level of operating system. Section 3, due to the nature of described method, discusses the memory management subsystem of the Linux kernel. The subsequent sections describe future work and highlight main conclusions.

2. SEU Tolerant Operating System

Radiation hardening at hardware level is the most intuitive way to protect sensitive electronic equipment against Single Event Upset. Given the use of specialized hardware, little care needs to be taken at the software level. However such level of convenience is very costly. Hardware based methods are used commonly in space systems but, due to the overall costs, this approach can not be directly used in general purpose applications such as e-business servers or in data centers. Should one want to use commercial-of-the-shelf (COTS) components, software based methods are the most viable solution [3]. Yet again, modification of each application to support detection and correction in case of memory corruption may incur a high cost. There is one exception however. Should the operating system, as being closer to the hardware and having a complete view of the memory, be modified to detect this class of errors, other software such as regular applications, would instantly gain a seamless and virtually cost-free protection. Research conducted by authors demonstrated that it is possible to design operating system capable of detecting and correcting SEUs in transparent way to the running applications, with little runtime overhead. A novel algorithm, called Interrupt Driven Immunity (IDI), was derived. It is operation is briefly shown in Fig. 2. The initial implementation was done in the sCore kernel and positively verified in accelerator tunnels located in DESY research center in Hamburg [4], where highly successful results were obtained. In final experiment, the test board (Device Under Test – DUT) was left for 14 days and 8 hours. During that experiment 12186 SEUs ware observed and any one of noticed SEUs had an influence on running applications.

The SEU tolerant algorithm utilizes the memory paging mechanism provided by Memory Management Unit (MMU) – inseparable part of modern processors. The paging mechanism allows the operating system to implements virtual memory, resulting in better, overall memory utilization [5]. The IDI is based on the idea of memory redundancy. However, a number of extra steps are required during operating system startup. The memory pages used by the operating system are copied into two redundant memory regions. Each time, a new task or a new memory request is serviced a copy needs to be done as well. Each page has a number of status bits, 'present' bit being one of them. Zero value means that the page might have swapped out (possibly to secondary storage). Program trying to access the contents of such page will cause the MMU to generate a page fault (hardware interrupt). IDI utilizes this behavior for providing SEU memory protection as show in Fig. 2.



Rys. 2. Schemat blokowy algorytmu IDI
Fig. 2. The block scheme of IDI algorithm

Each page, upon being released is marked as 'not present'. The next access will cause a page fault, what triggers IDI to perform memory comparison with both redundant copies. In case of no errors, the page is marked as present and the program may continue. Should errors occur, the triple voting mechanism would determine what is the page contains the correct data and correct the error by means of copy operation. When program is preempted by system scheduler, 'present' bit of all pages used during one system epoch is cleared and set as 'not present'. The total overhead of IDI algorithm is less than 20% in comparing to the analogues system without IDI and SEU protection.

The use if IDI algorithm adds very little runtime overhead, thus no significant degradation of performance is observed. The faulting memory page is checked only once during given scheduler period, which can usually be controlled at the level of configuration of given operating system.

3. Overview of Linux memory management

This section briefly describes the topic of memory management in the Linux kernel referring to the IA-32 architecture [6].

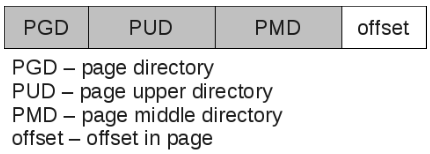
Given the context of this paper, the particular method of realization of memory protection as described in previous section, the main goal of analysis presented here, is to evaluate the suitability of Linux kernel as a target platform for porting the aforementioned method to.

For this purpose, the most recent (as of the time this paper is written) release of Linux kernel is used, namely 2.6.28 available from [7]. The Linux kernel supports a significant number of different architectures, thus for simplification, IA-32 architecture is further assumed to be used. Although only one processor architecture is considered, the kernel code is highly flexible, and only architecture specific changes will be required when moving to a new platform.

Keeping in mind the relatively scarce documentation, the great deal of information can be extracted by directly browsing the code of the kernel.

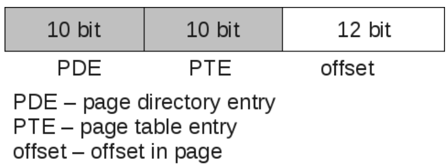
IA-32 provides two main mechanisms for memory management: paging and segmentation [8]. The Linux kernel makes very limited use of segmentation, mainly due to portability reasons. The paging mechanism, being widely available among different processor architectures, is the prevalent mechanism to deliver virtual memory and swapping functionality

The Linux kernel needs to support architectures of varying address width. For the best compromise between portability and efficiency, the internal paging mechanism can have up to 4 levels. The address structure is split into parts as show in Fig. 3.



Rys. 3. Prezentacja adresu pamięci w jądrze Linux
Fig. 3. Memory address representation in the Linux kernel

The simplest case, that the kernel can be configured to use is a 2-level paging, which mirrors the IA-32 address representation. This case is presented in Fig. 4.



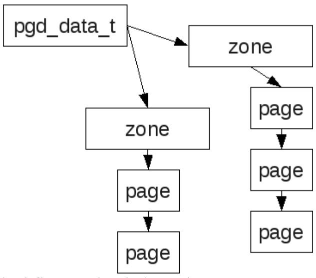
Rys. 4. Najprostszy przypadek reprezentacji adresu na architekturze IA-32
Fig. 4. Simplest case of address representation on IA-32 architecture

The kernel does not directly use the processor's data structures, but rather through a number of convenience macros, required manipulation is performed.

Each page is represented by corresponding data structure named `struct page`. It's size, although in general architecture dependent, on IA-32 is often 4 kB but also could be 2 and 4 MB [8].

The Linux memory manager splits the available physical memory into zones. Each zone acts as a memory pool, out of which new pages can be retrieved when needed. Zone contains a per-CPU memory map with a linked list of page descriptors, each corresponding to a chunk of memory. The top level structure, `pg_data_t`, and zone descriptors are always resident in memory at a well known address, thus they can be easily localized from within the kernel.

The general relation between physical memory descriptor structures is shown in Fig. 5.

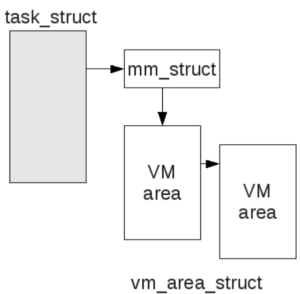


Rys. 5. Podzial pamięci fizycznej w jądrze Linux
Fig. 5. The fragmentation of physical memory in the Linux kernel

The IDI assumes that there are a number of redundant copies of particular page. Layered approach to memory management in the Linux kernel and the isolation of zones from the page management code executed in memory handling routines during the regular flow, may give an advantage when implementing redundant memory area mappings as hooks to zone handling code.

The memory of each task needs to be protected from SEU, thus the mechanism of correlating virtual memory mapping with given process needs to be investigated. The kernel assigns and tracks the memory of a running process by use of `mm_struct` data structure, referenced from `task_struct`. The `mm_stuct` contains a reference to a linked list of virtual memory areas, named `vm_area_struct`, that are used by given process. The relations between these structures are shown in Fig 6.

Linux, similar to other Unix-like kernels, provides the user with ability to map a file into task's address space, thus virtual memory mapping area may correspond to the data present on a permanent storage.



Rys. 6. Struktury przypisujące obszary pamięci wirtualnej do procesu
Fig. 6. Data structures used for assignment of virtual memory areas to a process

During the lifetime of a process, at a given instant in time, there may be more virtual memory areas assigned to it than the number of pages that are needed to cover the complete memory range. This is caused by mechanisms such as delayed allocation or copy-on-write [9], which allow for improvements in operating system performance. In case of an access to the memory that would have been in such page, a page fault is generated by the host processor.

Recall, that for the successful implementation of IDI, page fault handling is essential. Thus it is vital to place the implementation hooks carefully, so that the portability is not lost, just in case a need for supporting more architecture appears and performance of Linux kernel is still sufficient.

The actual code that performs handling of a page fault event is architecture independent and enclosed in two main functions: `handle_mm_fault` and `handle_pte_fault`. The action preformed by page fault handling code is dependent on the context and the state of virtual memory mappings for given task. IDI protection hooks added in this code (`handle_pte_fault` in particular) would bring the benefit of memory protection of a significant number of architectures.

Having in mind the general idea behind the operation of an IDI algorithm, one needs to address the last problem, ie. marking the pages as not present upon switching the task this can be easily addressed in the Linux scheduler. The relevant data structures related to memory assigned to given process were described in previous paragraphs.

4. Future Work

The authors plan to perform an initial implementation of IDI algorithm in the Linux kernel, with the idea of concentrating on IA-32 architecture. Moreover it will be interesting to evaluate the suit-ability of other operating system kernels such as K42 [10]. The modified Linux kernel is planed to be tested using software emulator such as Bochs and using embedded PC placed inside one of accelerators tunnels located in DESY.

The IDI algorithm was designed to protect systems running on COTS computers with single processor. Authors would like to evaluate the performance of IDI in multiprocessor systems as well as focus on developing algorithms that combine both memory and processor redundancy. The main architectures to be evaluated are Symmetric Multiprocessing (SMP) and Massively Parallel Processors (MPP).

5. Conclusion

As presented in this paper, the problem of radiation induced errors becomes more significant with the ongoing race for producing smaller, low-voltage processors. It appears that the easiest way to provide a transparent protection to user applications is to implement algorithms like IDI at the level of an operating system, which in fact acts as a wrapper, isolating the actual application form the hardware.

Algorithms like IDI are a viable solution to memory protection, their implementation cost may vary depending on platform, however the only requirement is that the MMU is available, what is a common case for contemporary processors. Basing on the example of Linux kernel, porting IDI to other operating systems is not overly complicated, thus it will be possible to deliver a decent level of radiation protection to a large user base.

6. Literatura

[1] L. Adams, A. Holmes-Siedle: Handbook of Radiation Effects, Oxford University Press, 2004
[2] Actel: Effects of neutrons on programmable logic, White Paper, 2002
[3] D. Makowski, B. Świercz, M. Grecki, and A. Napieralski. Projektowanie systemów niewrażliwych na wpływ promieniowania na potrzeby akceleratora X-FEL. Elektronika - Konstrukcje, Technologie, Zastosowania, 2005
[4] B. Świercz: The Algorithms for Protection of Operating Systems with Special Emphasis on the Neutron Radiation, Ph.D dissertation, 2008
[5] V. Abrossimov, M. Rozier, M. Shapiro: Generic Virtual Memory Management for Operating System Kernels, SOSP, ACM, 1989
[6] M. Gorman: Understanding the Linux Virtual Memory Manager, Prentice Hall, 2004
[7] Linux kernel source code: http://www.kernel.org
[8] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide
[9] M. Acetta, Mach: A New Kernel Foundation For UNIX Development, USENIX, 1986
[10] J. Appavoo, M. Auslander and all: K42 Overview, Kernel white paper, http://www.research.ibm.com/K42/white-papers/Overview.pdf

Artykuł recenzowany