The technique of locking memory on Linux operating system - Application in checkpointing

1st Xuan Huyen Do
Information Technology Dept.
University of Sciences, Hue University
Hue, Vietnam
doxuanhuyen@gmail.com

2nd Viet Hai Ha
Office for STIC
University of Education, Hue University
Hue, Vietnam
haviethai@dhsphue.edu.vn

3rd Van Long Tran
Information Technology Dept.
Hue Industrial College
Hue, Vietnam
tvlong@hueic.edu.vn

4th Éric Renault
SAMOVA, Télécom SudParis, CNRS
Institut Polytechnique de Paris
Évry, France
eric.renault@telecom-sudparis.eu

***Abstract*—Checkpointing is the technique that saves the images of a process at a point during its lifetime, and allows it to be resumed at the saving's time if necessary. It can be classified into two types: full checkpointing and incremental checkpointing, in which the second type is based on the memory lock mechanism. This paper aims to present two main memory lock techniques working in both kernel and user space and a performance comparison when applying them in checkpointing for the Checkpointing Aided Parallel Execution (CAPE).**

***Index Terms*—Checkpointing, virtual memory, memory lock, CAPE, Checkpointing Aided Parallel Execution**

I. Introduction

Checkpointing is a simple technique for rollback recovery: the state of an executing program is periodically saved to files (called checkpoints) from which it can be recovered after failure [4] - [6]. Thus, there is no need to spend time on initializing and executing it from the beginning. These techniques were introduced two decades ago. Nowadays, they are researched and used widely on fault-tolerance, applications trace/debugging, roll-back/animated playback, and process migration.

Creating a checkpoint involves saving the process state which in turns includes registers, virtual address space, open files, debug registers etc., and using these data to restart the process [1]. With the full storage of process states - full checkpointing technique - at different times, it creates big checkpoints that waste resources. Alternatively, in the incremental checkpointing technique, only the data which have changed comparing to the previous checkpoint are saved.

The key of incremental checkpointing is how to detect the changed data on the process memory. In order to do this, most of the approaches detect the memory pages of the process which being changed during the process execution. Most of the checkpointing techniques often apply page-based techniques for this. There are also other techniques such as hash-based techniques, proxy-based techniques. Page-based techniques require memory-protection hardware and support from the operating system (OS) to identify dirty pages [4] - [6] [16] [17] [25]. Hash based techniques [7], use a hash function to compare and identify the changed blocks of memory. Proxy-based techniques [18] [20], hook the memory access of the program to control the memory access of process.

In page-based technique, the process memory pages are marked as read-only (lock for write-protected). When a write-demand access to these memory areas, the OS will generate a SIGSEGV signal, then the checkpointer receives this signal, set the state of the related memory page to writable (unlock this page) so the write-demand can be performed, and marks this page as a changed (dirty) page. The task of memory lock/unlock can be implemented in kernel space or in user space. In kernel space, we can approach by: (i) monitor the dirty bit in the page table entry to user level by using a kernel patch [17] [25]; or (ii) using additional driver (no need to patch/rebuilt kernel) [16]. In the user space, it is possible to (iii) lock the process memory with the help of the functions provided by a high-level programming language or library support [4] to control the changed memory regions.

This paper focus on the incremental checkpointing approach for the detection of modified data pages (ii) and (iii). The rest of the paper is organized as follow: Section II presents related work. The next Section - the main part of this paper – present in details the techniques of locking/unlocking memory at the kernel and user levels. Section IV is an experiment of comparing the time of checkpointing using two lock/unlock techniques at kernel and user levels. The last Section is the conclusion and future works.

This material is based upon work supported by the MOET under Grant No. TN-265 Any opinions, findings and conclusions or recommendations expressed in this publication do not necessarily reflect the views of any organization or agency that provides support for this research.

II. RELATED WORK

A. The operating principle of the operating system based on kernel space and user space

Linux is a multitasking operating system introduced by Linus Torvalds in 1991. The Linux kernel is released under GNU General Public License version 2 (GPLv2) [9]. Linux uses a monolithic kernel architecture with all operating system works in kernel space [11]. A set of primitives or system calls implement all operating system services such as process management, concurrency, and memory management. Device drivers can be added to the kernel as modules.

User space is interpreted as code outside the operating system's kernel. Usually it is applications or libraries that run on the operating system through interacting with the operating system kernel. Each process run in its own virtual memory, and cannot access the memory of other processes. With enough privileges, processes may require the kernel to map a part of another process's memory space to its own, such as in the case for debuggers. Programs can also request shared memory regions with other processes. Most hardware platforms (including x86 and x86-64) and most operating systems have multiple levels of privilege, often called privilege rings. The x86 and x86-64 architecture has four privilege rings numbered from 0 to 3, as depicted in Fig.1. Linux uses the privilege rings mechanism, only use rings 0 and 3 for supervisory and user privileges, respectively [12]. In order for a ring 3 user application to execute privileged operations, such as requesting I/O or memory management services, the application must make a system call into the operating system kernel, where carefully vetted ring 0 supervisor-level kernel code executes the privileged request on behalf of this application.

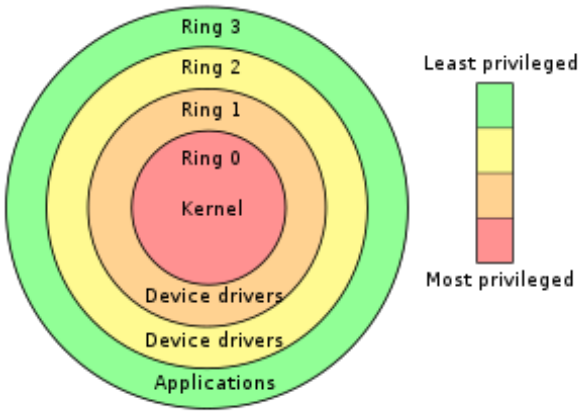

Fig. 1. Operating system privilege rings in most modern hardware platforms and operating systems.

B. Mechanism of process memory management in Linux

When a program needs to be executed, it is loaded onto the main memory by the operating system and it becomes a process. Linux manages the memory according to the page, the size of each page on x86_64 architecture is 4 KB. On 32-bit operating systems, each running program is allocated in a virtual (logical) memory of 4 GB, in which 1 GB for kernel space and 3 GB for user space. This virtual memory is organized into a series of continuous memory pages with the same size, as shown in Fig.2 [26]. When a

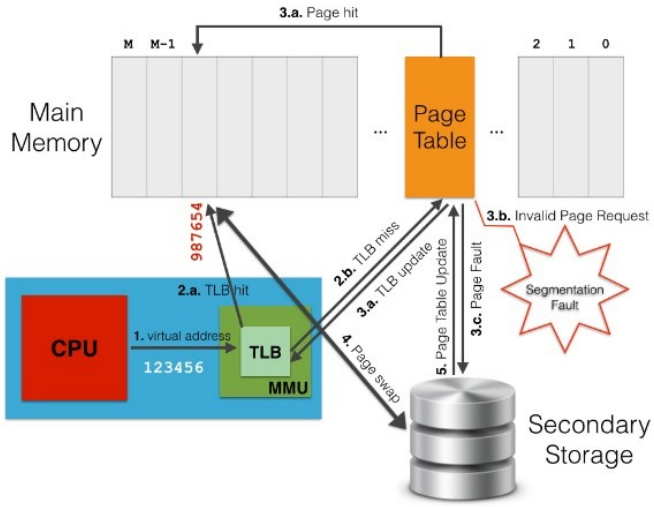

Fig. 2. Translate virtual address to physical address

virtual address needs to be translated into physical address, the MMU (Memory Management Unit) will first look in the TLB (Translation Lookaside Buffer) (step 1). If being found, the physical address is returned and the operation continues to step 2.a - read/write data. Otherwise, MMU will find the entire page table (page walk) step 2.b. If being found, it will update the TLB cache (step 3.a) and process. Activity continues steps 1 and 2.a.

In the case of step 2.b, the memory address cannot be found. There may be 2 situations: 1) This address does not exist, and 2) The related memory page is not loaded in memory. In both cases, the operating system manages the entire resources of the computer for capturing signals from the MMU. In the first case, the page supervisor typically raises a segmentation fault exception (step 3.b.). In the second case, instead, a page fault occurs (step 3.c.), which means the requested page has to be retrieved from the secondary storage (i.e., disk) where it is currently stored. At the same time, the memory manager of the operating system updates the page fault information (step 4) and updates the page table (step 5) and TLB with the new mapping between the virtual address and the physical address where the page has been archived (step 3.a.), then the operation continues with steps 1 and 2.a.

Since kernel 2.6.10, the page tables are supported with 4-level structure, beside of the old 3-level structure. The 5-level page table structure is supported since kernel 4.11-rc2 [10]. Fig.3 illustrates the 3-level page table structure [24].

With 4-level paging, the virtual address uses 48/64 bits as the address (from bits 0-47) and details of the bits used for each level is presented in Fig.4 [14]. With 5-level paging, the

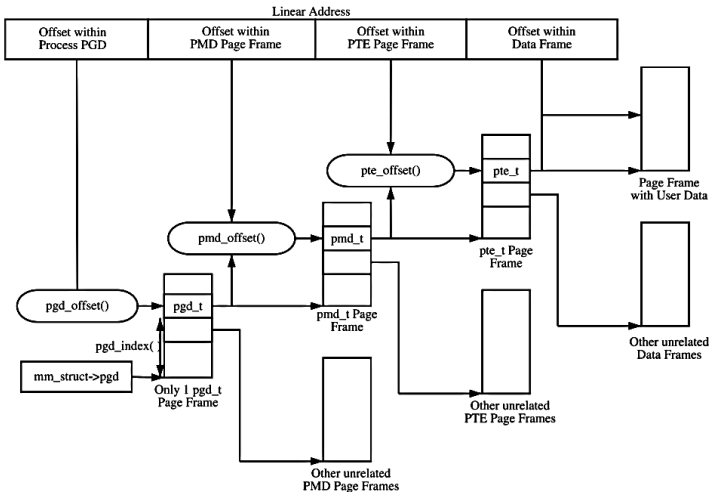

Fig. 3. 3-level page table paging

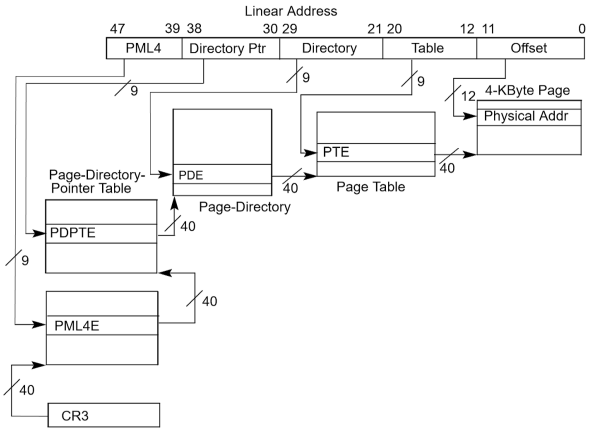

Fig. 4. Linear-address translation using 4-level paging of Intel 64

virtual address uses an additional 9 bits from 48-56 to level 5 [14] helps to support memory management with many times increased, from the limit of 128 TB to 128 PiB of virtual address space.

Corresponding to hardware support for 5-level paging, Linux has integrated this support in its kernel since version 4.11-rc2. Fig.5 [13] illustrates the 5-level paging, a new level is

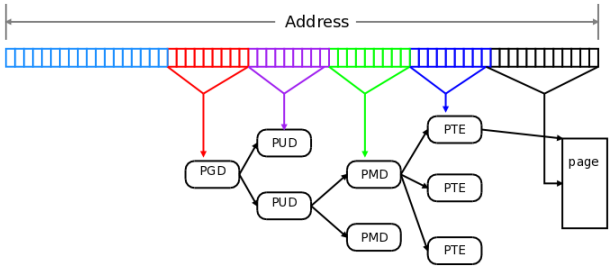

Fig. 5. Five-level page table paging

added with the PUD symbol between PMD and PGD levels to be compatible with 5-Level paging-enabled hardware. Detailed organization of the number of bits of each level is introduced in [13] [15].

C. Classification of checkpointing techniques

There are many different classifications of checkpointing techniques.

According to the independence of the operations performing the checkpointing, there are 2 types: Transparent checkpointing and Non-transparent checkpointing [1]. Transparent checkpointing: checkpointing is performed by the program itself. Transparency is usually achieved by compiling the application program with a special checkpointing library, although other methods are possible, such as rewriting executable and vice versa. Non-transparent checkpointing: programmers actively incorporate checkpointing into their programs, often with the help of libraries and preprocessors. One thing to keep in mind is that the storage of the checkpoint file, in the case of minimizing the impact on application execution time, is usually stored in internal memory. Then, in case of Non-transparent checkpointing, it is stored in the memory space of the application process itself and thus will affect its own memory space. In the case of Transparent checkpointing, it is possible to build a separate checkpointer to perform checkpointing for the application and then, the checkpoints can be stored in the memory space of the checkpointer. Thus, the memory space of the application process is not affected by checkpointing process.

Based on the composition of the checkpoints - as presented in the Introduction - checkpointing is classified into two main categories: Full checkpointing and Incremental checkpointing. Most checkpointing follow the incremental flavor checkpointing to optimize resources and performance [4] - [8], [16].

Another classification is based on the location of checkpointing operations: Kernel-level checkpointing [16] [17] [25] and User level checkpointing [4] [5] [20] [21].

In addition, there are other classifications according to programming techniques. Page-based, hash-based and proxy-based techniques which can be classified as described in the introduction.

Some well known checkpointers could be mentioned as BLCR [17], DMTCP [18] [19], CRIU [20].

In this paper, we present and compare the two programing techniques of checkpointing in the form of incremental checkpointing by page-based mechanism, only checkpointing the memory pages of the process by the technique of locking/unlocking memory. The first technique - detailed in section III-A - add a special driver to the operating system. This technique belongs to the category of Transparent checkpointing at kernel level. The second technique - discussed in detail in section III-B - belongs to Non-transparent checkpointing at user level, programmer needs to explicitly call library functions to perform the checkpointing.

III. TECHNIQUES TO LOCK /UNLOCK MEMORY REGIONS IN LINUX

A. Programming techniques to lock/unlock in kernel space

To lock/unlock the memory pages of another process, checkpointing codes must be written in the kernel level to have the privileges to call the necessary kernel functions that cannot be called in user space. It can be developed in form of a driver. Alternatively, it can be made in form an update of the kernel which requires a rebuilding the kernel, a very complicated process. Thus, the choice of developing a driver is more feasible and user friendly, in which the user only needs to install the driver for the first time and don't need to adjust or recompile the kernel.

Corresponding to 3-level paging Fig. 3, the code in form of a driver to lock all memory space of a corresponding process is as Fig. 6: In line 1, the task is the id of a process that

```
1. mm = task->mm;
2. spin_lock(&mm->page_table_lock);
3. for (vma = mm->mmap ; vma ; vma = vma->vm_next){
4.     addr = vma->vm_start;
5.     vma->vm_flags &= ~VM_WRITE;
6.     while(addr < vma->vm_end){
7.      pgd = pgd_offset(vma->vm_mm, addr);
8.      pmd = pmd_offset(pgd, addr);
9.      pte = pte_offset_map(pmd, addr);
10.     if (pte_present(*pte)){
11.       *pte = pte_wrprotect(*pte);
12.       flush_tlb_page(vma, addr);
13.     }
14.     pte_unmap(pte);
15. }
16. spin_unlock(&mm->page_table_lock);
```

Fig. 6. The code snippet to walk through 3 level-pages to lock all process's pages

is passed to the driver, so it can lock the memory of any given user application. Line 2 locks the page table. Lines 3-13 walk through each page in whole memory space. Lines 7-9 correspond to 3-level paging structure. Line 11 sets the current memory page into write-protected. Line 16 releases the lock table page. For the process of unlocking the code similar to the command lines 5 and 11 is replaced with vma $\rightarrow$ vm_flags $\models$ VM_WRITE; *pte = pte_mkwrite(*pte); to set the page into writable state.

For 5-level paging kernel, it is needed to insert 2 instructions into the above code (Fig. 6), between the lines 7 and 8 to process 2 intermediate levels. So, we have the new code as Fig. 7: The code of memory unlock needs to be similarly modified. On the other side, it is necessary to add some checks to the lock/unlock program and the read/write flag can various depending on the kernel version.

With a kernel driver containing the functions to lock/unlock memory pages, read/write from/to a memory region of a

```
7.     pgd = pgd_offset(vma->vm_mm, addr);
7.E1.  p4d = p4d_offset(pgd, addr);
7.E2.  pud = pud_offset(p4d, addr);
8.     pmd = pmd_offset(pud, addr);
9.     pte = pte_offset_map(pmd, addr);
```

Fig. 7. The code snippet is changed to walk through 5-level paging

process, the checkpointing can be developed in both transparent and non-transparent types. It means that we can call the functions of the driver inside the program which is being checkpointed or in an independent checkpointer. In our experiment, presented in Section IV, we chose the second choice, developing an independent checkpointer to perform the checkpointing operations.

B. Programming techniques to lock/unlock at user space

Using this technique, a program can only lock its own memory. Furthermore, it can only call the memory lock functions provided by the operating system or in programming language libraries. One of the most popular memory lock/unlock functions is the **mprotect** system call of Linux [22]. Its syntax is:.

```
int mprotect(void *addr, size_t len, int prot)
```

Where: addr: adress of the memory region, addr must be aligned to a page boundary; len: size of the memory region; prot: access flags. Thus, in the case of lock a memory region, the instruction becomes:

```
mprotect(void *addr, size_t len, PROT_READ)
```

And the instruction in the case of unlock is:

```
mprotect(addr, SIZE, PROT_WRITE | PROT_EXEC | PROT_READ)
```

To capture the error signal that occurs when there is a write demand to a locked memory region, the checkpointing program needs to register the SIGSEGV control of the system as the commands in Fig.8. When a SIGSEGV signal appears, checkpointer saves current values of the memory region into a buffer to create checkpoint later, then unlock the memory so the write requirement can be conformed. All actions which are executed when a SIGSEGV signal occurs must be placed in the Sigsegv_Handler function.

```
sa.sa_flags = SA_SIGINFO;
sigemptyset(&sa.sa_mask);
sa.sa_sigaction = Sigsegv_Handler;
if (sigaction(SIGSEGV, &sa, NULL) == -1)
          handle_error("sigaction");
```

Fig. 8. The code to register the SIGSEGV control

C. Comparison of the techniques to lock/unlock in kernel space and in user space

Beside the principles presented in the above sections, the most important differences between these two techniques are described in Table I. As shown in Table I, each technique has its own advantages and disadvantages. Therefore, depending on the needs and situations of use, we can choose the appropriate technique.

TABLE I
COMPARISON BETWEEN TWO MEMORY LOCK/UNLOCK MEMORY TECHNIQUES

Criteria	Kernel lock level	User lock level
Ability	Can lock the memory of another process	Can only lock the memory of the same process
Possibility of appearing errors when switching to the new kernel	Very high	Low
Customizing the lock/unlock operations	High	Low
Difficulty of developing the program	High	Low
Transparent to programmer	High	Low
Effect the memory space of application program	No	High
Speed[a]	High	High

[a]Details in session IV.

IV. EXPERIMENTS AND EVALUATIONS

We have developed two mentioned techniques to perform checkpointing operations which is the base tool of CAPE. (Checkpointing Aided Parallel Executing, a platform which ports OpenMP on distributed-memory architectures [16]) and conducted experiments to compare their speed while running OpenMP programs on this environment.

The checkpointing used in CAPE is developed in form of Discontinuous Incremental Checkpointing (DICKPT) - an improved version of incremental checkpointing. This technique uses the principle of incremental checkpointing and adds the ability of discontinuously monitoring and taking checkpoints. Fig. 9 shows the execution model of DICKPT while monitoring and checkpointing a process. The main idea is a monitor (checkpointer) always monitors the execution of an application process. Whenever it meets a **start** signal that requires to start the checkpointing, it will set all memory pages in the data region of the monitored process in write-protected state. As a result, whenever the monitored process wants to write into any write-protected page, a **SIGSEGV** signal is generated and sent to the monitor. The monitor saves data of this page into a buffer, removes the write-protected state and lets the monitored process continue to execute (write into this page). When a new checkpoint is required to be created (a **save** signal appears), the checkpointer compares the saved data in the buffer with the corresponding data in the memory space of the monitored process to extract the differences and saves them into a checkpoint. The **stop** signal is used to stop the tracking (monitoring) of the monitor. Inside a **start-stop** pair, many **save** signals can be set and many **start/save(s)/stop** triples can be set in the execution time of a process. This provides the ability of discontinuously monitoring and checkpointing an application program.

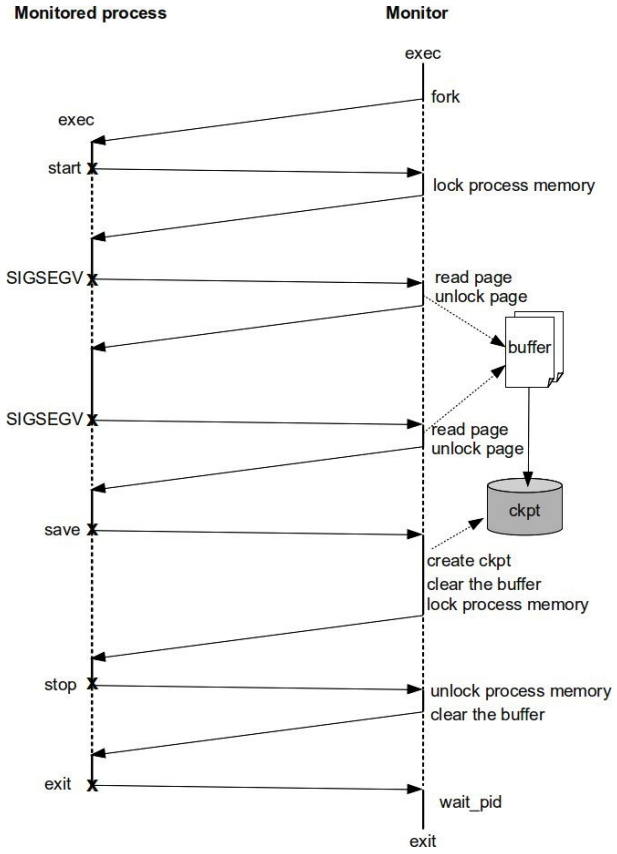

Fig. 9. Principle of DICKPT

We have implemented the experiment by running an application program which multiplies 2 square matrices on CAPE. Comparison is made for the case of checkpointing using memory lock technology in kernel level through a driver and in user level using the **mprotect** system call.

The testbed composed of 5 Intel (R) Core (TM) i3-2100 CPU @, 3.1GHz 8GB RAM, 240GB SSD computers, connected by a 100 Mbps LAN, run on Ubuntu 18.04 - Kernel 5.0.0.29.x. Experimental results are shown in Fig. 10.

In Fig. 10, the vertical axis indicates the running time in millisecond, the first horizontal axis 6000/9000 represents the size of the matrix, the second row represents the number of nodes that is 1 or 5. As be seen in the figure, running times of the two techniques are similar.

To understand this result, we have analyzed the code of **mprotect** function [28] and found that to lock/unlock a memory page, this function also has to walk through all the level-pages of memory structure as described in Part II-B and the run the similar instructions as presented in section III-A. However, the **mprotect** function has not the argument to lock

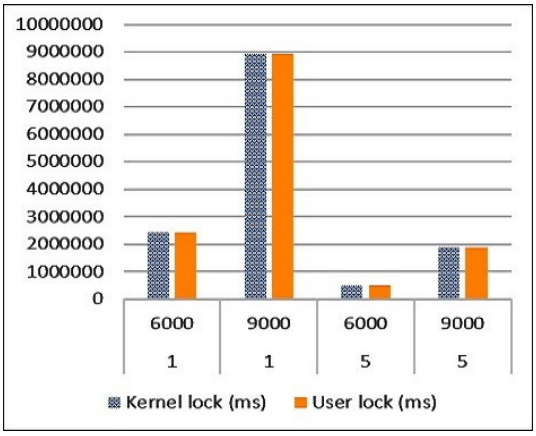

Fig. 10. Comparison CAPE running times in the cases of using 2 checkpointing techniques

the memory of another process, i.e. it can run only on the memory space of the same process.

An important difference between the two lock techniques presented in Part III is that in kernel-space lock/unlock technique we can develop an independent monitor (checkpointer) to perform the checkpointing tasks, so the checkpoints are stored outside of memory space of application program. On the other hand, in user-space lock/unlock technique, we have to insert the checkpointing instructions in the application program, so the checkpoints are processed and stored in the own memory space of this application. However, empirical evidence shows that these facts do not effect to the general running time of the both systems. Therefore, when using to perform the checkpointing in CAPE, the kernel-space technique is better than the user-space one.

V. CONCLUSION AND FUTURE WORKS

This paper presented and compared 2 programming techniques to lock/unlock memory on Linux. Each technique has its own advantages and disadvantages. The advantages of technique running in kernel space is its capability of customizing and allowing to develop an independent checkpointer which can monitor and checkpoint another process. On the other side, the technique running in user space can only lock the memory pages of the same process. However, in the case of kernel-space technique, the programming is much more complex and the possibility of appearing errors when the operating system is updated to the new kernel version is higher. In the aspect of running speed, both techniques give the same results, such as in case of being applied to develop the checkpointers for CAPE.

In the near future, we will develop a DICKPT checkpointer which can monitor and checkpoint the multi-threaded processes for improving the execution model of CAPE on multi-core processor systems.

REFERENCES

[1] James S.Plank. 1997. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance, Technical Report UT-CS-97-372, University of Tennessee.

[2] Pradeep Padala. 2002. Playing with ptrace, Part I, Linux Journal archive, Volume 2002 Issue 103, November 2002, https://www.linuxjournal.com/article/6100

[3] Pradeep Padala. 2002. Playing with ptrace, Part II, Linux Journal archive, Volume 2002 Issue 104, December 2002, https://www.linuxjournal.com/article/6210

[4] Plank, James S and Beck, Micah and Kingsley, Gerry and Li, Kai. 1994. Libckpt: Transparent checkpointing under unix, White Paper, Computer Science Department.

[5] Cores, Iv´an and Rodr´ıguez, M´onica and Gonz´alez, Patricia and Mart´ın, Mar´ıa J. 2016. Reducing the overhead of an MPI application-level migration approach, Parallel Computing, Elsevier, pp. 72–82.

[6] Z. Chen, J. Sun, and H. Chen. 2016. “Optimizing checkpoint restart with data eduplication,” Scientific Programming, vol. 2016.

[7] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. 2004. Adaptive incremental checkpointing for massively parallel systems. In ICS ’04: Proceedings of the 18th Annual International Conference on Supercomputing, pages 277–286, St. Malo, France, 2004. ACM.

[8] Cores, Iv´an and Rodr´ıguez, Gabriel and Gonz´alez, Patricia and Osorio, Roberto R and others. 2013. Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes, New Generation Computing, Springer, pp. 163–185.

[9] TfiR channel. 2014. Linus Torvalds says GPL v3 violates everything that GPLv2 stood for, Debconf 2014, https://youtu.be/PaKIZ7gJlRU

[10] Linus Torvalds. 2013. Linux kernel source tree. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/COPYING

[11] Daniel P. Bovet, Marco Cesati. 2005. Understanding the Linux Kernel, Third Edition 3rd Edition, O’Reilly Media, ISBN: 0-596-00565-2, p3.

[12] Jones, M., Arcand, B., Bergeron, B., Bestor, D., Byun, C., Milechin, L., … Reuther, A. (2016). Scalability of VM provisioning systems. 2016 IEEE High Performance Extreme Computing Conference (HPEC). doi:10.1109/hpec.2016.7761629

[13] J. Corbet, ”Linux info from the source,” Mar 15, 2017. [Online]. Available: https://lwn.net/Articles/717293/

[14] Intel Inc. 5-Level Paging and 5-Level EPT. White Paper. Revision 1.1, May 2017 [Online]. Available: https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf

[15] Kirill A. Shutemov, ”Linux info from the source,” 8 Dec 2016. [Online]. Available: https://lwn.net/Articles/708526/

[16] Viet Hai Ha and Eric Renault. 2011. Discontinuous Incremental: A new approach towards extremely lightweight checkpoints. In Computer Networks and Distributed Systems (CNDS), 2011 International Symposium on. IEEE, pp. 227–232.

[17] J. Duell. 2005. “The design and implementation of berkeley lab’s linux checkpoint/restart,” Lawrence Berkeley National Laboratory.

[18] J. Ansel, K. Aryay, and G. Coopermany. 2009. “Dmtcp: Transparent checkpointing for cluster computations and the desktop,” in Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, pp. 1–12.

[19] I. Ljubuncic, R. Giri, A. Rozenfeld, and A. Goldis. 2014. Be Kind, Rewind — Checkpoint & Restore Capability for Improving Reliability of Large-scale Semiconductor Design”, IEEE HPEC-14, Sept, 2014.

[20] Parallels; OpenVZ, ”Checkpoint/Restore In Userspace,” Open Source Software Community, [Online]. Available: http://criu.org/

[21] Red Hat. “CRIU - Checkpoint/Restore in user space” [Online]. Available: https://access.redhat.com/articles/2455211

[22] Michael Kerrisk. Linux man pages online. [Online]. Available: http://man7.org/linux/man-pages/man2/mprotect.2.html

[23] Bootlin. [Online]. Available: https://elixir.bootlin.com/linux/v5.0/source/mm/mprotect.c

[24] Mel Gorman. 2004. Understanding the Linux Virtual Memory Manager, Prentice Hall PTR Upper Saddle River, NJ, USA ©2004, ISBN:0131453483. pp. 33-34.

[25] Luciano A. Stertz. 2003. Readable dirty-bits for ia64 linux. Internal requirement specification, HewlettPackard.

[26] Wenwei Weng. 2016. Overview of Linux memory Management. [Online]. Available: http://weng-blog.com/2016/08/20/linux-mm.html