Adaptive Page Replacement to Protect Thrashing in Linux *

Song Jiang and Xiaodong Zhang
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795

Abstract

Analyzing the variations of page replacement implementations in recent Linux kernel versions of 2.0, 2.2, and 2.4, we compare their abilities to deal with system thrashing. We show that although the page implementation in Kernel 2.2 is relatively effective to protect thrashing among the three versions, none of them have adaptive ability, and thus the protection is limited. By running several groups of memory-intensive application programs on Kernel 2.2, we observe serious thrashing when memory shortage attains a certain level.

We propose and implement a thrashing protection patch in Linux kernels, which makes replacement policy responsively resolve excessive memory paging by temporarily helping one of the active processes quickly build up its working set. Consequently, thrashing could be eliminated at the level of page replacement, so that load controls at a higher level, such as process suspensions/swapping can be avoided or delayed until it is truly necessary. Our experiments show that our patch can significantly reduce page faults and the execution time of each individual thrashing process for several groups of interacting programs. We also show that our method introduces little additional overhead to program executions, and its implementation in Linux (or Unix) system is straightforward.

1 Introduction

Linux adopts the clock algorithm, an LRU approximation, to conduct its page replacement. In a multiprogrammed environment, the global LRU approximation replacement algorithm selects an LRU page for replacement throughout the entire user memory space of the computer system. The risk of low CPU utilization increases if the memory page shortage happens all over the interacting processes. For example, a process is not able to access its resident memory pages when the process is resolving page faults. These already obtained pages may soon become LRU pages when memory space is being demanded by other processes. When the process is ready to use these pages in its execution turn, these LRU pages may have been replaced to satisfy requested allocations of other processes. The process then has to request the virtual memory system to retrieve these pages by replacing LRU pages of others. The page replacement may become chaotic, and could cascade among the interacting processes, eventually causing system thrashing. Once all interacting processes are in the waiting queue due to page faults, the CPU is doing little useful work. This situation is affected by the following conditions: (1) the size of memory space in the system, (2) the number of processes, (3) the dynamic memory demands of each process, and (4) the page replacement algorithm. A robust page replacement algorithm allows the system to keep enough processes active in memory to keep the CPU busy. When a page replacement algorithm fails to prevent thrashing, some operating systems, such as FreeBSD and Solaris, employ the memory load control mechanism to suspend, even swap out some processes for a period of time. However, the memory load control has its drawbacks. It introduces too much intervention to the suspended/swapped processes. Some processes even can not afford to stop for a certain time period. After suspension or swapping out, the whole memory space of the processes can be lost, thus more efforts are needed thereafter in activation or swapping in. Moreover, process swapping is an expensive operation for both systems and user programs [8].

The most destructive aspect of thrashing is that, although thrashing may have been triggered by a brief, random peak in workload (such as all of the users of a system happening to press Enter at the same second), the system might continue thrashing for an indefinitely long time. Considering large variations of memory de-

* This work is supported in part by the U.S. National Science Foundation under grants CCR-9812187, EIA-9977030, and CCR-0098055.

mands from multiple processes and dynamic memory demands in their lifetimes of the processes, page replacement algorithms should be robust enough to deal with thrashing. Linux developers have tried to provide a solution to address the issue by improving the page replacement performance. Its main idea is to let one or more memory-intensive processes release more memory pages, in order to help others build up their working sets and then make full use of CPU cycles. In this paper, we first analyze the variations of page replacement in Linux kernels on this aspect. Considering the conflicting interests between CPU utilization and memory utilization, we show the effectiveness of the effort in this direction is limited. Our experiments also show serious thrashing could be easily found in Linux kernels. We propose a Linux patch to enhance the capability of replacement algorithms to eliminate the thrashing by dynamically monitoring system conditions and adjusting replacement algorithms accordingly. We implement the patch, and the resulting performance and its analysis are provided to show its effectiveness.

2 Variations of Page Replacement in Linux Kernel

Being an LRU approximation, the page replacement implementation in Linux is based on the following framework. The interacting processes are arranged in an order to be searched for NRU (Not Recently Used) pages when few free pages are available in the user space, and/or they are demanded by interacting processes. The system examines each possible process to see if it is a candidate from which NRU pages can be selected for replacement. The kernel will then check through all of the virtual memory pages in the selected process. In other words, it conducts its search for replaced pages in a process by process, then a page by page fashion. However, at some level of competition for memory, different implementations of this search can affect the memory usage behavior.

There are two ways in which page replacement can affect the possibility of thrashing. One is how many NRU pages are allowed to take away from a process continuously, another is how easily an NRU page can be produced. When the accumulated size of working sets of all active processes exceeds the available memory size, the amount of pages allowed to be replaced continuously from a process once it is selected determines the distribution of memory shortage among the processes. If only a small amount of NRU pages in each process is allowed to replace at a time, the memory shortage can spread all over the processes by searching

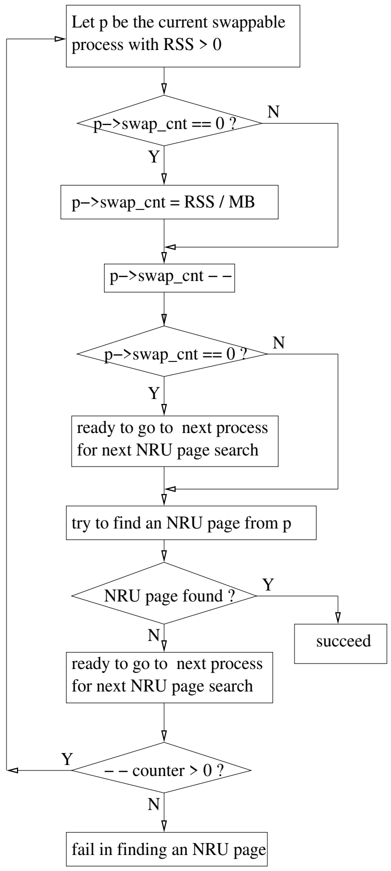

Figure 1: Selecting NRU pages in Linux Kernel 2.0.

the victim page from one process to another. If the replacement policy allows a large amount of pages to be evicted from a process once at a time, and it also has prepared enough NRU pages for eviction, memory shortage can concentrate on one or a few particular processes, which helps others have their working set established more easily. This alternative reduces the possibility of thrashing. However, in such an arrangement, the chance for the other processes to be searched for NRU pages is reduced. Thus it becomes hard to distinguish active pages in a working set from inactive pages in the other processes. This arrangement may also prevent memory space held by inactive pages from being reused by processes lacking memory space. In the following analysis of function swap_out() of Kernel 2.0, 2.2 and 2.4, where major steps regarding how to select a process for page replacement are shown in the respective flow charts, we can see the design challenge in page replacement policies.

2.1 Kernel 2.0

In Kernel 2.0, the NRU page contributions are proportionally distributed among interacting processes (see Figure 1). There is a "swap_cnt" variable for each process, which is initialized with a quantity (RSS/1MB) proportional to its resident set size (RSS). Once an NRU page is taken away from the process, its "swap_cnt" will be decreased by one. Only when its "swap_cnt" becomes zero, or the searching for an NRU page fails in resident space of the process, is the next process in the process list examined. When a process with "swap_cnt" of zero is encountered, it will be re-initialized using the same proportion rule. This strategy effectively balances memory usage by making all the processes proportionally provide NRU pages. Variable "counter" is used to control how many processes are searched before finding an NRU page. NRU pages are identified by "age", a variable associated with each page, which is increased by 3 with the maximum threshold of 20 when it is referenced, called page aging, and decreased by 3 each time the page is examined. Once the "age" decreases to zero, it will become an NRU page and ready to be replaced. Page aging helps the kernel make careful distinction among active and non-active pages. When a process is forced to stay in the waiting queue for resolving page faults, the pages in its working set are more resistant to be replaced by page aging than by simply checking the reference bit of each page. However this encourages the process to spread the memory shortage burden to others, even to those with low page fault rates, by protecting its own working set.

A major disadvantage of this approach to select NRU pages is its high potential for thrashing, resulting in low CPU utilization. This is because when all the memory-intensive processes are struggling to build its working set under heavy memory loads, no one will be given a priority for the purpose of thrashing protection.

2.2 Kernel 2.2

In order to address the limit in Kernel 2.0, Kernel 2.2 makes each identified process continuously contribute its NRU pages until no NRU pages are available in the process. Attempting to increase CPU utilization, this strategy allows the rest of the interacting processes to build up their working sets more easily by penalizing the memory usage of one process at a time. Figure 2 shows how to select a process for page replacement in the kernel.

In this kernel, the "swap_cnt" variable for a process can be thought as a "shadow RSS", which becomes zero when it fails to find an NRU page from the process. After the "swap_cnt"s of all the swappable processes become zeros, variable "assign" is set to 1, and the "swap_cnt"s are re-assigned with their RSS's in the second pass going through the process list in the inner loop. This inner loop will select the swappable process with the maximal RSS. Because the "swap_cnt" of a process remains unchanged until the NRU page search fails, a process selected for replacement will continue to be searched for NRU pages next time. Only after there is a failure in finding NRU pages, does "swap_cnt" become 0 and the process will not be selected next time. This allows its NRU pages continuously to be replaced until a failure on finding an NRU page in the process occurs.

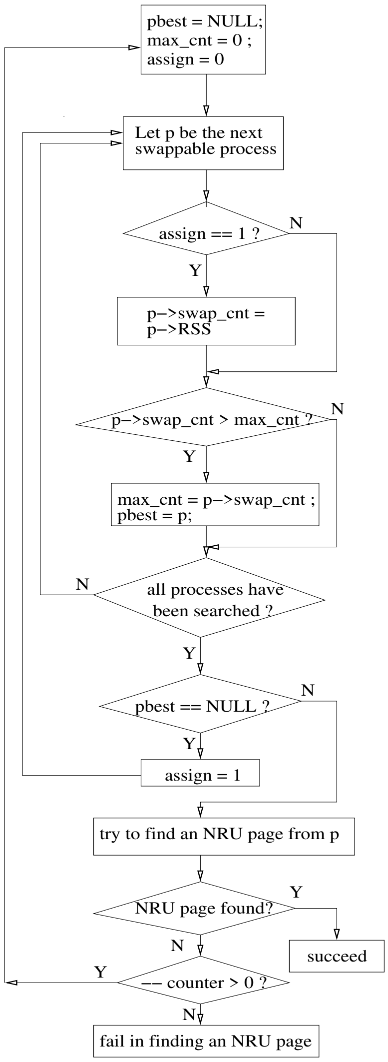

Figure 2: Selecting NRU pages in Linux Kernel 2.2.

In addition to the changes in the selection of processes for NRU pages, there has been another major change in this kernel. It eliminates page aging and only makes use of the reference bit in the PTE (Page Table Entry) of each page. The bit is set when the page is referenced and reset when the page is examined. Thus, in this kernel the pages with reference bits of 0s are NRU pages and ready to be replaced. This implementation will produce NRU pages more quickly for a process with a high page fault rate. Both these changes in kernel 2.2 take an aggressive approach to make an examined process contribute more of its NRU pages than others, attempting to help other interacting processes to establish their working sets to fully utilize the CPU.

2.3 Kernel 2.4

The latest Linux kernel is version 2.4, which makes considerable changes in the paging strategy. Many of these changes target at addressing concerns on memory performance that arises in Kernel 2.2. For example, without page aging, NRU replacement in Kernel 2.2 can not accurately distinguish the working set from incidentally accessed pages. Thus, Kernel 2.4 has to reintroduce page aging, just as Kernel 2.0 and FreeBSD do. However, the page aging could help processes with high page fault rates to keep their working sets, causing other processes to have serious page fault rate at the same time, and trigger thrashing. In kernel 2.4, the "age" variable in each page is decremented exponentially (the variable is divided by 2 instead of subtracting a constant value) when an unaccessed page is identified, which may be slightly beneficial to the thrashing protection compared with kernel 2.0.

To make memory more efficiently utilized, Kernel 2.4 reintroduces the method used in Kernel 2.0 for selecting processes to contribute NRU pages (see Figure 3). Going through a process list each time, it walks only about 6% of the address space in each process to search NRU pages before it goes to the next process. Similar to that of Kernel 2.0, this method increases its possibility of thrashing. However, before identifying a process in the list, Kernel 2.4 first takes an extra NRU search of about 6% of the address space of the process that is allocating memory to penalize it, which is the process encountering page faults in heavy memory load. This arrangement is only slightly helpful to prevent thrashing.

Kernel 2.4 distinguishes the pages with age of zero and those with positive ages by separating them into non-active and active lists, respectively to prevent inefficient interactions between page aging and page flushing [9]. The NRU pages with reference bit zeros possibly have positive ages, and are moved to the active list, where page aging is conducted. This change does not help protect the system against thrashing, because the system still has no knowledge on which particular working sets of the processes should be protected when frequent page replacement takes place under heavy memory workload. Similar argument can be applied in BSD and FreeBSD, where a system-wide list of pages forces all processes to compete for memory on an equal basis.

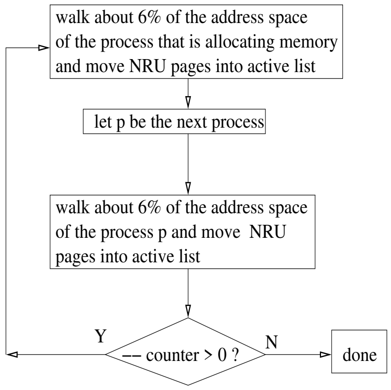

Figure 3: Selecting NRU pages in Linux Kernel 2.4.

2.4 Why an adaptive replacement policy is needed for thrashing protection?

The effort made in Kernel 2.2 tries to retain CPU utilization by avoiding widely spreading page faults among all the interacting processes. However, such an effort increases the possibility of replacing fresh NRU pages in the process being examined, while some NRU pages in other interacting processes that have not been used for long time continue to be kept in the memory. In other words, this approach benefits CPU utilization at the cost of lowering memory utilization. Unfortunately, Kernel 2.4 increases the potential of thrashing and lowering CPU utilization while it tries to address the weakness in its predecessor. The difficulty in the implementation of replacement algorithm calls for a solution which (1) makes memory resource be efficiently used when CPU utilization is not a concern; and (2) changes its replacement behavior to help system recover from thrashing when CPU utilization is low. An adaptive replacement policy adjusting its behavior to the system conditions such as CPU utilization and page fault rates, can achieve both goals at the same time. Our solution is a kernel patch which adjusts existing replacement implementation based on dynamically monitoring system conditions, making adaptive page replacement.

3 The Design and Implementation of Thrashing Protection Patch

The main idea of the patch is quite simple and intuitive. Once multiple "CPU cycle eager" processes but with high page fault rates and low CPU utilization coexist, the patch will make a temporal tuning on the page replacement to help one of the processes to establish its working set and let it consume CPU cycles. Otherwise, the patch is almost dormant and adds little intervention to the original system.

Our patch implementation on Kernel 2.2 consists of two kernel utilities: detection and protection routines. The detection routine is used to dynamically monitor the page fault rate of each process and the CPU utilization of the system. The protection routine will be awakened to adjust the page replacement when the CPU utilization is lower than a predetermined threshold, and when the page fault rates of more than one interacting process exceed a threshold. It then grants a privilege to an identified process which will only contribute a limited number of NRU pages. The detection routine also monitors if the identified process has lowered its page fault rate to a certain degree. If so, its privilege will be disabled. This action will retain the memory utilization by treating each process equally.

There are four predetermined parameters used in the patch:

1. CPU_Low: the lowest CPU utilization the system can tolerate.

2. CPU_High: the targeted CPU utilization for the patch to achieve.

3. PF_Low: the targeted page fault rate [1] of the identified process for the patch to achieve.

4. PF_High: the page fault rate threshold for a process to potentially cause thrashing.

We add one global linked list, high_PF_proc, in the kernel to record interacting processes with high page fault rates. A process enters the list when its page fault rate exceeds "PF_High", and exits from the list when its page fault rate lowers below "PF_Low".

The kernel memory management has the following three states with dynamic transitions:

1. *normal state*: In this state, no monitoring activities are conducted. The system deals with page faults exactly as the original Linux kernel does. The system keeps track of the number of page faults for each process and places the processes with page fault rates higher than "PF_High" into "high_PF_proc".

2. *monitoring state*: In this state, the detection routine is awakened to start monitoring the CPU utilization and the page fault rates of processes in the linked list. If the protection condition is satisfied, the detection routine will select a process in "high_PF_proc" for protection and go to the protection state. The system returns to the normal state when there is no more than one processes in "high_PF_proc".

3. *protection state:* The protection routine will mark the selected process and let its "swap_cnt" reset to 0 no matter whether a replaced page has been successfully found, which let the process contribute at most one page continuously and help it quickly establish its working set. In the protection state, the detection routine keeps monitoring the CPU utilization and the page fault rate of each process in the list. The detection routine is deactivated and the protection state transfers to the monitoring state as soon as the protected process obtains low page fault rate, and/or the CPU utilization has been sufficiently improved.

[1] In our experiments only those page faults that are revolved by loading pages from the swap files in disk are counted, because they are the most appropriate factors to reflect the effect of memory shortage on processes.

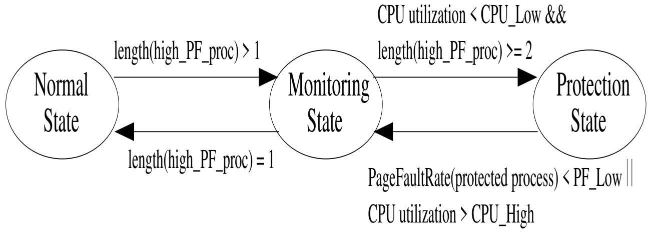

Figure 4: Dynamic transitions among normal, monitoring, and protection states in the improved kernel system.

Figure 4 describes the dynamic transitions among the three states, which gives a complete description of the patch. When the system is normal (no page faults occur), detection and protection routines are not involved. The patch only adds limited operations for each page fault and checks several system parameters with the interval of one second. So, overhead involved in detection and protection is trivial compared with the CPU overhead to deal with page faults.

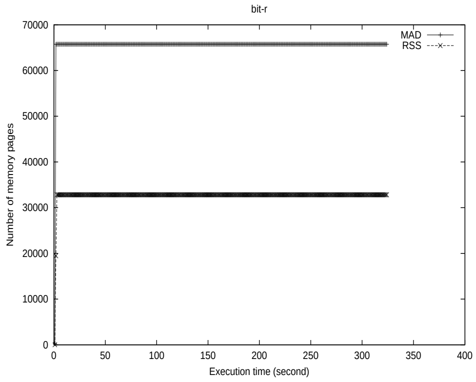

Figure 5: Memory usage pattern of program bit-r.

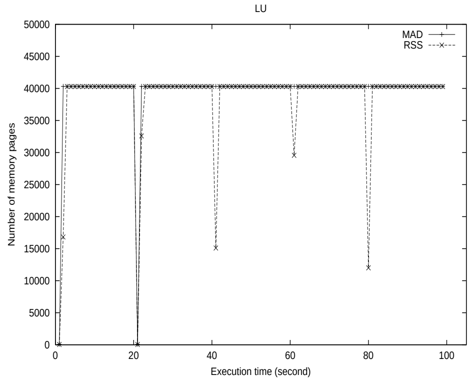

Figure 6: Memory usage pattern of program LU.

4 Performance Measurements

Our experiments are conducted on a Pentium II of 400 MHz with available user space 384 MBytes and an IBM Hercules disk. The operating system is Red Hat Linux release 6.1 with the kernel 2.2.14. The predetermined threshold values are set as follows: CPU_Low = 40%, CPU_High = 80%, PF_High = 10 page faults/second, PF_Low = 1 page fault/second. We also instrumented the kernel to adjust the available user memory so that different memory constrains can be formed to facilitate our experiments.

We have selected the following 3 both memory-intensive and CPU-intensive application programs:

- *bit-reversals*, (bit-r): This program conducts data reordering operations which are required in many Fast Fourier Transform (FFT) algorithms.
- *LU decomposition,* (LU): This is a standard matrix LU decomposition program for solving linear systems.
- *C compiler,* (gcc): This is an optimized C compiler from SPEC2000.

The memory usage patterns of the three programs are plotted by memory-time graphs for their isolated executions in Figure 5, 6, 7. In the memory-time graph, the x axis represents the execution time sequence, and the y axis represents two memory usage curves: the memory allocation demand (MAD), the resident set size (RSS), both can be obtained directly in the kernel data structure of *task_struct*. In our current implementation, the process we selected for protection among the candidates is the one with least (MAD - RSS), which is possibly easy to attain its full working set.

We first ran two groups of the interacting programs, gcc + bit-r with 31% memory shortage and LU-1 + LU-2 with 35% memory shortage, where LU-1, LU-2 are

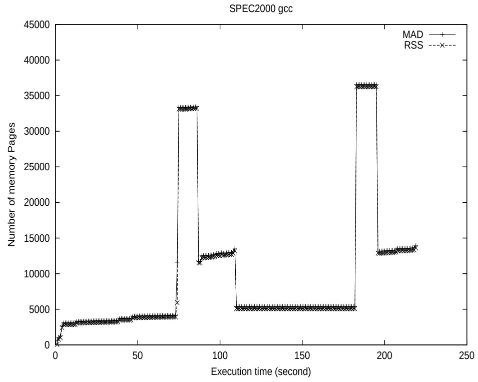

Figure 7: Memory usage pattern of program gcc.

two executions of program LU, on the original Kernel 2.2.14. Then we ran the same groups on the kernel with our thrashing protection patch, where all the other experimental settings are the same. We present the memory usage measured by MAD and RSS of these experiments in Figures 8, 9, 10, and 11.

Figures 8 and 10 show that there were serious thrashing in Kernel 2.2, even though it takes considerable thrashing protection into account in its implementation.

The program gcc has two spikes in MAD and RSS due to its dynamic memory allocation demands and accesses (see Figure 7). In Figure 8, the first spike of gcc made the RSS curve of bit-r dropped sharply at the 165th second without causing thrashing. However, the second RSS spike of gcc, which just has only 7% more memory demand, incurred thrashing. Program gcc began to loose its pages at about 450th second before it could establish its working set, shown by the decreasing of its RSS curve. After that, both programs exhibited fluctuating RSS curves and started thrashing. During most of the thrashing period, both of them were in the waiting queue for the resolving of their page faults, leaving CPU

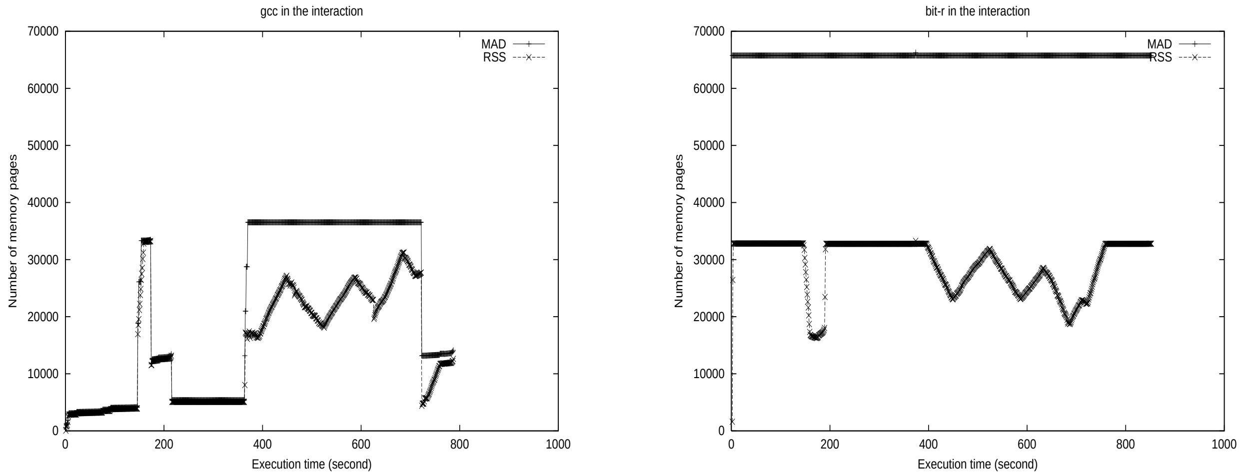

Figure 8: The memory performance of gcc and bit-r during the interactions with the original Linux Kernel 2.2.

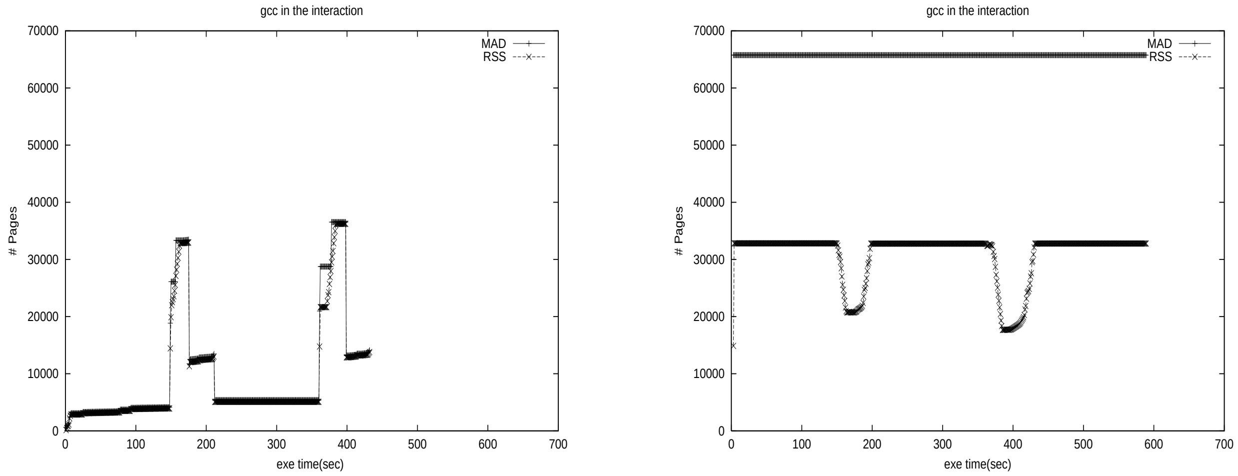

Figure 9: The memory performance of gcc and bit-r during the interactions with thrashing protection.

idle. Comparatively, in the same kernel with thrashing protection, the second spike of gcc went smoothly without visible thrashing (see Figure 9). Once monitor facility detected the sign of thrashing, protection routine changed the behavior of replacement policy: gcc was selected to reduce its contribution of pages for replacement, thus quickly built its working set. After gcc finished its second spike, the released memory went to bit-r. Thus CPU was able to retain its utilization for almost the whole time. Our measurements show that with our patch the slowdown of gcc is reduced from 3.63 to 1.97, the slowdown of bit-r is reduced from 2.69 to 1.81. The number of page faults reductions for gcc and bit-r are 95.7% and 49.8% respectively.

In Figure 10, both processes LU-1, LU-2 have the same memory usage pattern and started their executions at the same time. Thus frequent climbing slopes of RSS incurred memory frequent reallocations, and triggered fluctuating RSS curves, leading to inefficient memory usage and low CPU utilization. The dynamical memory demands from the processes caused the system to stay in the thrashing state for most of the time. However, when our protection patch is in effect, the protected process LU-1 can run very smoothly (see Figure 11), its RSS curve is much same as the one in the isolated execution (see Figure 6). This shows that our patch can responsively adapt replacement behavior to the dynamically changing demand of memory load from multiple processes, thus keep high CPU utilization. Our measurements show that the slowdown of LU-1 relative to its isolated execution time is reduced from 3.57 to 1.63, the slowdown of LU-2 is reduced from 3.40 to 2.18. The

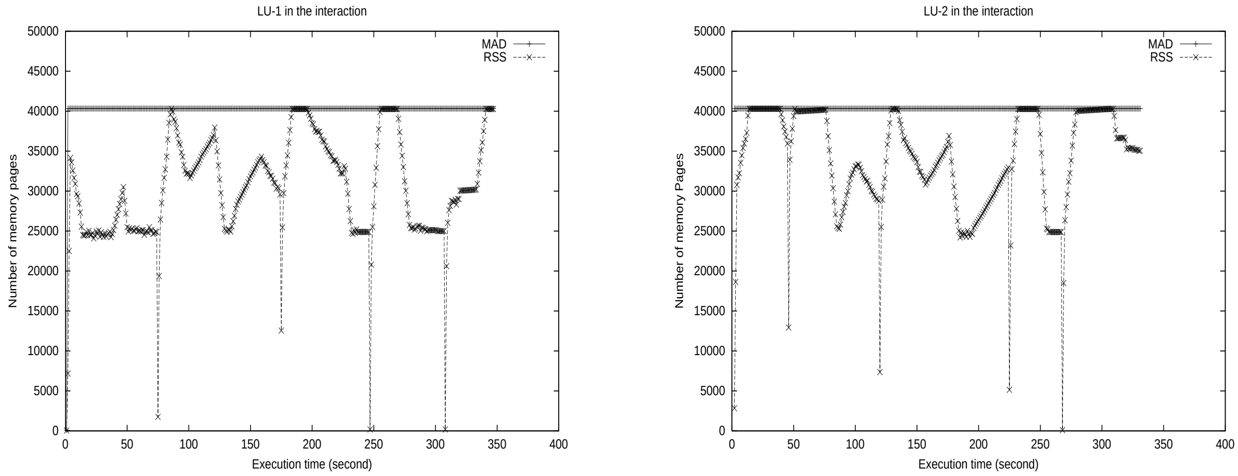

Figure 10: The memory performance of gcc and vortex during the interactions with the original Linux Kernel 2.2.

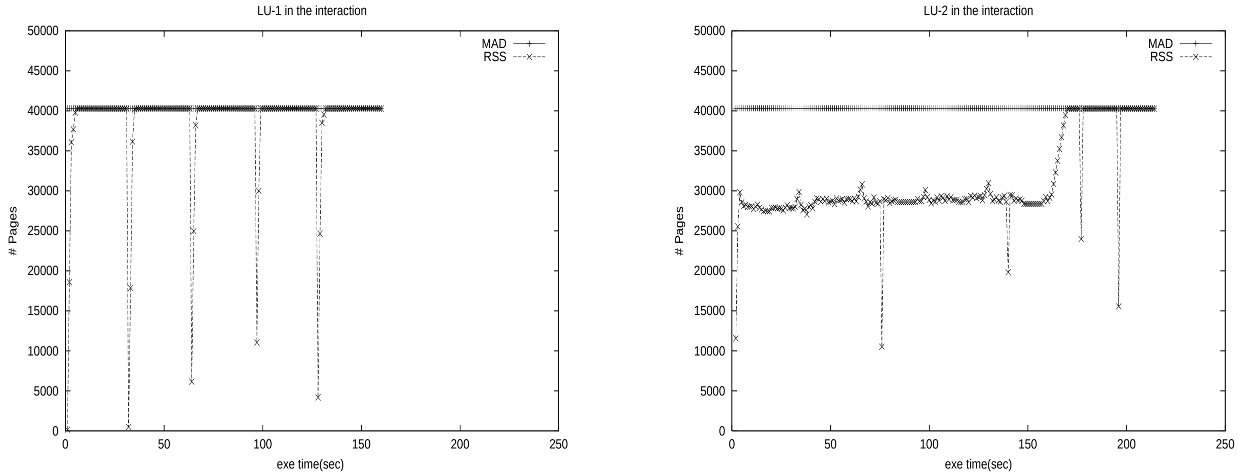

Figure 11: The memory performance of gcc and vortex during the interactions with thrashing protection.

number of page faults reductions for LU-1 and LU-2 are 99.4% and -39.6% respectively. Though the number of page faults of LU-2 is increased, its execution time is still greatly reduced, which is partly because of the increased I/O bandwidth when its peer process LU-1 has greatly reduced its page faults due to the protection.

5 Related work

Improvement of CPU and memory utilization has been a fundamental consideration in the design of operating systems. Studies of page replacement policies have a direct impact on memory utilization, which have continued for several decades. The goal of an optimal page replacement is to achieve efficient memory usage by only replacing those pages not used in the near future when available memory is not sufficient, reducing the number of page faults. In a single-programming environment, these proposed methods address both concerns of CPU and memory utilization since any extra page faults due to low memory utilization will make the CPU stall. However system thrashing associated with the concurrent execution in a multiprogramming can not be fully covered by the work on this aspect due to the conflicting interests between CPU and memory utilization mentioned in Section 2.4.

In the multiprogramming context, mainly there are two methods to eliminate thrashing. One is local replacement, another is load control. A local replacement requires that the paging system select pages for a program only from its allocated memory space when no free pages can be found in their memory allotments. Unlike the global replacement policy, the local policy needs a memory allocation scheme to satisfy the need of each program. Two commonly used policies are equal and proportional allocations, which can not capture dynamically changing memory demand of each program [2]. As a result, the memory space may not be well utilized. On the other hand, an allocation dynamically adapting to the demand of individual programs will shift the scheme to the global replacement. The VMS [5] is a representative operating system using a local replacement policy. The memory is partitioned into multiple independent areas, each of which is localized to a collection of processes that compete with one another for memory. Unfortunately, this scheme can be difficult to administer [6]. Researchers and system practitioners seem to have agreed that a local policy is not an effective solution for virtual memory management. Our patch is built on the global replacement policy of Linux.

A commonly used load control mechanism is to suspend/reactivate, even swapping out/in processes to free more memory space after the thrashing is detected. The 4.4 BSD operating system[8], AIX system in the IBM RS/6000[4], HP-UX 10.0 in HP 9000 [3] are the examples to adopt this method. In addition, HP-UX system provides a "serialize()" command to run the processes once at a time after thrashing is detected. Compared with load control, our patch works to the same objective but not through action on the whole processes, instead through adaptively adjusting page replacement algorithms. Memory allocation scheduling at this level allows us to carefully consider the tradeoff between CPU and memory utilization.

6 Conclusion

We have investigated the risk of system thrashing in page replacement implementations by examining the Linux kernel code of versions 2.0, 2.2, and 2.4, and running interacting memory-intensive programs in a Linux system. Our study indicates that this risk is hard to avoid in a non-adaptive replacement implementation due to the conflicting interests of requirements on CPU and memory utilizations. We have proposed and implemented a patch to enhance the existing replacement policy into an adaptive one in the Linux kernel to prevent the system from thrashing among interacting processes, and to improve the CPU utilization under heavy memory load. Conducting experiments and performance evaluation, we show that our method can effectively provide thrashing protection without negative effects to overall system performance for three reasons: (1) the privilege is granted only when a thrashing problem is detected; (2) although the protected process could lower the memory usage of the rest of the interacting processes for a short period of time, the system will soon become stable by the protection; and (3) Our patch is simple to implement with little overhead in the Linux kernels.

In our current experiment we always select the process which most possibly has the least memory shortage, because we intent to put least effort to fulfill our protection goal. However, this scheme may allow the same process to be protected more often than others. This may incur fairness concern. A simple solution is to let the protection privilege alternate among all the active processes. We would like include this in our future work.

Acknowledgments: We thank Phil Kearns for providing a kernel programming environment. We appreciate Bill Bynum for reading the paper and for his suggestions.

References

[1] M. Beck, et. al., *Linux Kernel Internals*, Second Edition, Addison-Wesley, 1998.

[2] E. G. Coffman, Jr., and T. A. Ryan, "A study of storage partitioning using a mathematical model of locality", *Communications of the ACM*, Vol. 15, No. 3, 1972, pp. 185-190.

[3] HP Corporation, *HP-UX 10.0 Memory Management White Paper*, January 1995.

[4] IBM Corporation, *AIX Versions 3.2 and 4 Performance Tuning Guide*, April 1996.

[5] L. J. Kenah and S. F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press, Bedford, MA, 1984.

[6] E. D. Lazowska and J. M. Kelsey, *Notes on Tuning VAX/VMS*, Technical Report 78-12-01. Dept. of Computer Science, Univ. of Washington, Dec. 1978.

[7] S. Maxwell, *Linux Core Kernel Commentary*, CoriolisOpen Press, 1999.

[8] M. K. McKusick, K. Bostic, M. J. Karels, J. S. Quarterman, *The Design and Implementation of the 4.4 BSD Operating System*, Addison Wesley, 1996.

[9] R. van Riel, "Page replacement in Linux 2.4 memory management", *Proceeding of USENIX Annual Technical Conference*, (FREENIX track), Boston, Massachusetts, June 2001.