REDUCING THE THRASHING EFFECT USING BIN PACKING

Moses Reuven
Computer Science Department
Bar-Ilan University
Ramat-Gan
Israel
reubenm@cs.biu.ac.il

Yair Wiseman
Computer Science Department
Bar-Ilan University
and School of Computer Science & Engineering
The Hebrew University of Jerusalem
wiseman@cs.huji.ac.il

Abstract

We suggest a method for minimizing the paging on a system with a very heavy memory usage. Sometimes there are processes with active memory allocations that need to be in the physical memory, and their total size exceeds the physical memory size. In these cases the operating system will start swapping pages in and out of the memory on every context switch. We minimize this thrashing by splitting the processes into a number of bins, using Bin Packing approximation algorithms. We change the scheduler to have two levels of scheduling - medium-term scheduling and short-term scheduling. The medium-term scheduler switches the bins in a Round-Robin manner, while the short-term scheduler runs the standard Linux scheduler among the processes in each bin. Experimental results show significant improvement on heavily loaded memories. The code of this project is free and can be found in: http://www.cs.biu.ac.il/~reubenm

Key Words: Thrashing, Bin-Packing, Kernel Manipulation.

1 Introduction

Many operating systems implement the virtual memory scheme[1] using the paging concept i.e. the operating system loads a memory page into the physical memory only when a process demands it. If no free memory pages are available, the operating system swaps some other pages back onto the secondary memory (hard disk). Different methods of deciding which pages should be swapped out to the disk have been proposed over the years [2].

When too much memory space is needed, the CPU spends a lot of time swapping pages in and out of the memory. This effect is called Thrashing [3], and the outcome is a severe overhead of time, and hence a significant slowdown of the system. Some studies for reducing the undesirable effects of the thrashing have been conducted over the years [4].

In [5,6] The authors suggest giving one of the interactive processes the privilege of not swapping its page out. Consequently, the privileged process will succeed in being executed faster, and will free its memory allocation earlier. This can help the operating system to clear out the memory and to return to normal behavior.

In [7] the authors suggest not admitting jobs which do not fit into the current available memory. Instead, the system waits for one or more processes to finish their execution, and only when enough memory is freed a new job will be admitted. The authors also discuss how to assess the memory size needed by a new job. This is actually very similar to what VMS does using the “Balance Sets” method. However, the authors of this paper taking the same “Balance Sets” concept to distributed systems.

In [8] the author suggests tackling the thrashing effect by adjusting the memory needs of the process to the current available memory. This solution is quite different from the others, because it modifies the processes instead of modifying the operating system.

This paper suggests a technique of changing the traditional process scheduling procedure. This technique will help the operating system to swap in and out a smaller number of pages, and thus to minimize the slowdown stemming from thrashing. The solution suggested in this paper is not restricted to a specific operating system; hence it can be implemented on any multitasking paging system. The figures and the results given in this paper have been achieved using the Linux operating system [9]. However, Linux is just a platform to show the feasibility of the concept.

The rest of the paper is organized as follow. Section 2 describes the Linux scheduling algorithms. Section 3 introduces the Bin Packing problem. Section 4 presents the reduced paging algorithm. Finally, section 5 gives the results and evaluates them.

2 The Linux Scheduling

2.1 Linux Scheduling

Linux divides the time into *epochs*, in which each process gets its time slice. Time slices are measured in ticks, each tick is 10.5 milliseconds long. Processes can get between 1 and 30 ticks, the usual time slice being 15 ticks. An epoch ends when all tasks have exhausted their time slices. When a process begins, a new time slice for this

process is calculated. Linux has two records of all of active processes. The "active" record holds the processes that have not exhausted their time slices yet. The record has 140 queues, one for each possible priority, and a bit-map of 140 entries to indicate which queues are not empty. The "expired" record has the same structure, and holds the processes which have exhausted their time slices. When a process exhausts its time slice, it will be moved from the "active" record to the "expired" record, and a new priority and time slice are calculated for it. When a new process is created, it will get the same priority as its parent process, and the reminder of the parent's time slice in the current epoch is equally split between the parent and the child. In the next epoch, the parent and the child each get distinct time slices.
The Linux scheduler is invoked when a process exhausts its time slice, when a process stops consuming CPU cycles or when a process returns to the "ready" queue. Linux then invokes the process with the highest priority. When all the processes have exhausted their time slice, the "expired" and the "active" records are switched. If no process is ready to run an "idle" process is invoked.
The queues of the priorities are implemented as follow: When the priority of a process is changed, it is requeued on the appropriate queue, and the corresponding bit is marked. The values of the priorities are within the range of 0-139, where the values of 0-99 denote real time processes of both types (real time FIFO and real time Round-Robin), and 100-139 is reserved for user applications processes. A bonus within the range of –5 to 5 is added to the static priority. The bonus is calculated according to the time the process has slept during the last epoch. The static priority of user applications is initialized to 120, so the effective priorities of user applications are in the range of 115-125, so actually only 25% of the available priorities are used for user processes.

2.2 Linux Thrashing Handling

The virtual memory capability, along with the paging mechanism, gives Linux the ability to handle processes, even when the real memory needs are larger than the available physical memory. However, virtual memory cannot handle every situation. If the demand for memory pages is very high over a short period of time, the swapping mechanism cannot satisfy the memory needs reasonably. Pages are frequently swapped in and out, because of the thrashing effect, and hence little progress is made.
Linux only kills processes when thrashing occurs and the system is out of swap space. In some sense there is nothing else that the kernel could do in this case, since memory is needed but there is no more physical or swap memory to allocate [10,11]. When such a case occurs, the Linux kernel kills the most memory consuming processes. This feature is very drastic; hence its implications might be severe. For example, if a server is running several applications with mutual dependencies, killing one of the applications may yield unexpected results.

3 Bin Packing

We would like to have a list of all the processes that are in the virtual memory, and to split these processes into groups, so that the total memory size of each group will be as close as possible to the size of the available real memory i.e. the size of the bin. How can we build these groups of processes? We have a set of processes P_i, each with a memory allocation. Let M_i denote the maximal size of memory which might be needed by process P_i. We need an algorithm which divides these M_i's into as few groups as possible, with the sum of the M_is in each group not exceeding the size of the real memory. Moreover, the kernel and some other daemons occupy part of the memory, so the sum should not exceed a smaller memory size. This is a well-known problem, called the Bin Packing problem [12].
The Bin Packing problem is defined as a set of numbers $X_1, X_2, ..., X_n$, with $X_i \in [0, 1]$ for each i. The problem is to find the smallest natural number m for which:

- $X_1, X_2, ..., X_n$ can be partitioned into m sets.
- The sum of the members of each set is no higher than 1.

The Bin Packing problem is NP-hard [13]. However, some polynomial time approximations have been introduced over the years, such as [14,15,16]. The approximation algorithms use no more than (1+E)*OPT(I) number of bins, where OPT(I) is the number of bins in the optimal solution for case I. If E is smaller, the result will be closer to the optimal solution, but unfortunately good approximations are usually time consuming [17]. We would like to choose one of the approximation algorithms which is not time consuming, but yet tries minimizing (1+E)*OPT(I).
A simple idea of an approximation algorithm for the Bin Packing problem is the greedy approach [18], also known the First-Fit approach. This algorithm is defined as follow:

- Sort the vector $X_1, X_2, ..., X_n$ by the size of allocated memory.
- Open a new bin and put the biggest number in it.
- While there are more numbers
  - o If adding the current number to one of the existing bins will exceed the size of the bin
    - ▪ Open a new bin and put the current number in it.
  - o Else
    - ▪ Put the current number in the current bin.

In our test we used a version of this approximation algorithm, with a slight change. We usually achieved the minimal number of bins, and the cost of execution time was usually low. This version is described below.

4 Bin Packing Based Paging

It is well known that increasing the level of multitasking in any operating system may sometimes cause thrashing, because an excessive demand for real memory causes the operating system to spend too much time swapping. In order to avoid thrashing, we would like to suggest a new approach: All the processes will be divided into groups in such a manner that the sum of physical memory demand within each group will be no greater than the amount of physical memory available on the machine.

4.1 The Medium-Term Scheduler

A new scheduler procedure will be added to the Linux operating system. The new scheduler will operate in the manner of the medium-term scheduler, which was part of some operating systems [19]. The medium-term scheduler will load the groups into the Ready queue of the Linux scheduler in a Round-Robin manner. The traditional Linux scheduler will do the scheduling within the current group in the same way the scheduling is originally done on Linux machines. The time slice of each group in the medium-term scheduler will be significantly higher than the average time allocated to the processes by the Linux scheduler. The processes in the real memory will not be able to cause thrashing during the execution of the group because their total size is no greater than the size of the available physical memory i.e. the size of the bin. Only at the beginning of a group execution will there be intensive swapping, when the new group's pages are swapped into the memory. This can improve the ability of the system to support memory-consuming processes in a more tolerant way than killing them.

In our implementation, we used a group time slice of one or two seconds, while the Linux scheduler gives time slices of some dozen milliseconds. When Linux thrashes, any context switch causes many page faults, while with the medium-term scheduler, intensive swapping will occur only when switching between groups. This makes the operating system in our implementation swap a significant amount of pages only in a few percents of the cases, in contrast to conventional Linux during thrashing conditions.

When the medium-term scheduler replaces the current running group by the next group, the processes which are not in the current group should be kept on a different queue, so that the Linux scheduler will not be able to see them. In order to implement this feature, we added a new record to the code of Linux kernel. This record has the same structure as the "active" and "expired" records which we described in section 2.2, and this record holds the hidden processes.

When the last group completes its execution, the medium-term scheduler is invoked, and rebuilds the process groups, taking into account any changes to the old processes (e.g. exited or stopped) and adding any new processes to the groups.

Sometimes the current group finishes all of the processes within the time slice awarded to it by the medium-term scheduler. Even if there are still some processes in the group, these processes might be sleeping. If all the processes in the group are not ready to be executed, the Linux scheduler has been changed to invoke the medium-term scheduler, which in turn switches to the next waiting group.

The medium-term scheduler takes the sum of the memory sizes that are currently resident in the physical memory and divides this sum by the bin size. The quotient is taken to be the number of bins. Then, the medium-term scheduler scatters the processes between these bins. The medium-term scheduler uses the greedy algorithm until the medium-term scheduler is unable to fit another process into the bins. Next, the medium-term scheduler tries to find room for all the remaining processes in the existing bins. If it fails to find room in one of the existing bins, it exceeds the size of the smallest bin by adding the unfitting process to it. The original Bin Packing problem does not allow such a solution, but in this case it might be preferable to have a few page faults within a group than adding an additional bin.

4.2 Swapping Management

When the time slice of a group ends, a context switch of groups will be performed. This context switch would probably cause a lot of page faults: The kernel uses its swap management to make room for the processes of the new group, and this procedure might be long and fatiguing. The previous group of processes has most probably used up most of the available physical memory, and when the swap thread executes the LRU function to find the best pages to swap out to the disk, it will find pages of the old group. This procedure is wasteful because the paging functions is performed separately for every new page required. The Linux kernel does not know at the context switch time that even the hottest pages, which were recently used, will not be needed for a long time, and can be swapped out.

In order to take care of this issue, when the medium-term scheduler is invoked, it calls the Linux swap management functions to swap out all of the pages that belong to the processes of the previous group. This will give Linux a significant amount of empty frames for the new group. This swapping management approach will be much quicker than loading the pages of the new process group, and for each page fault searching for the oldest page in the physical memory to swap out. When a round of the medium-term scheduler is completed, the medium-term scheduler rebuilds the groups of processes, and some processes may migrate from one group to another; hence the medium-term scheduler does not call the Linux swap management, because it might swap out pages that may be needed again for the next group.

4.3 Group Time Slice

Sometimes the sizes of the total memory needs in the different groups are almost equal. This is the best situation, because the fixed time slice that will be given to the groups is usually quite fair. However, when the sizes of the memory allocations are significantly different, some processes might get an implicit high priority. When the medium-term scheduler uses the greedy approximation, such a situation usually occurs when the last processes are assigned to a bin. The last bin is sometimes almost empty; hence the processes in this bin gain precedence, because in the time slice of this bin, there are less processes competing for CPU cycles. It

should be noted that when the size of the last bin is not small, this solution will function efficiently.
One possible solution is to break up the small group, and to scatter the processes owned by the small group into the other groups. This solution is good when the size of the small group is not big, and when there is just one small group. If the size of the small group is big, scattering it might cause thrashing in the other groups.
A better solution can be a dynamic group time slices, instead of constant time slices. E.g. if the size vector is [1,1,0.5] and the default group time slice is one second, the medium-term scheduler should assign each of the first two groups one second, while the last group will get only 0.5 second. (The vector representing the group's memory size as the total memory allocations divided by the total memory available for user application). This solution gave us the best results; therefore, it has been implemented.

4.4 Interactive Processes

The interactive processes should be dealt with differently. If we treat them the same as the non-interactive processes, they will not be able to be executed as long as their group is not current. This is not a disadvantage for non-interactive processes, because the total execution time is smaller. However, interactive processes need fast response time, and a few seconds delay can be a major drawback.
To remedy this drawback, we will allow an interactive process, which can be identified by directly quantifying the I/O between an application and the user (keyboard, mouse and screen activity) [20], to run in each of the process groups. So, actually the process will belong to all of the groups, but with a smaller time slice in each group:
p->time_slice = time_slice(p)/num_of_groups;
This feature can assure us a short response time for interactive processes while keeping fairness towards other processes. The resident pages of interactive processes will be marked as low priority swappable, so the interactive processes will not be swapped when a group context switch is done. However, we need to calculate the memory needs of the interactive processes in every group.

4.5 Real Time Processes

The handling of real time processes is somewhat similar to interactive processes. Real time processes must get the CPU as fast as possible. The management of these processes will be the same as interactive processes, but with a slight difference. Real time processes will belong to all of the groups, as the interactive processes do, but they will not have a shrunken time slice.
The real-time processes will not be swapped out, because they belong to all of the groups. In addition, they will have the same privilege Linux traditionally gives them. Indeed, we need to calculate their memory needs in every group as we did for the interactive processes. This handling is identical for FIFO Real-Time processes and for RR Real-Time processes. This treatment has also been given to the "init" process and the "Idle and Swapper" process of Linux, which cannot be suspended.

5 Evaluation and Results

5.1 Testbed

We tested the performance of the kernel with the new scheduling approach using two different benchmarks, to get the widest view we could:
1. SPEC – cpu2000 [21]. The SPEC manual explicitly notes that attempting to run the suite with less than 256Mbytes of memory will cause a measuring of the paging system speed instead of the CPU speed. This suits us well, because our aim is precisely to measure the paging system speed.
2. A synthetic benchmark that forks processes which demand a constant number of pages – 8MBytes. The processes use the memory in a random access; therefore they cause thrashing. This benchmark was tested within the range of 16MBytes-136MBytes. The parent process forks processes whose total size is the required one, and collects the information from the children. Let us denote this test by SYN8.

The benchmarks were executed on a Pentium 400MHZ machine, with 128MBytes of internal memory and 512KBytes of cache, running Redhat 9.2 kernel version 2.4.20-8. The size of the page was 4KBytes and one tick of the scheduler was 10.5 milliseconds. It should be noted that even though the platform machine had 128MBytes of physical memory, we should take into the bin size considerations that a certain portion of this memory is occupied by the daemons of Linux/RedHat and the X-windows, plus the kernel itself along with its threads, so after an evaluation of the extra size, we used bins of 96MBytes.

5.2 Execution Time

Figure 1a and Figure 1b show the performance of the synthetic benchmark SYN8. Figure 1a shows the number of swaps that were performed in both the schedulers as a function of the total size of the processes, while Figure 1b shows the execution time of SYN8 as a function of the same processes' total size. In this figure, the time slice of the medium-term scheduler was 2 seconds.

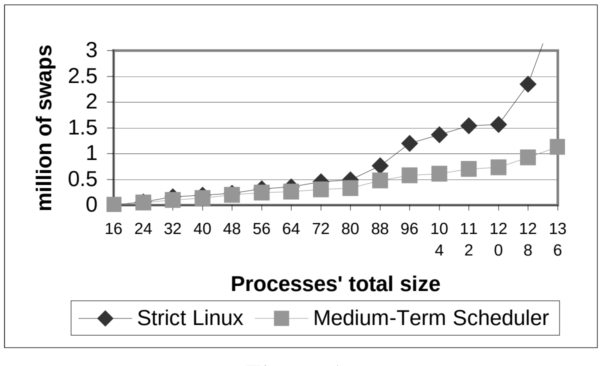

Figure 1a

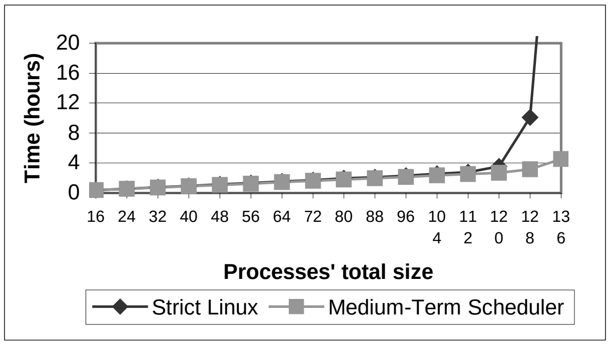

Figure 1b

From roughly 64MB Linux swaps more pages, but there is no noteworthy influence on the I/O time, because Linux lets other processes run while the I/O is performed. Roughly from 128MB the I/O buffer is incapable of responding to all the paging requests. The medium-term scheduler dramatically reduces the number of the page faults; thus less swaps are performed and the execution time remains reasonable. Processes that require 144MB were sustainable for the medium-term scheduler, but not for the Linux scheduler. Actually, after 7 days of running on the Linux scheduler, we killed the processes.

Figure 2a and Figure 2b show the performance of the medium-term scheduler vs. the Linux kernel using the tests of SPEC cpu2000 benchmarks. The prefix 3 (or 2) before the test name indicates that we iterated the test 3 (or 2) times. Sometimes we divided the numbers by some constants in order to fit the data to the scale of the diagram. These constants are denoted as Test/Constant. When we used more than one test, we added a '+' sign between the names of the tests.

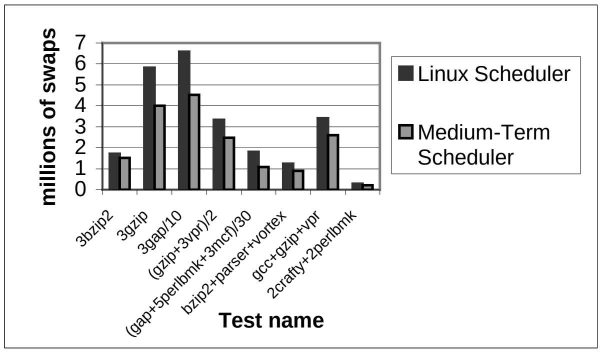

Figure 2a

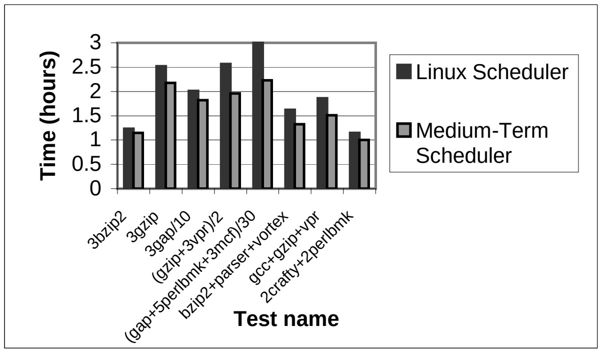

Figure 2b

When each group contains just a few big processes, the idle task might be invoked too often, even though there are other processes that can be executed. This can reduce the time saved by eliminating the thrashing effect. When the test was big, and was executed in a different group, the results were not as good as when executing several smaller SPEC tests concurrently in one group, due to the higher idle time when the content of each group is just one process. Thus the results of Figures 2a and 2b are not as good as the results of Figures 1a and 1b. However, the elimination of the thrashing saved more time than was wasted idling, and the medium-term scheduler still outperformed the traditional Linux scheduler.

Figure 3a and Figure 3b show the effect of the medium-term scheduler time slice on the process' execution time. The tests were conducted using SPEC. It can be seen that when the time slice exceeds a certain limit, the execution time might suffer. This damage is caused by the higher average idle time. When the number of processes per group is too small, a situation that all of the processes in the current group will not be on the Ready queue can occur. Such a case can happen due to a lot of I/O operations. Clearly, this might happen with a lower group time slice as well, but it will not happen as often as with a higher time slice, because at the beginning of the time slice all the processes are ready to run and not waiting for an I/O. Also, new processes are not admitted until the Round-Robin of the medium-term scheduler finishes a complete cycle.

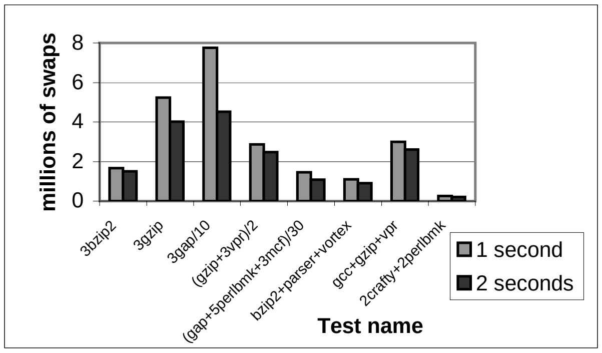

Figure 3a

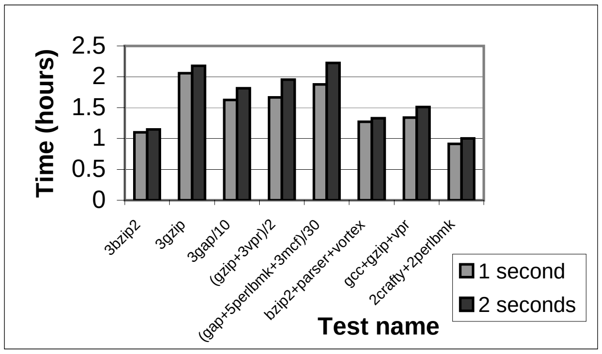

Figure 3b

An extremely high time slice will actually make the medium-term scheduler behave like a FIFO scheduler. On the other hand, the page faults rate is lower for the 2-second scheduler, because of the longer time slice.

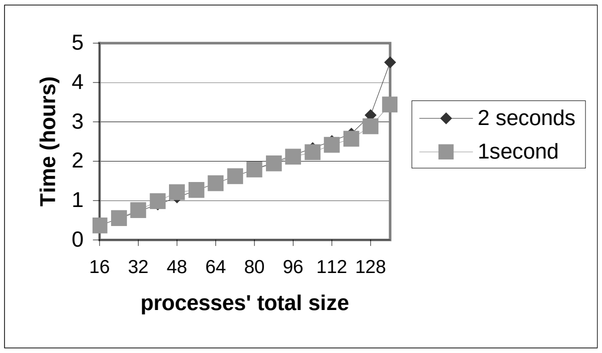

Figure 4

Pages are usually swapped when the group context is switched, so if all the pages are replaced on context switch, the one-second scheduler should have twice the number of pages faults the two-second scheduler has. However, sometimes the bins are not full, and some shared memory can be present, so the ratio between the rates is actually less than 2.

Figure 4 shows the same time slices but with more processes. This test was conducted using the synthetic benchmark SYN8. It can clearly be seen that the effect of increasing the time slice damages the execution time when processes for more than one bin are present.

6 Conclusions and Future Work

The results of the experiments are promising. Given a high memory load used by some processes, the medium-term scheduler can drastically reduce the thrashing overhead. In addition, no decline in the performance is observed when the load is low and no swapping is needed. The medium-term scheduler has been written as a patch to the kernel and can be easily installed on any Linux machine. Such an installation can help the machine handling the massive paging in a more tolerant way than killing processes. Moreover, the responsiveness keeps being reasonable for heavier load. The medium-term scheduler does not require special resources or extensive needs; hence it can be easily adapted by many Linux machines. Moreover, there is no obstruction to implement the medium-term scheduler on a cluster; hence heavy load projects like the Human Genome Project can benefit from such a kernel.

In the future we would like to check the shared memory effect on the medium term scheduler and how different priority of the processes can influence the medium term scheduler. In addition, we would like to check the performance of some approximations for the Bin Packing problem and even to adaptively change the approximation according to the current conditions in the system. In addition, we would like to dynamically change the group time slice of the medium-term scheduler. This feature can improve the performance when there are too few processes and the idle process is invoked too often.

References

[1] Denning P., Virtual Memory, ACM Computing Surveys, Vol. 2(3), ACM Press, NY, USA, pp. 153-189, 1970.

[2] Belady, L. A. A Study of Replacement Algorithms for Virtual Storage Computers, IBM System Journal Vol. 5(2), pp. 78-101, 1966.

[3] Abrossimov V., Rozier M. and Shapiro M., Virtual Memory Management for Operating System Kernels, Proceedings of the 12th ACM Symposium on Operating Systems Principles, Litchfield Park, AZ, December 3-6, pp. 123-126, 1989.

[4] Galvin P. B. and Silberschatz A., Operating System Concepts (Sixth Edition), Addison Wesley, MA, USA, 1998.

[5] Jiang S. and Zhang X., Adaptive Page Replacement to Protect Thrashing in Linux, Proceedings of the 5th USENIX Annual Linux Showcase and Conference, (ALS'01), Oakland, California, November 5-10, pp. 143-151, 2001.

[6] Jiang S. and Zhang X., TPF: a System Thrashing Protection Facility, Software - Practice & Experience, Vol. 32, Issue 3, pp. 295-318, 2002.

[7] Batat A. and Feitelson, D. G. Gang scheduling with memory considerations, In 14th Intl. Parallel and Distributed Processing Symp., pp. 109-114, May 2000.

[8] Nikolopoulos D. S., Malleable Memory Mapping: User-Level Control of Memory Bounds for Effective Program Adaptation, Proc. of the 17th International Parallel and Distributed Processing Symposium (IPDPS'2003), Nice, France, April 2003.

[9] Card R., Dumas E. and Mevel F., The Linux Kernel Book, John Wiley & Sons, New York, N.Y., 1998.

[10] Gorman M., Understanding The Linux Virtual Memory Management, B. Peren's Open Book Series, Chapter 13, 2004.

[11] Marti D., System Development Jump Start Class, Linux Journal 7, 2002.

[12] Scholl A., Klein R. and Jurgens, BISON: A Fast Hybrid Procedure for Exactly Solving the One-Dimensional Bin Packing Problem, Computers and Operations Research 24, pp. 627-645, 1997.

[13] Karp R. M., Reducibility Among Combinatorial Problems, Complexity of Computer Computations (R.E. Miller and J.M. Thatcher, eds), Plenum Press, pp. 85-103, 1972.

[14] Fekete S. P. and Schepers J., New Classes of Fast Lower Bounds for Bin Packing Problems, Mathematical Programming 91(1) pp. 11-31, 2001.

[15] Gent I., Heuristic Solution of Open Bin Packing Problems, Journal of Heuristics 3 pp. 299-304, 1998.

[16] Martello S. and Toth P., Lower Bounds and Reduction Procedures for the Bin Packing Problem, Discrete Applied Mathematics 28, pp. 59-70, 1990.

[17] Coffman Jr. E. G., Garey M. R., and Johnson D. S., Approximation Algorithms for Bin Packing: A Survey, Approximation Algorithms for NP-Hard Problems, D. Hochbaum (editor), PWS Publishing, Boston pp. 46-93, 1997.

[18] Albers, S. and Mitzenmacher, M. Average-Case Analyses of First Fit and Random Fit Bin Packing. Random Structures Alg. 16, pp. 240-259, 2000.

[19] Stallings W., Operating Systems Internals and Design Principles, 3rd Edition, Prentice-Hall, New-Jersey, p. 383, 1998.

[20] Etsion Y., Tsafrir,D. and Feitelson D. G., "Desktop Scheduling: How Can We Know What the User Wants?". In the 14th ACM Intl. Workshop on Network & Operating Systems Support for Digital Audio & Video (NOSSDAV), 2004.

[21] SPEC, Standard Performance Evaluation Corporation, Warrenton, Virginia, http://www.spec.org/.