A thread scheduling technique for avoiding page-thrashing

Summary of Master's Thesis defended by Manoussos-Gavriil Athanassoulis on March 2008

***Abstract*—Page thrashing is the process utilization collapse due to increase of the multiprogramming level. Several papers have discussed this matter up to know, focusing on specific aspects of the phenomenon in order to address it. As a result the effect of such techniques depends on the type of the processor's load. There two main issues in addressing page-thrashing: time scheduling (process and thread scheduling) and memory management. In this text a distributed thread scheduling technique is designed. The proposed technique is based on decisions made by each process when scheduling the contained threads, utilizing data that a process has already available. The proposed thread scheduling technique is simulated using a discrete time simulator system.**

I. Introduction

Page thrashing is the result of the increase of multiprogramming level without having the necessary resources to support it. This phenomenon can happen to every computer and it causes very bad user experience and utilization of the computer resources.

In this text, a new thread scheduling mechanism was introduced, implemented and studied. The proposed interprocess mechanism assigns a utility at every thread of each process every time the parent process is scheduled, and the subsequent thread-scheduling decision is based upon the assigned utilities. The utility value is based on the contribution of each thread on the increase or the decrease of the system's condition metrics (processor utilization and page fault rate). Depending on the system's conditions either the thread that maximizes or the thread that minimizes the utility value is chosen to be executed. The core idea is to schedule threads that cause low page fault rates and high processor utilization when the system's conditions are hard and the opposite when the system's conditions are normal.

The designed mechanism is a distributed paradigm of thread scheduling technique aiming at the reduction of page-thrashing. In the end of the thesis, simulation results are presented, where it is shown that the proposed mechanism increases the processor utilization and reduces the average turnaround time.

II. Prior and Related Work

In the literature one can find several papers addressing the problem of page-thrashing by proposing either different time scheduling techniques or different memory management algorithms.

In [5] a balancing mechanism between the interval of two consecutive page faults and the serving time of a page fault in order to achieve higher processor utilization. *Load control policy* is technique to reduce the intensity of page thrashing by select for execution processes, for which the resident set is almost as big as the working set. Carr proposed another approach of addressing the issue of page thrashing by adapting the clock page replacement algorithm. Carr [8] describes a technique that involves monitoring of the rate at which the pointer scans the circular buffer of frames. According to this rate the level of multiprogramming is tuned aiming at the maximization of the processor utilization. Another family of mechanisms preventing page-thrashing is *Process Suspension*. If the degree of multiprogramming needs to be reduced a process must be chosen according to one of the criteria: lowest-priority process, faulting process, last process activated, process with the smallest working set, largest process and process with the largest remaining execution window.

Jiang and Zhang [3] designed a Linux kernel patch which permits a faulting process to load its working set if the processor utilization is low and there are more than one process with high page fault rate. The core idea behind letting a faulting process to continue executing and loading pages is that after establishing the working set the process will consume CPU cycles and the system will start functioning at higher processor utilization ratio.

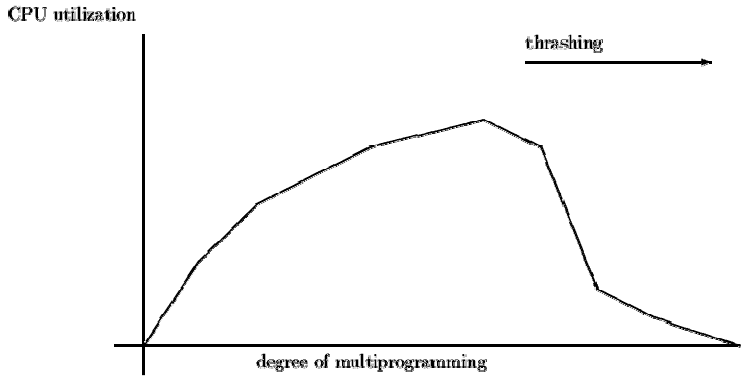

Figure 1: Qualitative graph of processor utilization at the occurrence of page-thrashing

III. THREAD SCHEDULING

In this thesis a thread scheduling mechanism was proposed, designed and simulated. This mechanism utilizes data which each process has already available in order to perform priority scheduling of the threads contained by each process. A two-level scheduling is supposed: process scheduling using Round-Robin or Multi-Level Feedback Queues and thread scheduling performed by each process. This is a scheduling paradigm that is encountered in Unix-type operating systems, where threads are implemented in user-level by packages like pthread.

Thread scheduling is performed each time a process is about to be executed. If a process has more than one thread, then a *Thread Priority Function* (TPF) gives each ready thread an arithmetic value as priority and the thread which has maximum priority (thus minimum arithmetic value, like in Unix) is chosen to be executed.

A. Thread Priority Function

The arithmetic value of a thread's priority given by TPF is computed based on the contribution of the current thread on the metrics describing the conditions of the system.

These metrics are process utilization, which will be represented as CPU and page fault rate, which will be represented as PFR. Depending on the exact value of these two metrics the system can be either on *normal* conditions or *hard* conditions. The condition for determining the system condition is the "violation" of two thresholds: CPU_LOWER_BOUND and PFR_UPPER_BOUND.

$$\text{System Conditions} = \begin{cases} normal, & \text{PFR} > \text{PFR\_UPPER\_BOUND and CPU} < \text{CPU\_LOWER\_BOUND} \\ hard, & \text{otherwise} \end{cases}$$

The reason that both thresholds have to be violated is that the violation of only one threshold cannot ensure that the system faces page-thrashing. Low process utilization can be the result of having no processes ready to consume CPU cycles either because the existing processes are not CPU bound, or because there no ready processes. On the other hand, high page fault rate can be the result of a new process being loaded. In both cases the system did not experience page-thrashing but one condition could have been violated. Only if both thresholds are violated one may assume that page-thrashing occurs.

Before describing the way of computing the value of TPF, the contribution of a thread *t* at time *i* in the

variation of processor utilization (typed as CCU) and in the variation of page fault rate (typed as CPFR) must be defined.

Thus, CCU is the mean proportional of the change of the CPU utilization at each quantum of execution:

$$ccu(t,i) = \alpha \cdot ccu(t,i-1) + (1-\alpha) \cdot \left(CPU(i) - CPU(i-1)\right)$$

where $CPU(i)$ is processor utilization at the end of current quantum and $CPU(i-1)$ is processor utilization at the end of the previous quantum. Respectively, CPFR is the mean proportional of the change of the PFR at each quantum of execution:

$$cpfr(t,i) = \alpha \cdot cpfr(t,i-1) + (1-\alpha) \cdot \left(PFR(i) - PFR(i-1)\right)$$

where $PFR(i)$ is page fault rate at the end of current quantum and $PFR(i-1)$ is page fault rate at the end of the previous quantum. The corresponding metrics of the process $cpfr(p,i)$ are $ccu(p,i)$ computed in order to normalize the values of $ccu(t,i)$ and $cpfr(t,i)$.

The computation of Thread Priority Function is given by:

$$tpf(t,i) = sign(sc) \cdot \left(w_1 \cdot \frac{ccu(t,i)}{ccu(p,i)} - w_2 \cdot \frac{cpf(t,i)}{cpf(p,i)} \right), \; w_i > 0 \qquad sign(sc) = \begin{cases} +1, sc = normal \\ -1, sc = hard \end{cases}$$

Sign function $sign(sc)$ is equal to -1 or +1 depending on the system conditions. This function is necessary because the value of the equation

$$\left(w_1 \cdot \frac{ccu(t,i)}{ccu(p,i)} - w_2 \cdot \frac{cpf(t,i)}{cpf(p,i)} \right)$$

represents the contribution of thread *t*, up to now, to the system conditions. As greater this value is, so better is the contribution to the metrics' values and, as a result, to the system conditions. The sign function is used in order to promote threads with bad contribution in case of normal system conditions. This is important in order to achieve fairness between threads and to avoid starvation.

IV. Simulation of the Proposed Thread Scheduling Mechanism

The proposed mechanism was simulated using a discrete time simulating system. The system at every given step can:

- Be idle (Idle)
- Execute a thread (CPU)
- Switch to another thread (Context Switch)
- Schedule Input/Output (Doing I/O)

In the following figure, the aforementioned states along with some major actions are depicted. The rectangles represent the states and the shapes with smooth or no edges, represents actions that are taken between state transitions. At each minimum time step (which is called "tick") a single transition from one state two another can take place.

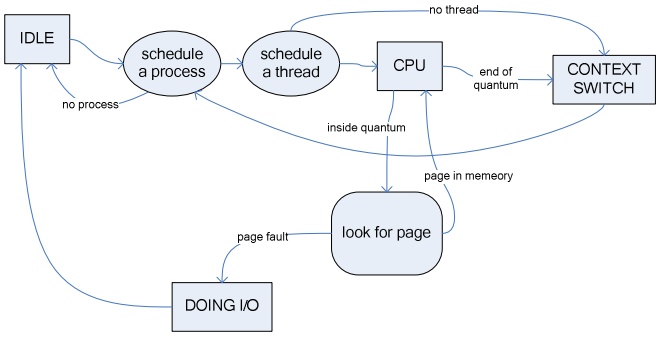

Figure 2: Simulation States

The actions “schedule a process” and “schedule a thread” show the fact that two-level scheduling is performed by the proposed system. Action “look for a page” tries to retrieve a page from main memory and if the page is not yet loaded, a page fault occurs. Then, the page is being loaded, but the thread (and as a result the process) is swapped out.

During the simulation, a different procedure is responsible for creating new threads and processes. The intercreation time between two threads is computed utilizing a random variable distributed according to the exponential probability distribution function. The execution duration of each thread and the process size in pages are two more parameters of the conducted simulation which values are random variables that are distributed according to the exponential distribution.

The mechanism computing the reference string of a thread is also based on the exponential distribution. Suppose that a process has n pages, then for each thread, these n pages are listed in order. Using the exponential distribution with mean $\sqrt{n}$, a number between 1 and n is picked. The page at that point is the next page in the reference string. The selected page is moved at the beginning of the list and the described algorithm is repeated in order to form the rest of the reference string.

V. Results - Conclusions

The described simulation was used for a series of experiments. More than eight different processor and workload setups were simulated. The work load was either created at the beginning of the simulation or created both at the beginning and during the simulation. The chosen values of the parameters of the simulation created page-thrashing and, as a result, the proposed thread scheduling technique was utilized by the simulation system. The average processor utilization measured at the experiments using the proposed technique compared to the average processor utilization measured using round-robin thread scheduling was increased by 5% to 22% and, respectively, the decrease of page fault ratio varied between 6% and 26%. Thread penalty ratio was always decreased when the proposed mechanism was utilized.

Very interesting results provides the experiment during which the time-series of the values of the system metrics where stored and the execution of the created workload was simulated until the end of all processes. As the reader can see in Figure 3, the utilization of the proposed scheme resulted in higher processor utilization, lower page fault ratio, as well as, lower average response time. The red lines are the metrics of the system when the proposed thread scheduling technique was used and with blue lines the metrics of the system when round-robin thread scheduling was chosen are depicted. Another very important result of the proposed mechanism is the reduction of the oscillations of the system performance.

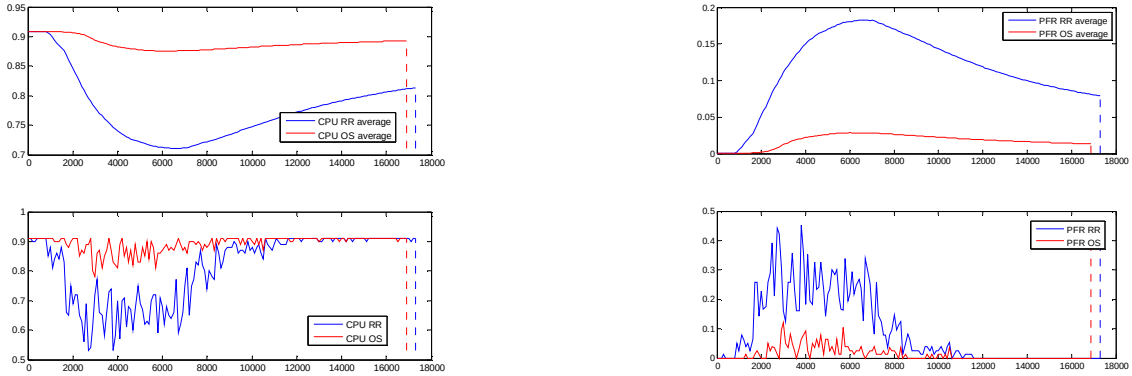

Figure 3: Average and current processor utilization and page fault ratio

VI. Future Work

In this text I presented a distributed paradigm of thread scheduling addressing the page-thrashing issue. It is interesting to use a theoretic tool to model the proposed technique. Game Theory is a classical theoretic tool for modeling resource sharing techniques. The hawk-dove game described in [15] is a possibly suitable game in which each player shows either a more aggressive strategy or a more submissive strategy in order to finally use the shared resource.

Moreover, it would be very interesting to combine the proposed thread scheduling technique with a different page replacement policy than LRU. The LRU memory management algorithm is often used but it is also a not so good strategy to address page-thrashing. The combination of a different page replacement policy (perhaps an adaptive one) with the proposed mechanism may lead to better results.

Finally, it is very interesting to implement the presented scheme in a user-level package implementing thread in order to see the implications and the results of utilizing this mechanism upon real workload and system conditions.

References

[1] Stallings William, “Operating Systems: Internal and Design Principles”, 5th edition, Prentice Hall, 2005
[2] Sudo Y., Suzuki S. and Shibayama S., “Distributed-Thread Scheduling Methods for Reducing Page-Thrashing”, HPDC 1997: 356-364
[3] Jiang S. and Zhang X., “Adaptive Page Replacement to Protect Thrashing in Linux”, 5th Annual Linux Showcase & Conference, November 2001
[4] Jiang S. and Zhang X., “TPF: a dynamic system thrashing protection facility”, In Software - Practice & Experience, 32 (3): 295-318, March 2002
[5] Finkel Raphael, “An Operating Systems Vade Mecum”, Prentice Hall, 1986
[6] Denning P., “Working Sets Past and Present”, IEEE Transactions on Software Engineering, January 1980
[7] Leroudier J. and Potier D., “Principles of Optimality for Multiprogramming, Proceedings”, International Symposium on Computer Performance Modeling, Measurement, and Evaluation, March 1976
[8] Carr R., “Virtual Memory Management”, Ann Arbor, MI: UMI Research Press, 1984
[9] Baer J., “Computer System Architecture”, Rockville, MD: Computer Science Press, 1980
[10] Belady L., “A Study of Replacement Algorithms for a Virtual Storage Computer”, IBM Systems Journal, No. 2 1966
[11] Duda K. and Cheriton D., “Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler”, SOSP 1999: 261-276
[12] Jain Rah, “Art of Computer Systems Performance Analysis Techniques For Experimental Design Measurements Simulation And Modeling”, Wiley Computer Publishing, 1991
[13] Fortier P. and Michel H., “Computer Systems Performance Evaluation and Prediction”, Digital Press, 2003
[14] Saucier Richard, “Computer Generation of Statistical Distributions”, Army Research Laboratory, 2000
[15] Hargreaves-Heap S. and Varoufakis Y., “Game Theory: A Critical Introduction”, Routledge, 1995