2017 2nd International Conferences on Information Technology, Information Systems and Electrical Engineering (ICITISEE), Yogyakarta, Indonesia

A Biological-like Memory Allocation Scheme using Simulation

Gasydech Lergchinnaboot
Department of Mathematics and Computer Science
Faculty of Science, Chulalongkorn University
Bangkok, Thailand
Gasydech.l@student.chula.ac.th

Peraphon Sophatsathit
Department of Mathematics and Computer Science
Faculty of Science, Chulalongkorn University
Bangkok, Thailand
Peraphon.s@chula.ac.th

***Abstract*—This research proposes a novel memory allocation scheme to efficiently handle memory management. The scheme employs biological behavioral principles of the unicellular life form. At the principal construct lives the cells having limited resources, yet passively operates with little overhead. The proposed scheme imitates this unicellular characterization to execute one task at a time using First-In-First-Out queue. Execution is regulated by a global clock that permits one active task at any given time in memory. Consequently, low overhead memory allocation can be achieved without the need for elaborate scheduling and other supporting algorithms. The most anticipatory benefit is simplicity that permits straightforward technological transfer of the proposed scheme to hardware. The contributions are to systematically mitigate the memory wall and reduce power consumed by memory management activities.**

Keywords—FIFO queue; Memory allocation; Biological-like architecture; Simulation.

I. INTRODUCTION

Memory unit has been one of the biggest problems that plagued microcomputers since their inception in the 1970s. On the contrary, hardware components have been consistently developed to fulfill the memory requirements. However, the progress still could not break through Moore's Law that said approximately every 24 months the number of transistors in a circuit would double [1][2]. Later in 1975, David House had revised this Law to 18 months [3]. In essence, this is approximately 60 percent growth per year, while the efficiency of memory has only improved by 10 percent per year. There is a gap between these memory chipsets and the CPU chipsets that stretches out approximately 50 percent every year. This gap is known as the infamous "Memory Wall" [4] that must be eliminated to maximize system performance. Some conventional solutions are listed below:

- Provide a matching memory bandwidth with CPU performance. This approach helps boost up transferring rate and tighten the gap between those two chipsets.
- Apply efficient memory allocation scheme. This approach provides a good replacement strategy to existing systems. The benefit of choosing this choice is that it could be applied to any existing systems without hardware change but refine logical control of the memory unit.

The above approaches never yield any improvements. The execution discrepancies between processor and memory still persist. One viable solution is to simplify memory access that will compensate for processing speed. To arrive at simplicity of implementation, nature can serve as a solution model basing upon the simplest unicellular life form. The unicellular is a life form that can survive on its limited resources in extreme conditions. It possesses some important major characteristics [5] such as self-contained, independent, simplicity, as well as autonomy.

This research exploits the living activities of this unicellular creature by managing memory access and task execution as a new memory allocation scheme. This is essentially based on first-in first-out (FIFO) queueing discipline to simplify memory allocation process, making use of the above unicellular survival activities to achieve optimal memory management effectiveness.

This paper is organized as follows. Section II recounts some related works that are pertinent to this work. Section III describes the proposed scheme in detail. An experimental simulation is carried out to measure the viability of the proposed scheme, wherein the outcomes are summarized in Section IV. Some research considerations are discussed in Section V. Section VI concludes this research study with potential future work.

II. RELATED WORKS

Kagi et al. [6] addressed memory bandwidth problems that were caused by processor stalled, insufficient memory, or memory utilization. New processing chipsets kept getting faster to gaining both advantages and disadvantages. Obviously, faster CPU takes less execution time, but requires more memory bandwidth. They attempted to solve this problem using latency-reduction technique by combining lockup-free and reschedule of operations. These techniques did not go well as a result of lockup-free that caused bandwidth stalls by allowing more memory requests in a short period of time. Consequently, queueing in memory system would possibly be delayed.

Rixner et al. [7] stated that DRAM accessing operation could cause noticeable effect on both memory throughputs and latency. They proposed a Memory Access Scheduling that reordered process at DRAM level and introduced several policies to cope with every circumstance. When DRAM was ready to operate, it would not require any sophisticated access scheduler. However, when DRAM was busy, a complex access scheduler had to be initiated to reclaim resources by oldest and idle references first. These memory references were represented by six parameters, namely, (1) Valid, (2) Load/Store, (3) Row address, (4) Column address, (5) Data, and (6) State. This technique was governed by supporting policies as follows: (1) In-order, (2) Priority, (3) Open, (4) Closed, (5) Most pending, and (6) Fewest pending. Thus, it could improve 40% bandwidth on application traces and 30% media processing.

Designing high performance computer with single high-performance processor is no longer workable since the number of tasks being executed per unit time yields relatively low throughputs. Moreover, heat also causes performance throttling. To avoid these problems, multicore architecture is adopted. Liu et al. [8] proposed triplet-based architecture to obtain massive communication advantages. However, this implementation required a hardware object table (OT) to deploy the indirect addressing. Consequently, memory utilization boosted up and was easy to extend.

Kish [9] addressed the increase of chip density toward physical limits. The prospect of major causes from speed, size, and heat dissipation would eventually become thermal noise phenomena. All these works served as the forerunners to the proposed scheme.

III. Proposed Scheme

In a unicellular life span, they can survive with limited resources and many adversary conditions and environments. In order to mimic the unicellular survival capability for the architectural design of the proposed memory allocation scheme, some preliminary researches have been conducted to address the following problems.

1. How can memory allocation and access overhead be reduced?

   A unicellular life form performs its activities in a simple sequential process. The process exemplifies a straightforward FIFO discipline to arrange memory allocation and access. Moreover, the fact that forgetfulness causes old activities to be replaced by the new ones leads process replacement to be done in place without having to reclaim the memory space. Hence no memory fragmentation overhead will be involved.

2. How can biological constructs be adapted to memory scheme?

   The simplistic and effective biological process of the unicellular life form so created by nature instills the architectural design of the proposed scheme. By virtue of the solution to the first problem, arrangement of process scheduling and execution can be done in a similar manner. Two types of prioritization will be set up in this scheme, namely, user process pool to hold user or voluntary process and system process pool to hold system or involuntary process. The memory allocation scheme is laid out as depicted in Figure 1, where:

   - S_f and U_f are indices denoting the first available slot for next incoming system and user processes, respectively.
   - S_e and U_e are indices denoting currently executing system and user processes, respectively.
   - S_n and U_n are total amount of spaces that are allocated for system and user processes, respectively.

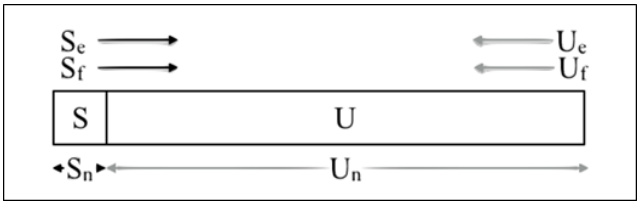

Fig. 1. Memory Allocation Scheme.

Process execution is arranged in the following states, namely, *incoming*, *waiting*, *execution*, and *blocked* as shown in Figure 2. Their characterization is defined as follows:

- Incoming state defines the state that contains newly arrived process being assigned to a specific queue.
- Waiting state defines the state that places those processes in the queue to occupy the available resources.
- Executing state defines the state that a process is running in the processor.
- Blocked state defines the state that a process is held until the required resource is available.

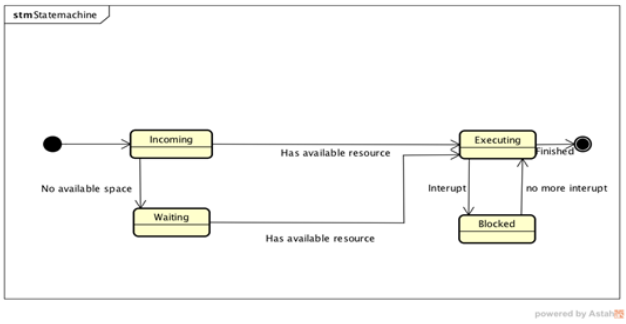

Fig. 2. Process execution arrangement.

The first incoming state denotes a newly arrived process. If there is memory available to hold it, the process will enter the execution queue. Otherwise, it will be put to waiting state.

In waiting state, the processes residing in executing queue are arranged in FIFO order. In blocked state, a timing measure is imposed to control the allowable duration of existence in the memory without progress. Therefore, no process will remain indefinitely blocked in the memory. This is called the Time-To-Live (TTL) factor.

In executing state, the process to be executed is fed either from waiting or blocked state. There will be one process active at a time. The executing process is pointed to by S_e and U_e depending on the type of priority. A process will move out of executing state when all of its subtasks are finished or its allotted TTL is up to prevent the process from holding up the CPU indefinitely [10].

3. How can memory operations be speeded up in order to reduce the Memory Wall problem?

   Referring to Figure 1, separation of user and system pools stems from the fact that there is no definitive criterion to distinguish between voluntary and involuntary processes in the unicellular life form. This arrangement calls for some mechanisms to impose memory allocation priority to both types of processes, while preserving the simplicity of unicellular functions. This analogy is used to speed up the operations of the proposed scheme. The flat architecture is set up as fixed-size memory blocks for fast FIFO access and retrieval. This sequentially ordered blocks can hold processes in execution or replacement by overriding the current process in place. Such a set up help reduce reclamation in as much as 50% of processing overhead on garbage collection or reclamation of existing memory allocation. Consequently, memory leak problem can be avoided.

These preliminary researches lead to the related adaptation of unicellular biological process. First and foremost, memory blocks are partitioned in fixed size to mimic the unicellular construct. Next, space allocation uses FIFO in-place replacement since the unicellular has only limited cell space to operate. From Figure 1, let A and B be system processes and *m* and *n* be user processes in multi-tasking execution. Suppose process *n* is executing. A new system process G arrives but all memory space is fully occupied. How should G be handled? This is a typical scenario that can be solved by many efficient process scheduling algorithms. In this study, process G is assigned to user space U by overriding *m* as illustrated in Figure 3. At the same time, *n* is blocked to relinquish control for G execution.

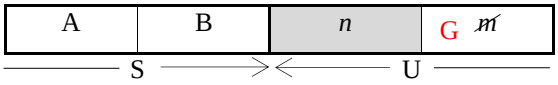

Fig. 3. Process space allocation and replacement.

The above scenario demonstrates the simplicity of allocation and replacement that mimics the uni-cell activities to be performed one activity at a time. As process execution continues, its existence is governed by TTL (set forth by a global clock) in a similar fashion as the new uni-cell is reproduced and the old one dies. A process is replaced by the next one in waiting queue when its TTL expires. If it has yet completed the execution, its contents will be transferred to wait queue in the FIFO manner. Otherwise, no transfer is performed. Hence, at most only two memory transfers take place: one out-going and one incoming

Due to FIFO access and retrieval of processes residing in memory, no process scheduler is required. The flatten consecutive fixed blocks arrangement needs no hierarchical traversal. Thus, logical construct of the proposed scheme boils down to physical linear ordering that could lend itself to hardware implementation. In so doing, operational overheads and power consumption would be reduced, while memory access could be considerably faster. As a consequence, the memory wall problem would gradually be mitigated.

IV. Experimental Results

The simulation was written in Python running on Intel® Core i7 4790, 8GB DRAM DDR3 memory, Ubuntu 16.04 LTS system. Since there were no supporting environments that operated in a similar manner as the proposed scheme, a simulation was performed to verify the viability of the memory allocation scheme. Three operations were exercised to evaluate how the proposed scheme performed in comparison with known algorithms. The three operations were sort, transfer of control, and remove being tested against FIFO, shortest remaining time first (SRTF), and Round-Robin (RR) algorithms. Performance evaluation was measured by (1) the number of processes being generated per simulation run (*freq*), (2) time to run the process (*t*), and (3) process class (*c*), i.e., system or user. The rationale was because they were part of all the candidate algorithms.

From the preliminary researches, the above evaluation parameters were established as follows: *freq* ranged from 1 to 5; *t* spanned 1 to 50; and *c* was proportionated by system to user processes at 1:9. Table 1 summarizes the operations being tested which can be further elucidated below.

TABLE I. Operations Tested

	FIFO	**SRTF**	**RR**	**Proposed Scheme**
Sort	0	$\sum_{i=1}^{N} T_i$	0	0
Transfer of control	T_n	T_n	$\lceil T_n \% t \rceil$	$\lceil T_n \% TTL \rceil$
Remove	T_n	T_n	0	0

TABLE II. TIME USED TO FINISH 10,000 OPERATIONS (IN CLOCK TICKS)

APPROACH	TIME	RESULT	DIFFERENCE
FIFO	$n(\bar{x}+3)$	26815	-31.061%
SRTF	$N \log N + \sum_{i=0}^{N} n(\bar{x}+3)$	74340	+91.120%
RR	$\sum_{i=0}^{N}[P_i \div 5] \times 8$	41631	+7.0288%
PROPOSED SCHEME	$\sum_{i=0}^{N}(\lfloor P_i \div 5 \rfloor \times 8 + (P_i \div 5) + 3)$	38897	$\pm 0\%$

Sort was required only by SRTF since it had to fetch the next process having the smallest remaining time to execute. Transfer of control measured the context switch between processes. Note that execution duration of the proposed scheme was confined by TTL which varied from class to class, while RR applied fixed time slice to all processes. Remove dispatched processes from waiting queue to execution. Simulation run was performed for 10,000 processes. The results are shown in Table 2.

V. DISCUSSION

In this paper, it was obvious that the proposed scheme ran only single thread execution per process type. The rationale behind this implementation was to reduce the number of operations and kept space usage as low as possible.

Consider the results in Table 2, the proposed scheme spent almost half the time of SRTF method and slightly less than that of RR method. The difference was that traditional RR method always operated until the time slice was reached. However, the proposed scheme allowed process to exit as soon as it was finished or TTL expired.

In comparison with FIFO approach, the proposed scheme fell behind because it also incorporated FIFO as part of its operations. Hence, the short-coming of FIFO method became an inherent part of the proposed scheme, i.e., starvation. We took care of this problem using TTL to limit this indefinite wait or blocking to avoid the starvation problem. Nonetheless, the extra context switches caused by TTL expiration lengthened the execution time considerably, hence the excess 31% deficit. This issue will be taken care of in the future work.

From the selected operations being demonstrated, only SRTF required sort operation while the rest did not. Remove operation was required by FIFO and SRTF while RR and the proposed scheme simply loaded the new process in place of the finished one. The transfer of control operation required shifting of control from one process to the next. FIFO and SRTF took the basic execution transfer from start to finish. The proposed scheme and RR, on the other hand, required regular resource occupation from current process to the next when TTL of the proposed scheme or time slice of RR expired.

The above three problems demonstrated how the proposed scheme exploited the strengths of simplicity in unicellular life form to arrive at low overhead and fast memory allocation scheme. In so doing, the memory would become available to be allocated which, in turn, relieved the CPU from execution delay, as well as the memory wall problem.

VI. CONCLUSION AND FUTURE WORKS

In this paper, the proposed scheme employed novel design of memory allocation scheme by introducing a new solution from biological unicellular life form. By adopting traditional FIFO technique and unicellular living characteristics, certain parameters were removed from memory allocation scheme. Consequently, the proposed scheme was still able to perform the necessary functions and yet comparable with standard algorithms at lower overheads.

Furthermore, the proposed scheme was meant to be hardware implementable due to its simplicity. Memory allocation overhead was considerably reduced, in particular, remove operation was done in place. Ultimately, the hardware implementation would speed up running time to lessen memory access delay. Certain look ahead techniques and associative memory could be deployed in future work to reduce the number of context switches, whereby improving the efficiency of FIFO allocation. The gap of memory wall would eventually be mitigated.

REFERENCES

[1] E. Mollick, “Establishing Moore’s law,” *IEEE Ann. Hist. Comput.*, vol. 28, no. 3, pp. 62–75, 2006.

[2] C. A. Mack, “Keynote: Moore’s Law 3.0,” *Microelectron. Electron Devices (WMED), 2013 IEEE Work.*, p. xiii, 2013.

[3] Intel, “Moore’s Law and Intel Innovation,” *Intel*, 2012. [Online]. Available: http://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html. [Accessed: 01-Mar-2017].

[4] S. Derrien and S. Rajopadhye, “FCCMs and the memory wall,” *IEEE Symp. FPGAs Cust. Comput. Mach. Proc.*, vol. 2000–Janua, no. ii, pp. 329–330, 2000.

[5] H. Nozaki, *Sexual Reproduction in Animals and Plants*. 2014.

[6] A. Kagi, J. R. Goodman, D. Burger, J. R. Goodman, A. Kagi, and W. D. Street, “Memory Bandwidth Limitations of Future Microprocessors,” *23rd Annu. Int. Symp. Comput. Archit. ISCA96*, vol. 24, no. 2, pp. 78–89, 1996.

[7] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory access scheduling,” *Proc. 27th Int. Symp. Comput. Archit. (IEEE Cat. No.RS00201)*, vol. 27, no. c, pp. 1–11, 2000.

[8] M. Liu, W. Ji, Z. Wang, J. Li, and X. Pu, “High performance memory management for a multi-core architecture,” *Proc. - IEEE 9th Int. Conf. Comput. Inf. Technol. CIT 2009*, vol. 1, pp. 63–68, 2009.

[9] L. B. Kish, “End of Moore ’ s law : thermal (noise) death of integration in micro and nano electronics,” vol. 305, pp. 144–149, 2002.

[10] W. Stallings, *Operating Systems: Internals and Design Principles*. 2008.