BIOLOGICAL-LIKE MEMORY ALLOCATION SCHEME SIMULATION

Mr. Gasydech Lergchinnaboot

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science Program in Computer Science and Information Technology

Department of Mathematics and Computer Science

Faculty of Science

Chulalongkorn University

Academic Year 2017

การจำลองแผนจัดสรรหน่วยความจำคล้ายเชิงชีววิทยา

นายกษิดิ์เดช ฤกษ์ชินบุตร

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2560

Thesis Title	BIOLOGICAL-LIKE MEMORY ALLOCATION SCHEME SIMULATION
By	Mr. Gasydech Lergchinnaboot
Field of Study	Computer Science and Information Technology
Thesis Advisor	Associate Professor Peraphon Sophatsathit, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

.....Dean of the Faculty of Science

(Associate Professor Polkit Sangvanich, Ph.D.)

THESIS COMMITTEE

.....Chairman

(Professor Chidchanok Lursinsap, Ph.D.)

.....Thesis Advisor

(Associate Professor Peraphon Sophatsathit, Ph.D.)

.....External Examiner

(Assistant Professor Kriengkrai Porkaew, Ph.D.)

กษิดิ์เดช ฤกษ์ชินบุตร : การจำลองแผนจัดสรรหน่วยความจำคล้ายเชิงชีววิทยา (BIOLOGICAL-LIKE MEMORY ALLOCATION SCHEME SIMULATION) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: รศ. ดร. พีระพนธ์ โสพัศสถิตย์, หน้า.

เมื่อ กอร์ดอน มัวร์ ได้สังเกตุรูปแบบการเพิ่มขึ้นของจำนวนทรานซิสเตอร์ และได้พบว่าความกว้างของแถบความถี่ของหน่วยความจำ ไม่สามารถที่จะพัฒนาให้ตามทันกับประสิทธิภาพของหน่วยประมวลผลได้ อัตราความแตกต่างนี้ค่อยๆเพิ่มขึ้นอย่างต่อเนื่อง จึงส่งผลให้เกิดปัญหา “กำแพงของหน่วยความจำ” และผลกระทบนี้ได้ก่อให้เกิดปัญหาประสิทธิภาพคอขวดขนาดใหญ่ หลากหลายวิธีการกำจัดคอขวดได้ถูกจัดทำขึ้น วิธีการเหล่านี้ได้มีการใช้ทรัพยากรและมีความความซับซ้อนสูง การวิจัยนี้ได้เสนอแผนการจองหน่วยความจำแบบใหม่ที่ได้ใช้หลักการพื้นฐานของพฤติกรรมทางชีววิทยาของสิ่งมีชีวิต ขณะที่เซลล์ได้ถูกสร้างมาด้วยข้อจำกัดทางทรัพยากร แต่ทว่ายังสามารถปฏิบัติงานได้อย่างต่อเนื่องโดยใช้ทรัพยากรเพียงเล็กน้อย วิธีการที่นำเสนอเลียนแบบลักษณะพิเศษของสิ่งมีชีวิตเซลล์เดียวที่จะทำงาน 1 งาน ต่อ 1 ช่วงเวลา และได้ใช้หลักการทำงานแบบเข้าก่อนออกก่อน (ไฟโฟ) การประมวลผลสามารถควบคุมได้โดยการใช้นาฬิกาโลกที่จะอนุญาต 1 งาน ต่อ 1 ช่วงเวลา จึงส่งผลให้เกิดแผนการจัดสรรหน่วยความจำที่ใช้ทรัพยากรน้อยและสามารถทำให้บรรลุล่วงได้ โดยที่ไม่ต้องใช้ขั้นตอนวิธีการทำงานที่ซับซ้อน และยังส่งผลให้แผนการจัดการนี้สามารถนำไปประยุกต์ใช้ได้บนฮาร์ดแวร์ซึ่งจะบรรเทาปัญหากำแพงของหน่วยความจำไปที่สุด

ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	ลายมือชื่อนิสิต .....
		ลายมือชื่อ อ.ที่ปรึกษาหลัก .....
สาขาวิชา	วิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ	
ปีการศึกษา	2560	

5872601123 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
KEYWORDS: FIFO QUEUE / MEMORY ALLOCATION / SIMULATION / BIOLOGICAL-LIKE

GASYDECH LERGCHINNABOOT: BIOLOGICAL-LIKE MEMORY ALLOCATION SCHEME SIMULATION. ADVISOR: ASSOC. PROF. PERAPHON SOPHATSATHIT, Ph.D., pp.

When Gordon Moore observed the number of transistor increasing pattern while memory bandwidth could not catch up with processing unit performance, this diverging rate kept stretching out to create what eventually transpired to be “Memory Wall.” This consequence becomes a major performance bottleneck. Many bottleneck elimination approaches have been attempted. They incorporate considerable overhead and high complexity. This research proposes a novel memory allocation scheme that employs biological behavioral principles of the living creatures. At the principal construct of their life form lives the cells having limited resources, yet passively operates with little overhead. The proposed method imitates this unicellular characterization that operates on one task at a time, thereby memory occupation is reduced to First-In-First-Out activation discipline. Processing can thus be regulated by a global clock that permits one active task at any given time to reside in memory. Consequently, low overhead memory allocation scheme can be achieved without the need for elaborate algorithms. The most anticipatory benefit is technological transfer of the proposed scheme to hardware that will eventually alleviate the Memory Wall problem.

Department:	Mathematics and Computer Science	Student's Signature .....
		Advisor's Signature .....
Field of Study:	Computer Science and Information Technology	
Academic Year:	2017	

ACKNOWLEDGEMENTS

I would like to first and foremost express my sincere appreciation to my supervising adviser Associate Professor Dr. Peraphon Sophatsathit. Dr. Peraphon was always there for supporting me with great advices, patient guidance and constant encouragement throughout our two and a half years of working together. Also, when discussions were needed, he was always available to exchange ideas and consistently pushed me further. Besides these helps, Dr. Peraphon had taught me to work under pressure. Without him this thesis could not have been done.

I also would like to thank to my committee, Professor Dr. Chidchanok Lursinsap, and Assistant Professor Dr. Kriengkrai Porkaew, for their valuable times, valuable advice, every suggestion during examination period and my thesis book.

Thank to Graduate School Chulalongkorn University for their 60/40 Support for Tuition Fee and AVIC research center for providing such a stunning working environment and excellent facilities. I would also like to thank all supportive words from my colleagues in AVIC research center, especially Mr. Thitiwat Piyatamrong and Miss Satanat Kitsiranuwat.

Special thanks to very supportive developer community from PyCharm and Stackoverflow that helped me went through struggle and made it here.

Finally, my sincere gratitude goes to my parents who are my life-long loving, caring, and supporting mentors.

CONTENTS

Page

Page

Page

CONTENT OF TABLES

CONTENT OF FIGURES

Chapter 1 INTRODUCTION

1.1 Introduction

Calculation has been around since prehistoric age. Since then, the advancement of calculation technologies has consistently developed. The development rate was incredibly fast, starting from abacus and counting to the introduction of computer. During that time computer technology was not as powerful as it is today. Computer's components such as processing units and memory units were developed to fulfil the customer and manufacturing needs. However, these 2 units did not progress at the same pace. In 1970s, Gordon Moore noticed these phenomena. His observation led to the conclusion that "the number of transistors in circuit will doubling themselves every 24 months"[1] [2]. This statement stayed merely half a decade. In 1975, David House rewrote the statement into 18 months[3]. Moore's law has proven itself to be accurate enough to use as a reference for chipset manufacturer. The law can be translated into numeric form as follows: CPUs grow approximately 60 percent per year, while memory speed improves just 10 percent per year. This gap keeps widening 50 percent per year that subsequently is known by the infamous "Memory Wall"[4]. The problem must be eliminated to maximize overall resources utilities.

There were several attempts to conquer Memory Wall problem. Two possible solutions are:

1. Provide more memory performance to matching CPUs performance. This solution aims to replace existing components with faster, more effective, and more efficient ones. The system could gain a performance boost ranging from slightly change or could be huge jump depending on the available budget. However, there are drawbacks listed below:
   - Costs fortune to replace annually or decennially.
   - New configurations must be involved whenever hardware changes.

2. Hardware advancement that keeps no end in improvement. Employ more efficient memory allocation scheme. This solution aims to utilize existing hardware by deploying better replacement memory allocation scheme. Thus, the existing system remains reusable, no hardware changes except some fine-tuning in memory control unit procedures.

There were many experiments to solve the Memory Wall problem with efficient memory allocation scheme. To date, some of the approaches have worsen the problem with different techniques, performing look ahead techniques which end up to be a miss, not to mention extra memory usage from look up table. The performance gap between CPUs and memory remains.

This research introduces biological knowledge as an innovative approach to mitigate the memory wall problem. One approach to be considered is to exploit a unicellular life form as the basis for design and implementation of a memory allocation scheme. This proposed scheme has to be simple, yet efficient in its own right. The unicellular life form hereafter is alternately referred to as unicellular animal, or unicell fits this desired philosophy and consequently is chosen as the reference architectural model of the proposed memory allocation scheme.

Alongside the simplicity, the unicellular animal can live in various surfaces, extreme weather conditions, survive in their limited resources in enclosing environment. The provision of nature establishes some of their predominant characteristics [5] as follows: independence, self-contain, autonomous, and versatile.

By virtue of the above unicell properties, the reference architectural model must be laid out straightforwardly to preserve the simplicity. In other words, memory arrangement must be organized to permit easy, fast access and retrieval using the simplest algorithm. A viable candidate is First-In, First-Out (FIFO) method. Its straightforward operational construct lends itself to hardware realization which, in terms of the proposed scheme, represents a single unit memory package imitating the unicell structure.

It is envisioned that the proposed scheme will help mitigate the memory wall problem. Detail on how it is derived, designed, and implemented will be discussed in the remaining chapters of this work.

1.2 Statements of the problems.

A number of problems are set up to be explored in this research.

1. How can a biological scheme be imitated in memory context?
2. How can the unicellular animal characteristics be deployed to help solve the memory wall problem?

1.3 Objectives

1. Addressing performance gap between processing units and memory units to solve the memory wall problem with a biological-like memory allocation scheme.
2. Devising efficient algorithms based on the unicellular characteristics to support the proposed memory allocation scheme.
3. Pursuing the hardware-implementable on this work by reducing the process state complexity with operation and space usage reduction.

1.4 Scope of this thesis

1. Define potential existing problems and solutions.
2. Propose a new memory allocation scheme based on unicellular life form.
3. Focus on memory allocation of job distribution as shows in Figure 1.1.
4. Evaluate results, compare proposed scheme to well-known algorithms.

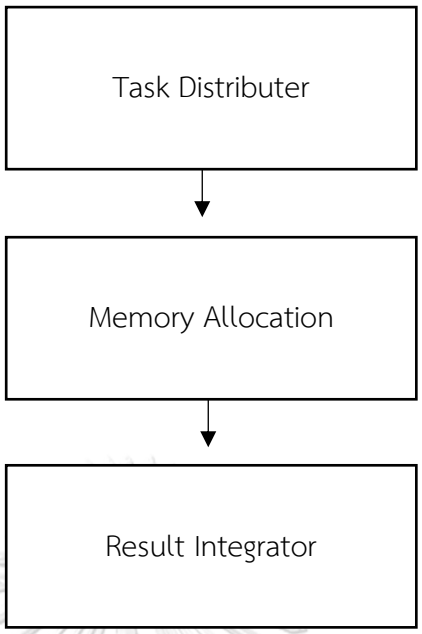


Figure 1.1 Job distribution

1.5 Organization of this thesis

Chapter 2 recaps the background on how unicell biological structure help shape the reference architectural model, the transformations from unicell to process states. Some related works including memory bandwidth limitation, process reordering, single high-performance processor issues, and logical limit problems are described.

The proposed scheme will be explained in Chapter 3. The reference architectural model is presented, as well as modified process states to suit the proposed scheme. In addition to these design elements, management policies will also be described.

Chapter 4 carries out research experimental simulation. Environment set up of both hardware and software will be defined, as well as inputs for this simulation. The experimental results are measured to gauge the viability of proposed scheme by comparing with related benchmarking methods.

Some inferences precipitated from this study will be discussed in Chapter 5, particularly, why the proposed scheme is viably more efficient than other comparable methods. Chapter 6 concludes the thesis along with future improvement on the proposed scheme.

Chapter 2 BACKGROUND KNOWLEDGES AND LITERATURE REVIEWS

2.1 Motivation

As mentioned earlier, there were problems regarding performance gap between CPUs and Memory units. These problems had been around for ages due to their different pace where the CPUs were growing 60 percent a year and memory units were progressing merely 10 percent yearly.

Inconveniently, performance gap kept stretching out as time went by. There were numbers of researches attempted to conquer "Memory wall." Various techniques had involved, namely, reorganizing memory structure, perform look ahead technique, or even put extra physical components. But most of successful ones had shared one similarity, being involved with fancy techniques, which caused extra operations and memory usages.

In this research, different perspectives were investigated about conventional approach to see if conventional approaches would worsen the memory wall problem. Numerous observations were performed including nature, human, animal, also artificial living form that involved from their properties, functionality, as well as behaviors. These mentioned life forms were managed to operate their living basis with and without help depending on activities. Some activities were tough to achieve, the rest were effortless as these activities usually involved resources to accomplish. Even though these life forms had limited functionalities and resources, yet they managed to carry out their living activities.

2.2 Unicell biological structure

Since previous works were not working as expected, alternative solution was investigated. This work was targeting to employ non-computer science knowledge to work alongside with conventional knowledge.

Biology was one choice of the solution. It was found that there were plentiful life forms ranging from well-developed ones to the simplest ones. Well-developed life

forms consist of functions, they can work flawlessly through array of routines, but the complexity is also come with. The inspection has gone down to lower level animals. Multicellular animal's body consists of multi-level structures as illustrate in Figure 2.1.

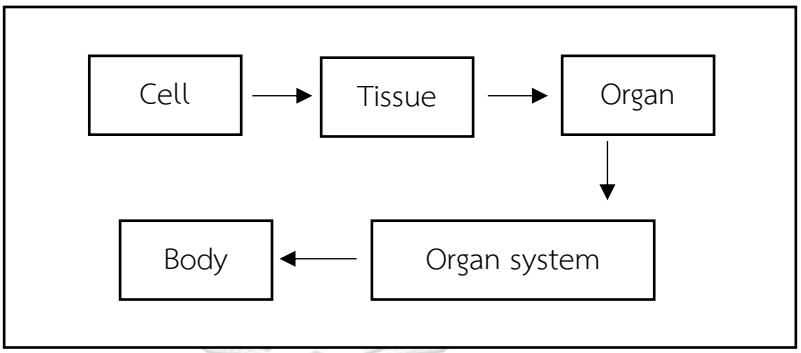


Figure 2.1 Body structure

According to Figure 2.1 multicellular animals are consisting of sophisticated components that work together and become a living body. However, unicell animals consist of only single cell in their boy. Every activity is done by this single cell[5], i.e. digestion, excretion, reproduction, etc. Considering all of resources that are limited, the outcome is astonishingly performed.

Aside from their stunning performance, their other abilities are also impressive as well. Unicell animals are capable of living through variety places even on other animals. Summarizing these properties together, unicellular animal becomes a reference model in this work.

2.3 Operating systems background knowledge

2.3.1 Process states

Processes are program in execution. They are part of the entire system. There are several of process types based on events, CPU bound and I/O bound processes. No matter which type of processes, they could be concurrently handled by the operating systems. Every process will be assigned to one of the states, typically consisting of the following 5 states[6]: new, ready, running, blocked, and terminated as depicted in Figure 2.1.

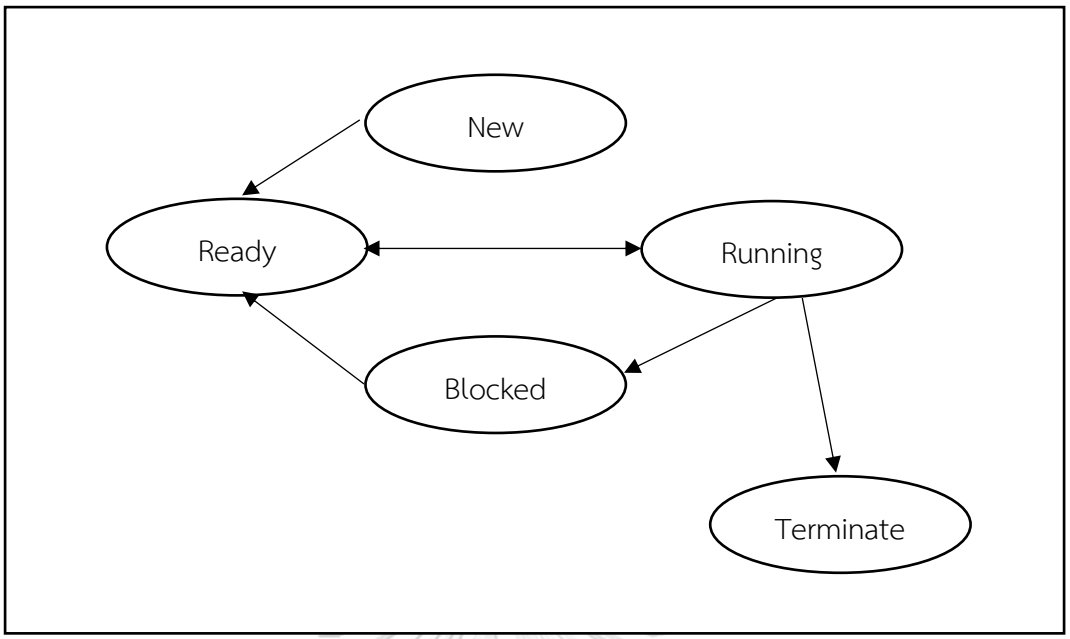


Figure 2.2 Process states

1. New: freshly created process yet to settle or grant a permission to specific resources.
2. Ready: processes that settle in memory yet to execute. Processes in this state are waiting to their execution iteration.
3. Running: processes that are executing.
4. Blocked: Interrupted processes by events which caused by insufficient resources, high priority process, or waiting for I/O devices.
5. Terminated: processes completed from Running, occasionally, from ready or blocked state. Some processes remain in memory even the corresponding jobs are finished.

2.3.2 Scheduling

2.3.2.1 First-In First-out (FIFO)

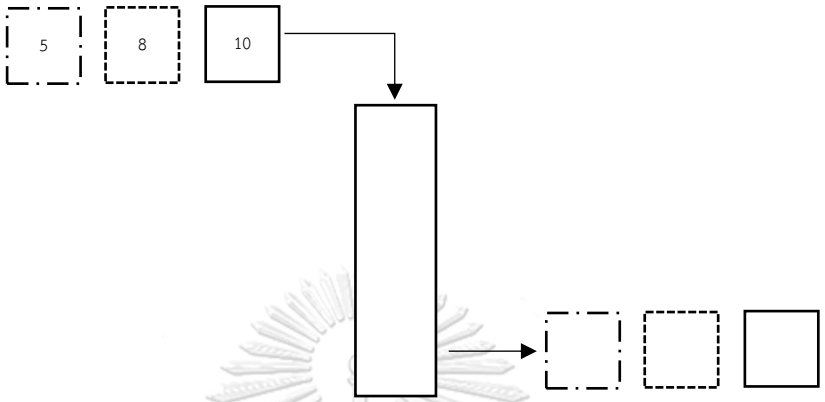


Figure 2.3 First-In First-Out (FIFO)

FIFO is the simplest and most straightforward method that serves the oldest entry first and runs until the job has finished. Then the second oldest is run and so on until no process is left in process queue as demonstrated in Figure 2.2. No extra operations are required to operate this scheduling method.

2.3.2.2 Shortest Remaining Time First (SRTF)

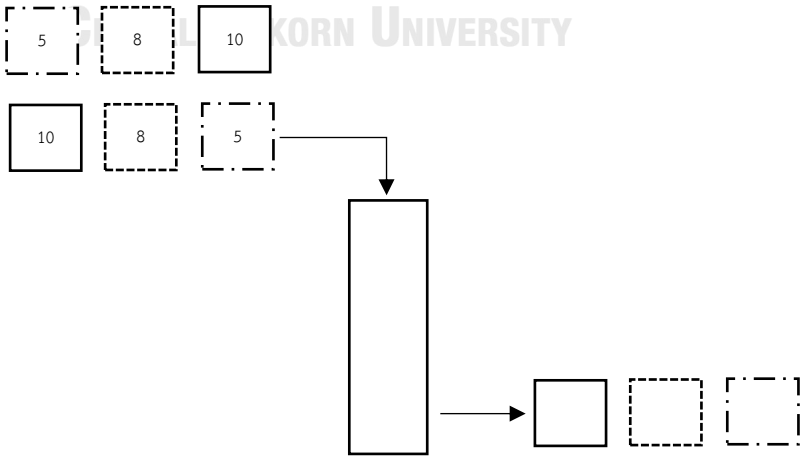


Figure 2.4 Shortest Remaining Time First (SRTF)

SRTF picks the shortest remaining time process first to run. The next iteration will sort all remaining processes. Then the procedure repeats until no process is left in the queue. This is demonstrated in Figure 2.3.

2.3.2.3 Round Robin (RR)

The first two approaches could face starvation problem. For FIFO, if the queue is long, the processes at the end of the queue would never get their turn to execute. For SRTF, if all incoming processes are smaller than the current one, it also never be granted resources. RR fixes this problem by granting limited time window for execution. When the time window expires, the resource will be shifted to the next process and so on.

2.4 Literature Reviews

2.4.1 Memory Bandwidth Limitation

Kagi et al. [7] addressed upcoming memory-related problems that would occur within decades. With present technology, the numbers of instructions that could be executed per unit time was already high. If the development trends kept increasing, CPUs utilization would decrease. Unfortunately, the research trends remained the same by trying to push as many instructions as possible through memory bandwidth. This would worsen memory problems because CPUs took less time to complete its jobs. Consequently, memory bandwidth would increase its usage that was already limited. Whenever this problem occurred, it was hard to determine whether the problem was originated from memory-related processors stalled or insufficient memory bandwidth.

They attempted to solve this problem by focusing on hit/miss rate at cache level. They tried to utilize loaded instructions and avoid missed load instructions by helping lookup-free cache, software and hardware prefetching techniques. However, this technique did not live up to the expectancy, memory stall problems still occurred but had slightly decreased.

2.4.2 Scheduling Policies

This research also focused on hardware utilization which was manageable on existing hardware components particularly on DRAM level. Rixner et al. [8] declared that memory access scheduling could be used to optimize memory system performance by rearranging operations in DRAM. It could make a big impact on both memory throughput and latency. To rearrange operations in DRAM encompassing pre-charge, activate, read, and write, the policies had to support the following arrangements: (1) in-order, (2) priority, (3) open, (4) close, (5) most pending, and (6) fewest pending. These policies were interacted with memory references, which represented by 6 parameters as follows: (1) valid (2) load and store (3) row address (4) column address (5) data and (6) additional state if required any.

2.4.3 Object Table introduction

This time strategy was moved to physical components by focusing on hardware architecture. Making high performance machine with only single high-performance was not applicable any more. Single process was not scalable in performance-wise and also could face parallelism issues eventually. If attempting to squeeze out performance from single processor, overclocking was the only choice. Nevertheless, it could potentially end up with throttling problems, only air-cooled unit could not handle.

Therefore Liu et al. [9] introduced triplet-based architecture. Naturally, this architecture style was already multi-core ready, and easy to expand in every aspect. In addition, triplet-based architecture had performance boost when problem structure matched with communication structure. However, triplet-based required hardware object table (OT) for implementation to communicate with indirect addressing which was questionable.

2.4.4 End of Moore's law

Kish [10] mentioned about shrinking of transistor sizes, while further computer chips density increment would face a physical limit. The expected range problem immerged at 40 nanometers. The miniaturization would face energy dissipation when sizes were met at certain point. Thermal noise would result which caused crossing of logic threshold voltage. Consequently, it could create false bit flip. However, in their experiment safe range was defined with fractions of threshold amplitude limit and thermal noise voltage was less than 12, where threshold amplitude limit was equal to 0.6.

Chapter 3 METHOD

3.1 Observation

The unicellular animals lead us to a new level of operating process and memory functions ranging from basic living to reproduction. This characteristic offers a few distinctive properties, namely, Versatility, Self-contained, Autonomous, and Simplicity. For versatility, unicellular animals can live everywhere from the ocean, forest, boiling hot desert, or even on other animals as parasite. Various varieties are adaptable through climate changes or even diverse terrains.

Unicellular animals can reproduce by asexual reproduction. Reproduction process can be done by fission, budding, fragmentation, etc. Thus, it expands the limited resources to be unlimited. This principle will be exploited in the proposed scheme design by reusing memory blocks without removal or involving sophisticated replacement algorithms.

Simplicity property of unicellular animals allows them to live and perform all the necessary activities. This property will be a mandate for the reference architecture design and implementation.

To obtain the most benefits from these mentioned properties, FIFO queue is employed to reduce memory usage and memory reference as many as possible. This will benefit for several reasons. First, processes are stored in memory in the FIFO manner. Second, it requires no extra operation to handle these processes. Third, starvation can be avoided with the help of a threshold Time-to-Live (TTL) to keep the processes rolling. The reference architecture is depicted in Figure 3.1

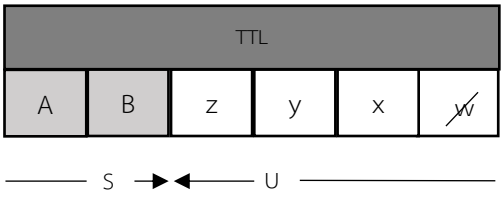


Figure 3.1 Reference architecture

3.2 System Architecture

3.2.1 Schematic operations

The following sections describe software design, development, and pertaining deployment policies of the proposed system. Figure 3.2 shows the class diagram of overall system design.

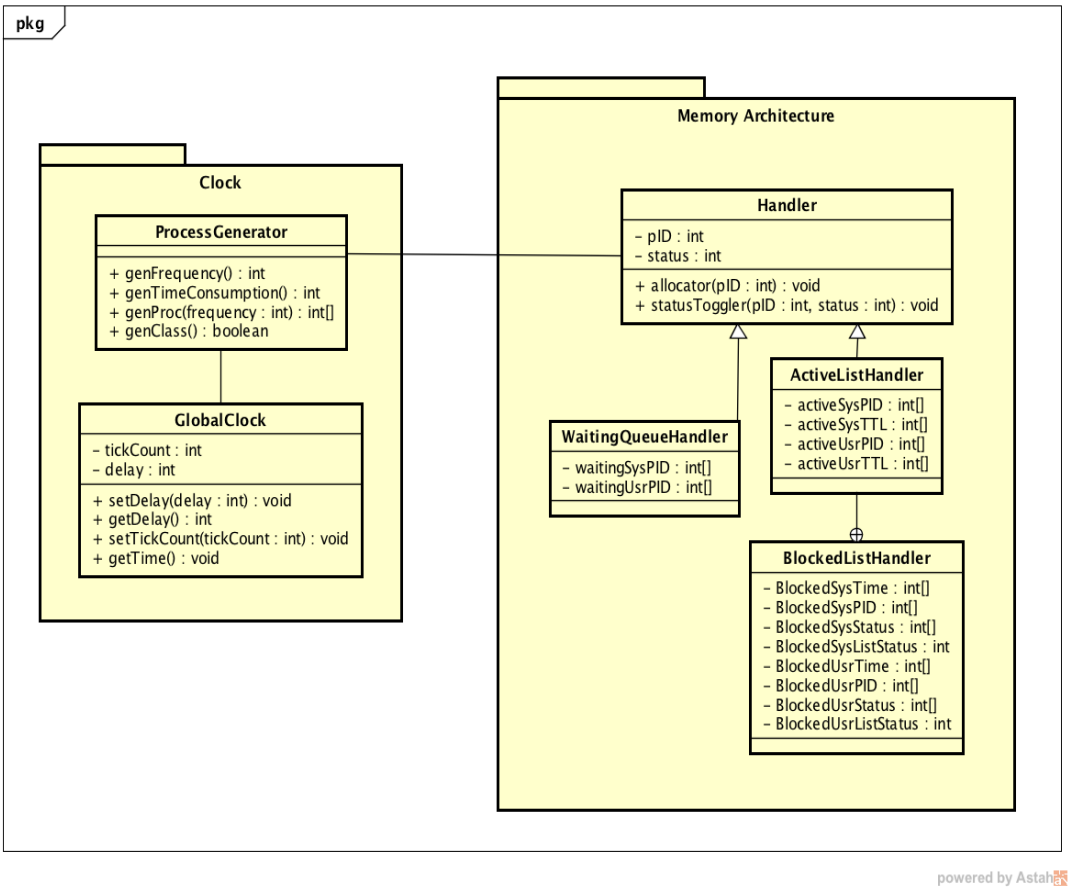


Figure 3.2 Class Diagram

To achieve minimal memory usage as less as possible, the number of operations and variables usage must be minimized. This system consists of 2 packages, namely, Clock and Memory. Clock is made up of process generator and global clock classes. Clock package focuses on pre-execution phase, dealing with input and time keeping. The memory architecture package describes the resource pool, the size of memory pool, status of each processes and its handlers.

The process generator class encompasses these operations to manage process operation, namely, genFrequency, genTimeConsumption, genClass, genProc, and clock tick.

```
def genFrequency(self):
    frequency = randint(1, 5)
    return frequency
```

When the main function triggers process generator, the sequence of operations starts with genFrequency having the burst number running from 1 to 5.

```
def genTimeConsumption(self):
    time = randint(1, 50)
    return frequency
```

Then genTimeConsumption is invoked to set up the duration of requested time for each sub processes. The value ranges from 1 to 50 clock ticks.

```
def genClass(self, val):
    usr = [True, True, True, True, True, True, True, True, True]
    sys = [False]
    type = sample(usr+sys, val)
    return type
```

Next, genClass is started to mark individual process as user and system, indicated by Boolean. This operation takes one parameter “val” to determine the number of members to be generated for the process list.

These three operations generate samples for simulation. The first two methods randomly pick numbers in pre-determined range. While genClass randomly generates system and user weighted range of 1:9. To comply with unicell biological construct, the processes to be created consist of 2 types, namely, voluntary or user process and involuntary or system process. Voluntary process is a controllable process type. It will respond under conscious decision depending on the actors that carrying out a task (in this context human behaviors will be used as a reference) i.e. walking, eating, performing body movements, etc. On the other hand, involuntary is unmanageable type of process that is not under controlled by will. It is automatically committed. The required resources will be spontaneously fed to these processes when the resources are available. Resources that are held when the process finishes using will be released to the resource pool. For this reason, there is no proper parameter value for this weight distribution. The value depends on body rigidity, health, age, and other attributes that are not considered here as they are beyond the scope of this work. In this research, for the sake of simplicity, 1 and 9 are chosen to represent these 2 given classes

```
def genProc(self, freq):
    proclist = []
    for i in range(freq):
        proclist.append(self.genTimeConsumption())
        self.numberOfProc += 1
    return proclist
```

Finally, genProc will initiate the generating sequence with the help of the previously mentioned parameters and freq which specifies the number of sequence occurrences. Once this generating sequence is completed, the list of inputs will be marked and matched to the corresponding pre-generated input processes.

After the input process is set up simulation will commence. This will be described in the next section.

3.2.2 Memory Architecture Design

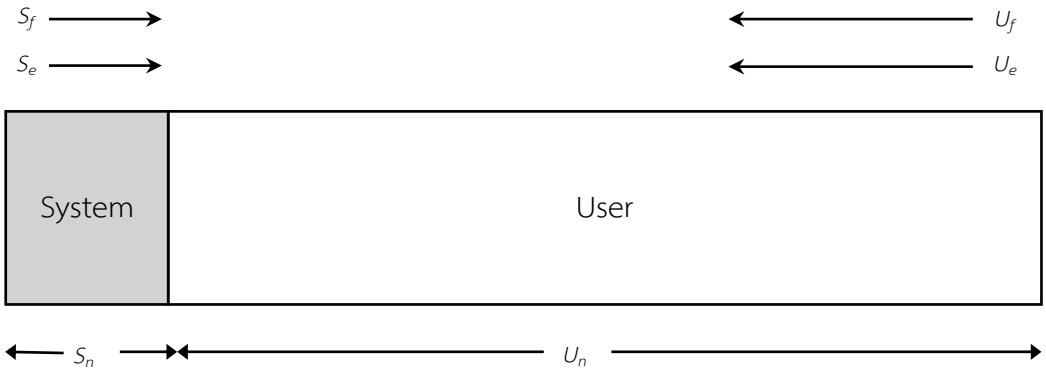


Figure 3.3 Memory Architecture

Consider living the unicell as model that has no clear cut as to how resource pool in their body is managed. The memory will be treated as one uniform block are as illustrated in Figure 3.3. The user process starts from one end while the system process starts from the opposite end.

The 2 types of processes share the above resources pool. Memory allocation is divided as follows: 10% belongs to system process space, and the remaining 90% belongs to user process space. The parameters of this memory pool are defined below:

- U_f indicates next free slot in user reserve spaces.
- U_e indicates process that currently executing in user reserved space.
- U_n indicates total user reserve space.
- S_f indicates next free slot in system reserve spaces.
- S_e indicates process that currently executing in system reserve space.
- S_n indicates total system reserve space.

As mentioned earlier, FIFO scheme serves several benefits. However, FIFO scheme has infamous starvation problem. There are many starvation solutions. In this work, time slice is chosen in the form of TTL. TTL will grant only limited execution time to each process. When TTL expires, the resource will be shifted to the next process until no process left.

At this point, the proposed scheme might look similar to Round-Robin (RR) scheme. However, they are not identical, RR has only escape criterion, which occurs when their time slice runs out. On the other hand, the proposed scheme is mimicking life form behaviors. When human executes their routines, they can either finish or abort it. The same goes for this simulation. When the currently executing process is finished, it will leave. Otherwise, abort will take place when TTL expires. In both scenarios, the next waiting process will run.

Comparing conventional RR with the proposed method, suppose the quantum time is set to 8 clock ticks. Given a job requests 16 clock ticks to finish that task, to finish 16 clock ticks task both RR and the proposed scheme need 7 execution iteration. Assume that during 4 iterations, there is no interruption. However, during execution iteration, not all of 8 ticks are allowed to execution, only 5 ticks are executable, full detail will be described later on.

RR holds entire execution iteration since RR will not release resource at any time except when time quantum runs out. While the proposed scheme utilizes its resource by spend only 3 entire iterations and 4 ticks on last iteration. Since the proposed scheme has 1 additional exit condition which the process can exit immediately after it finished. In this experiment, TTL timer is set to 8 ticks, each tick denotes the followings:

Table 3.1 Tick indication

Tick	Description	Instruction
1	Preload upcoming instruction	Load target process, and assign to closest available space.
2-6	perform task execution	Process that pointed with either S_e or U_e will be executed.
7	perform accumulation	Progress that previously accomplished will be stored in this iteration.
8	perform transformation to next process	Advance pointer by one and reset TTL timer to initial value.

3.3.3 Modified Process States

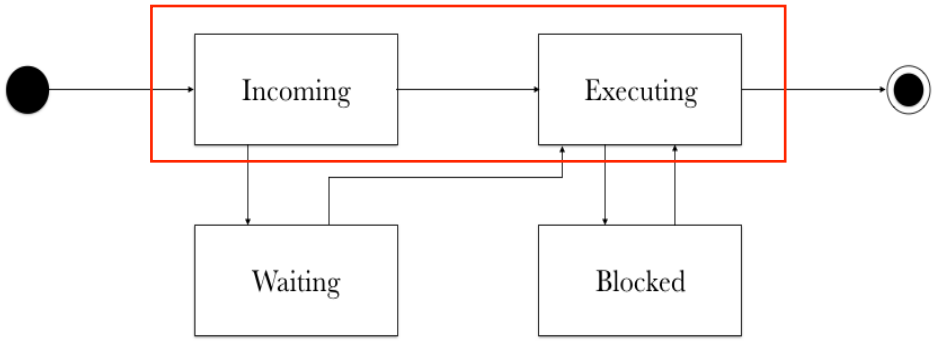


Figure 3.4 Process States

Process execution states are arranged as follows:

- Incoming handles freshly arrived processes yet to assign a certain address. When incoming state is flooded and there are no resources left to be used. The system will shift those processes into Waiting, which are place where those overflow processes are stored. Processes that are stored in Waiting will be transferred to Incoming either an entering execution iteration or resources pool become available.
- Executing manages execution procedure while those processes will be held only certain amount of time.
- Blocked holds processes that are being transferred from executing list as urgent task has arrived.

Notice that process execution states are somewhat different comparing to conventional approaches. Since this work aims to solve the memory wall problem, modification is done to simplify the process states. The first two states, incoming and executing, are designed to be hardware-implementable ready components. However, the rests are more complicated components since additional code are needed to control the actual hardware, making it more complex. The following sections will describe each process state in detail.

Incoming

1. Freshly generated process has arrived at incoming list.
2. Look for a slot in the list.
3. Perform availability checking.
4. Place those processes in available spaces of the executing list.
5. Push into Waiting list if no space is available.

Waiting

1. Place the process in waiting list.
2. Perform availability checking.
3. Push process back to executing list one process at a time if there is available space.
4. Repeat until there is no process left in waiting list.
5. Hold until next iteration. If no space is available, repeat step 2-5 again.

Executing

1. Enter the executing list.
2. Grant a permission to qualified process for execution.
3. Check whether currently executing process is finished.
4. Push that process executing, if it is finished, Mark it and advance pointer by 1.
5. Check for interruption in next iteration. If it is not finished.
6. Push currently executing process on to blocked list if there is an interrupt.
7. Repeat 2-6 again, if otherwise.

Blocked

1. Shift process to Blocked list if urgent task arrives.
2. Perform checking for urgent tasks.
3. Hold current task for next iteration if there is urgent task.
4. Place process back to Executing list otherwise.

3.3.4 Policies

Placement of process in memory pool requires policies to handle every state as follows:

3.3.4.1 Incoming

When process generator launches a number of processes, the process handler will manage those processes by placing them one by one until the reserved spaces are exhausted. The order of placement starts with executing list to be filled first, followed by waiting list.

3.3.4.2 Execution

There are 2 groups of pointers associated with user and system processes. User pointers consist of U_e and U_f denoting user free space and user currently executing, respectively. System pointers also have 2 pointers, S_f and S_e, denoting system free space and currently executing operating system task, respectively. There are 4 sub-policies governing in the execution process:

Figure 3.5 Memory pool layout for execution

1. Both user processes and system processes are not present.

This situation occurs at the very beginning and the end of simulation. When the simulation starts, no user and system processes exist in the resource pool simultaneously. The simulation system will be waiting for new arriving process one at a time as illustrated in Figure 3.5. Similarly, at the end of simulation, all jobs must be completed, leaving no process in the system.

2. there is 1 user process with 1 or more system processes at any given time.

Figure 3.6 1 User and 1 system process

When simulation encounters this circumstance, both system and user pointers will be pointed at the top of both executing lists. The normal execution will start from system side as demonstrated in Figure 3.6.

3. There is no system process but several user processes.

When the system list is empty, execution control is transferred to user process side as demonstrated in Figure 3.7.

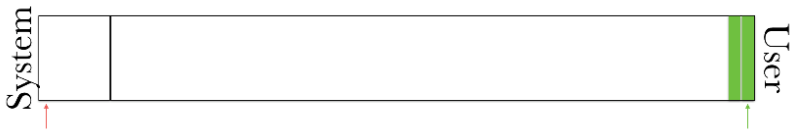


Figure 3.7 Only user processes, but pointed has not yet moved

This deployment helps minimize executing time by reducing context switches as demonstrated in Figure 3.8.

Figure 3.8 Only user processes, pointers move to user space

4. System processes fill up their space and some user processes.

This situation is similar to situation 3, except, there are system processes. In this case, the user processes will be ignored. Execution control is shifted to system side as

demonstrated in Figure 3.9. The time utilization is not a main concern so as to achieve the system objectives. The previously executing user process will be suspended.

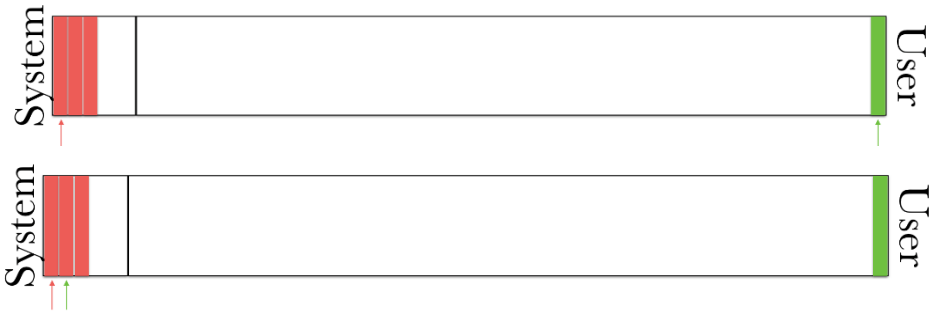


Figure 3.9 Multiple system processes

3.3.4.3 Overflow

At some point, overflow situation might occur.

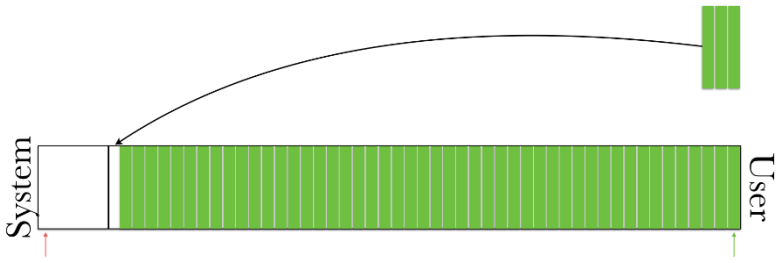


Figure 3.10 User space is flooded

The first scenario occurs when the user reserve space is flooded with user processes. The incoming user processes will be rejected as demonstrated in Figure 3.10. The simulation will not allow to executing partial result that could be erroneous.

Figure 3.11 System space is flooded

On the other hand, when the system side is flooded and there is incoming process waiting to enter as demonstrated in Figure 3.11, simulation will halt and perform restart procedure.

Since this implementation is developed by using living creature as a reference, the policies are adopted from their behaviors. Take the last policy as an example, suppose the system process hold an illness and user processes hold daily routines. Once illness starts human body is capable of sustaining certain amount of illnesses. When the limit is reached, the body will collapse.

Chapter 4 EXPERIMENTAL RESULTS

Since there was no supporting environment that work in the same manner as the proposed scheme, implementing a simulation was viable. This section will explain hardware specifications, software used, as well as techniques used in the experiments.

4.1 Experimental Setup

Experimental set up is described in the following sections.

4.1.1 Hardware Specification

This simulation was developed and simulated on 2 machines. The primary machine handled the simulation and its environment by Intel core I7 4790, 8GB DDR3 DRAM and running on Ubuntu 16.04LTS. The secondary machine concurrently ran the application tasks using Intel core m3, 8 GB DDR3 DRAM on OS X 10.12.6.

4.1.2 Software Used

Several software tools were used in this implementation such as Integrated development environment (IDE), design tools, and programming language tool.

4.1.2.1 Programming language and Tools

Python was chosen to be the programming language. Its simplicity made the development easy to fix, add, or update source code, and less number of lines of code. Consequently, the program was readable and easy to understand.

The support IDE also made the development task easy by means of PyCharm. Its distinctive structural type color coding provided built-in code completion, code hinting, and local version control. In addition to these handy features, many add-on

configuration issues were solved by the suggestive assistance of the development community.

4.1.2.2 Operating Systems

This implementation employed 2 operating systems, namely, OS X and Ubuntu. These two operating systems were used in development phase for three reasons: stability, functionality, and popularity.

- Stability-Ubuntu is a freeware, cross operating system that also support command line tools that makes it easy to manage a number of source files.
- Functionality-the command line tools provide more efficient and effortless to manage the development process. Plenty of version control software are available through command line.
- Popularity-the more users use, the more suggestions and pointers are revealed and solved.

4.2 Input Specification

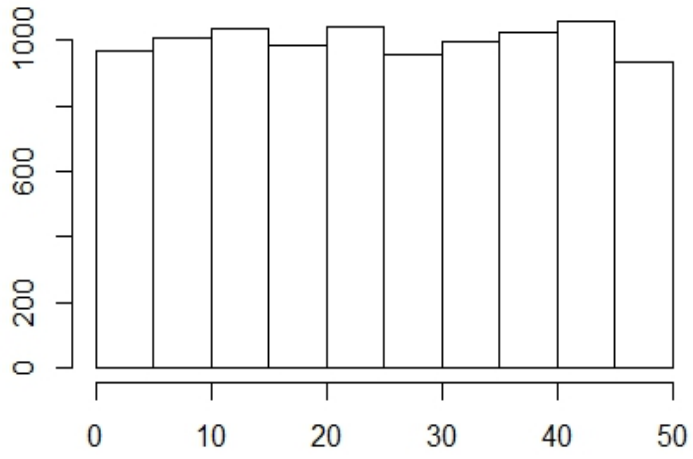


Figure 4.1 10,000 pre-generated inputs

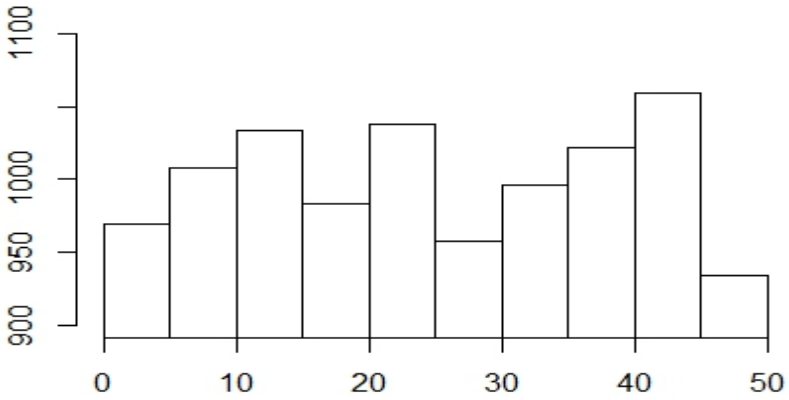


Figure 4.2 10,000 pre-generated inputs (enlarged)

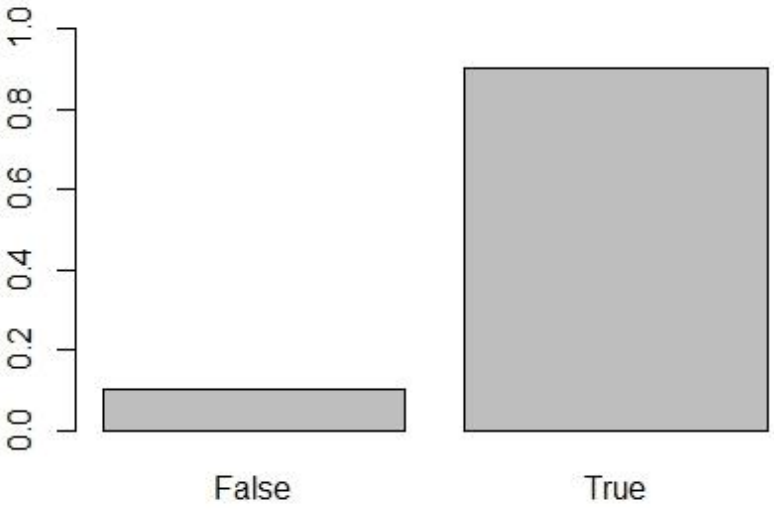


Figure 4.3 Process classes

In this simulation, both user and system processes combined to create 10,000 instances as input references. Each process consisted of 2 values, requested time and class. Each requested time interval was spread relatively equal as shows in Figure 4.1 and Figure 4.2. The mean value of requested time was approximately 25.1642. The minimum and maximum were set at 1 and 50, respectively. User and system processes

were initiated by 8,991 and 1,009, respectively, according to the pre-determined ratio of 9:1.

4.3 Results

This experiment was organized to verify the contributions of this work. Three well-known algorithms were chosen to compare with the proposed scheme, namely, First-in, First-out (FIFO), Shortest remaining time first (SRTF), and Round-Robin (RR).

Mandatory operations were chosen to measure the number of operations used during process execution. Sort was required by SRTF only, as it needed to obtain least remaining time process to execute first. Transfer of control was involved for shifting resource between processes. Since the proposed scheme employed TTL time slice concept, it was similar to RR that needed more resources to perform context switch when the process quantum time ran out. Remove expelled processes from the resources at certain states, from waiting queue to execution, and from execution to finished. This simulation performed 10,000 processes. The results are shown in Table 4.1 and Table 4.2.

Table 4.1 Operation used

	FIFO	SRTF	RR	Proposed Scheme
Sort	0	$\sum_{i=0}^{N} T_i$	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 4.1 shows the number of operations performed during execution process. Notice that only SRTF required sorting operation to looks for the shortest process first in every execution iteration. While the other 2 candidate algorithms and the proposed scheme did not required sorting. This sorting operation costed $\sum_{i=0}^{N} \mathrm{T}_i$ with very optimistic approximation.

FIFO and SRTF transferred execution grant when currently executing process was finished, so they required only total number of process (T_n). On the other hand, RR and the proposed method involved time slicing. Hence, the number of operations used by these two methods were significantly high. However, theses prevented the starvation problem. Hence, these two methods shared the resources evenly.

Remove performed when FIFO and SRTF finished their tasks. The number of removes was equal to the number of incoming inputs. Unlike RR and the proposed scheme, they just simply replaced the outgoing process with a new incoming one.

Table 4.2 Execution time in clock ticks

Approach	Time	Result	Difference
FIFO	$n \times (\bar{x} + 3)$	26815	-31.061%
SRTF	$\mathrm{N\ LOG}\ N + \sum_{i=0}^{N} n \times (\bar{x} + 3)$	74340	+91.120%
RR	$\sum_{i=0}^{N} \lceil p_i \div 5 \rceil \times 8$	41631	+7.0288%
Proposed Method	$\sum_{i=0}^{N} \lfloor p_i \div 5 \rfloor \times 8 + (\boldsymbol{P_i} \div 5) + 3)$	38897	$\pm 0\%$

Table 4.2 reveals time consumption and differences among the chosen methods and the proposed scheme. FIFO took the least time complexity, followed by the proposed method, RR, and SRTF, respectively.

FIFO used only 26815 to finished 10,000 processes, spent only 60 percent of the proposed method's time consumption. SRTF required sorting operation before allocating the designated process to memory which cost extra $\mathrm{N\ LOG}\ N$ assuming merge sort was deployed in this component. This sorting cost extra runtime.

Besides runtime speed, FIFO and SRTF might encounter starvation problem when very long processes arrived in waiting list. RR and the proposed scheme were free from this predicament by virtue of the time slice.

RR was almost on par with proposed method, yet slightly slower. Notice that this simulation executed only 10,000 processes, where RR already gained an extra 10 percent. In practice, the number of processes would be much higher than this.

Finally, the proposed method outperformed SRTF and slightly quicker than RR by gaining from last execution iteration for each process. Traditional RR would release the resources when time quantum ran out, while the proposed method could exit as soon as it finished.

In addition to execution time, average waiting was also measuring in this experiment. The results will be shown in following figures.

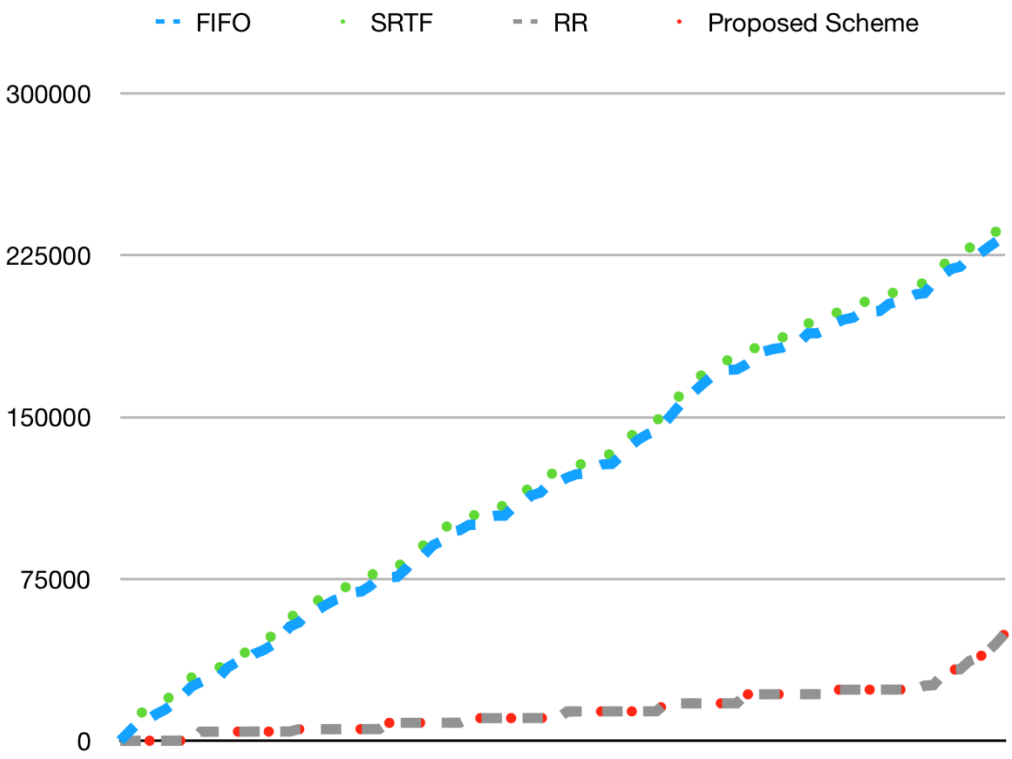


Figure 4.4 Average waiting time

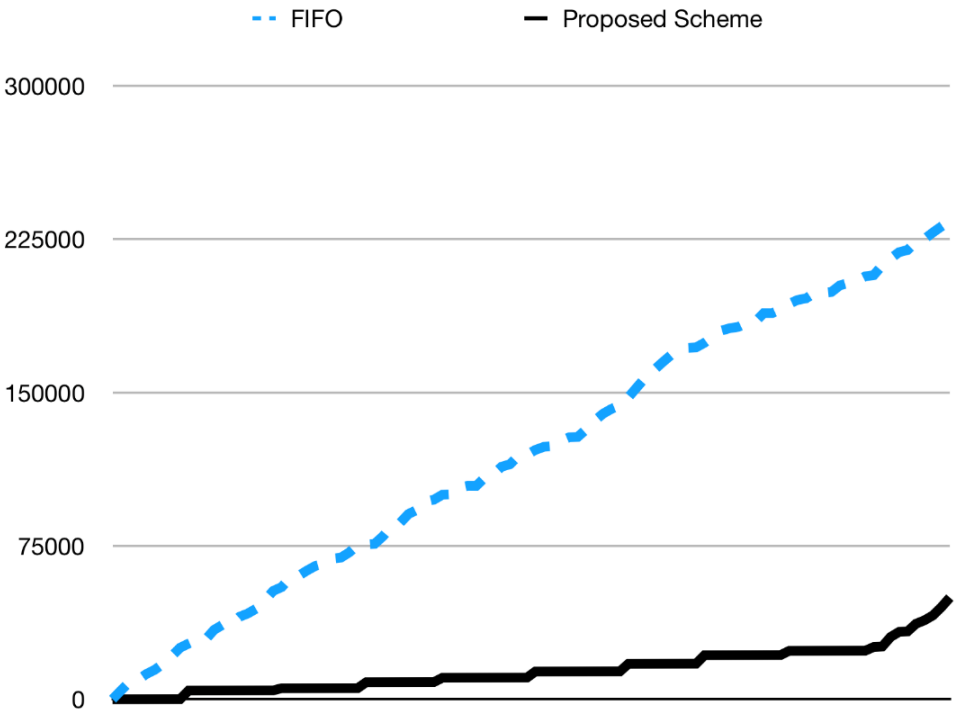


Figure 4.5 Average waiting time between FIFO and the proposed scheme

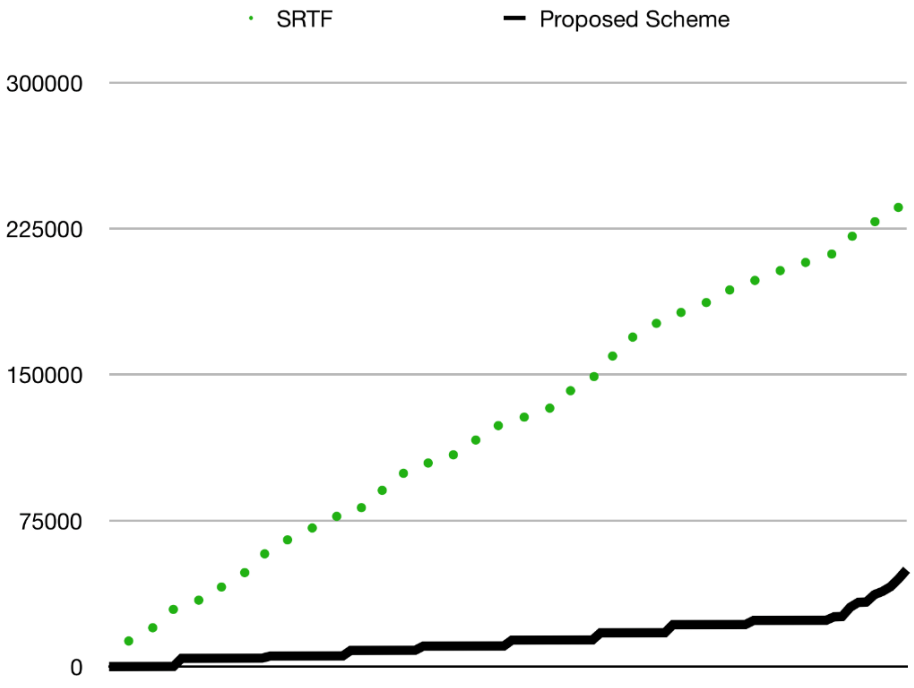


Figure 4.6 Average waiting time between SRTF and the proposed scheme

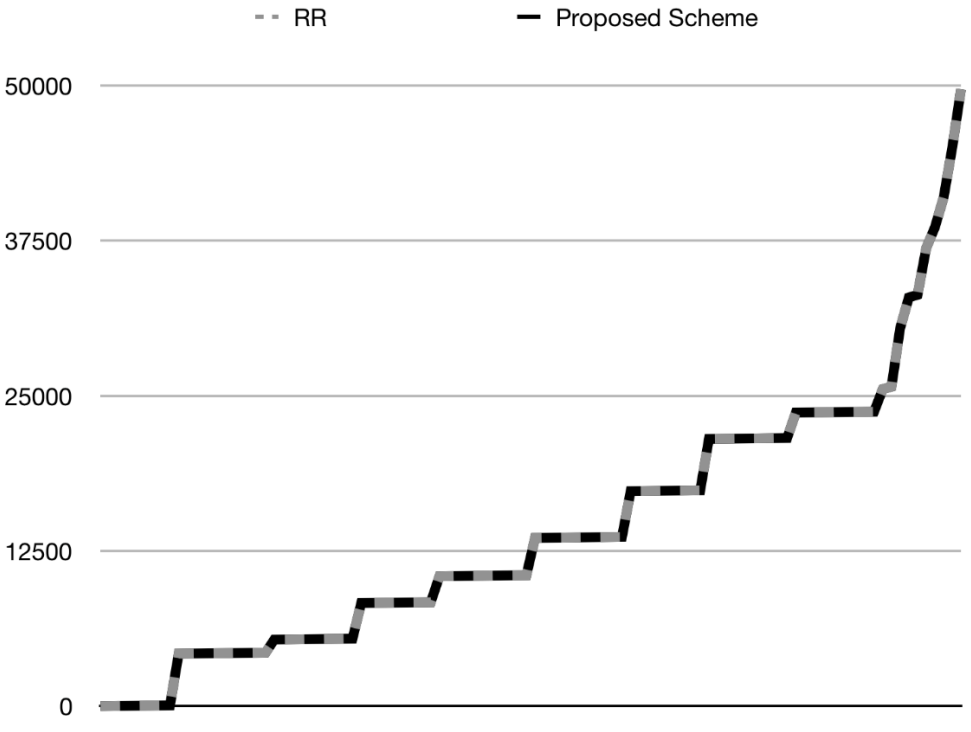


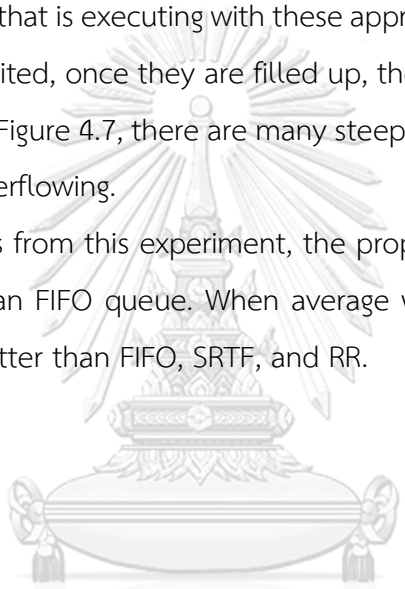
Figure 4.7 Average waiting time between RR and the proposed scheme

These measurement figures are chosen to measure the accessibility of resources. In this measurement also deploy 10,000 processes.

Figure 4.4 shows all of candidate methods compare with the proposed scheme. Results are divided into 2 groups separated by their characteristics. For the first group, FIFO and SRTF, neither of these methods is employing time slice concept which mean entire process will be done in one execution. Obviously, processes in waiting queue have to wait as illustrated in Figure 4.5 and Figure 4.6.

On the other hand, RR and the proposed method are imposed by time slice concept. Every process that is executing with these approaches will be serviced evenly. Since, resources are limited, once they are filled up, the rest of the processes have to wait. As can be seen in Figure 4.7, there are many steep rises in the graph which reflect the memory pool is overflowing.

Consider the results from this experiment, the proposed scheme took more time to finished the jobs than FIFO queue. When average waiting time is considered, the proposed scheme is better than FIFO, SRTF, and RR.



Chapter 5 DISCUSSION

5.1 Discussion

The proposed method ran only single-thread per process type. In case the number of execution threads increases, the running time could be improved. However, with trade-off issues between complexity and simplicity, further investigation still is required.

From the performance standpoint, tightening the gap with FIFO arrangement is important. The rationale why modified FIFO was adopted was because this work was aimed to benefit the simplest existing placement scheme by employing TTL time slicing to overcome starvation problem.

Notice that none of these algorithms uses every candidate operation. SRTF requires sorting to find the smallest processes, while FIFO requires none of these excepts remove operation when currently executing process is finished.

While RR and the proposed method do not have to perform removal, they just simply replace new process to the old one's place. No sorting is needed. In transfer of control, there is difference in time computations. RR rounds up so the ceiling function is used since the process could only be taken out of the memory when time slice is up. The proposed method truncates and add extra value because the process could exit when it is either finished or TTL is expired.

This small refinement could lead to high performance boost. In reality, single program could be decomposed into several processes. Consider Table 4.2 for 10,000 processes, the proposed method gains approximate 7 percent. In a real production environment, the more processes run, the higher gain on processing time.

Chapter 6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, unicellular animal biological structure and life form's behaviors were employed to create a novel memory allocation scheme and to combine with known computer science knowledge. FIFO was chosen to be the basis of allocation scheme. As a consequence, several memory related parameters and operations were deemed unnecessary for memory allocation scheme to achieve simplicity and hardware-implementable ready scheme. Certain policies were precipitated as the by-product. Ultimately, memory placement and new allocation methods could be performed directly. This array of refinements permitted the proposed scheme to operate with minuscule overhead and in reasonable time. The memory wall problem would methodically be mitigated as more memory spaces were made available. This could fulfil necessary functions and remained competitive to other well-established algorithms.

6.2 Future work

Further development will focus on current memory allocation scheduling techniques such as runtime and effectiveness. Runtime efficiency improvement can be handled by multithread processing. Care must be taken on extra operations that complicate the supporting algorithm, not to mentioned parallelism issues. In addition, new evaluation approaches are needed to justify between simplicity and performance, as the performance gain will be the extra operations and runtime efficiency.

Hardware implementation is another challenging point to be explored. Preparation has been made from the outset. A number of unique characteristics have also gathered from unicellular animals to be adapted.

Simplicity swiftly becomes a schematic idea of this research. The process state reduction is mimicking unicellular life form to system structure, which in turn is utilizing

well-developed life form's behaviors to administer process placement and execution policies. Both incoming and executing states as shown in Figure 3.4 are already hardware implementable. The instructions in these two components can be straightforwardly operated but waiting and blocked still need further minimization to run at hardware level.

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

[7] 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.

[8] 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.

[9] 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.

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

APPENDIX

VITA

Name: Mister Gasydech Lergchinnaboot

Country: Bangkok, Thailand

Education: B.Eng. (Software Engineering); King Mongkut's Institute of Technology Ladkrabang, 2014

Affiliation: Advanced Virtual and Intelligent Computing (AVIC) Center, Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University.

Publication: G. Lergchinnaboot and P. Sophatsathit, "A Biological-like Memory Allocation Scheme Using Simulation", 2nd International Conferences on Information Technology, Information Systems and Electrical Engineering (ICITISEE 2017) Yogyakarta, Indonesia 2017: 426-429.