

# Visualizing the Modern Operating System: Simulation Experiments Supporting Enhanced Learning

Besim Mustafa  
Business School  
Edge Hill University  
Ormskirk, UK.  
+44 (0)1695 657640

mustafab@edgehill.ac.uk

## ABSTRACT

An important area of modern computer organization and architecture is the operating system the internals of which is normally inaccessible for teaching and learning purposes. This paper describes an educational operating system simulator that is part of an integrated set of simulators designed to support students of computer architecture and operating systems. Examples of classroom assignments are presented demonstrating the simulator's support for a wide range of practical experiments. The pedagogical value of the simulator is assessed in terms of the educational impact of its visualization features and its functional capabilities for supporting students at different levels of learning. Finally, the preliminary results of the evaluation of the simulator that provide an indication of its value as a teaching and learning resource are presented.

## Categories and Subject Descriptors

K.3.1 [Computers Uses in Education]: Computer-assisted instruction (CAI); K.3.2 [Computers and Information Science Education]: Computer Science Education --- *operating systems*.

## General Terms

Algorithms, Experimentation.

## Keywords

Operating system, visualization, simulation, pedagogy.

## 1. INTRODUCTION

The study of operating systems forms an important and essential part of computer science students' education [3,4] and as a result many degree level courses offer study modules on the internals of operating systems at introductory and at advanced levels.

The author has been responsible for designing and delivering two modules on computer architecture and operating systems at undergraduate degree level for the past seven years. The

introductory computer architecture module is taught during the first year and includes an introduction to operating systems. The module on operating systems is taught in the second year and includes lectures on advanced features of operating systems. In order to support the practical lab sessions an integrated system simulator [7] that includes a teaching compiler, a CPU simulator and an operating system (OS) simulator has been implemented. A unique feature resulting from this integration is the system simulator's ability to clearly demonstrate interdependencies and levels of support between these three areas in ways that no other simulator provides. The author has been able to successfully integrate the simulators into several of his teaching modules and they have been supporting students for the past five years.

## 2. PREVIOUS WORK

The traditional teaching of operating systems has been following one or a combination of the following three main methods: a) students modify or extend parts of an operating system [2], b) students write code to demonstrate aspects of technology on a commercial operating system [11,12], c) students run code simulating parts of operating system technology [5,6,9,10]. Both (a) and (b) often require moderate to substantial knowledge and experience in using development environments and in writing code in languages such as C and Java. The simulators in (c) are often isolated individual pieces of code from various sources each with different look and feel and aimed at different target audiences. The author took the decision that in order to maximize the pedagogical benefits to his students and to better support his classes a new integrated set of simulators was justifiable. This way not all the students would be required to possess good systems programming skills and the simulations of different aspects of OS and computer architecture would be seamlessly integrated providing a common look and feel.

## 3. THE SYSTEM SIMULATOR

Although this paper is primarily about the operating system simulator, it will be helpful to briefly describe the rest of the system in order to clarify the context in which the OS simulations have been implemented and utilized.

The system simulator integrates three important areas of computer architecture in one educational software package: generation of CPU instructions using a high-level language compiler and an assembler; the CPU as the instruction processor; the operating system as the facilitator of multiprogramming and multi-threading of the CPU instructions.

### 3.1 THE CPU SIMULATOR

The CPU simulator simulates the hardware functionality of a fictitious, but highly realistic, CPU based on RISC type architecture. It incorporates a five-stage pipeline simulator and both data and instruction cache simulators. The CPU simulator can execute instructions either generated by the integrated compiler from high-level source code or manually entered by the students. Multiple CPU simulations are supported and can be used to simulate parallel processors.

### 3.2 THE OS SIMULATOR

The OS simulator is designed to support two main aspects of a computer system's resource management: process management and memory management. The CPU code is visible to the OS simulator which is able to create multiple instances of the code as separate processes. The process scheduler includes support for scheduling policies including priority-based, pre-emptive and round-robin scheduling with selectable time slots. Virtual resources can be allocated and de-allocated to processes allowing demonstration of deadlocks associated with resources and investigation of deadlock prevention, detection and resolution techniques. Threads are supported via special teaching language constructs which allow parts of program code to be executed as threads and process synchronization concepts to be explored.

### 3.3 THE TEACHING COMPILER

A *teaching* compiler is developed for a basic but complete high-level teaching language in order to support the CPU and the OS simulations. This language incorporates standard language control structures, constructs and system calls which are used to demonstrate a modern computer system's key architectural features. The compiler can generate assembly-level code as well as binary byte-code as output and includes a code optimizer.

## 4. TEACHING - LEARNING STRATEGY

Each one-hour lecture is supported by a two-hour practical tutorial session. The students work in groups of two or three. The simulator software is provided on a removable disk drive and runs under Windows operating system. The groups follow instructions on the exercise sheets and are directed to interact with different stages of simulations. The work completed during the practical sessions forms the student's tutorial portfolio which is assessed at the completion of the module.

### 4.1 EXAMPLES OF USAGE

In order to better demonstrate the pedagogical capabilities of the OS simulator, this section is devoted to describing a number of practical assignments that the students have been asked to undertake during the practical tutorial sessions.

Assignment 1: Process scheduling, task switching, PCB data

Using the built-in compiler's editor the students write a simple tight loop as in Figure 1. They then compile it and load the code in the CPU simulator's memory. Using the OS simulator the students manually create several instances of this code as processes. They select a scheduling mechanism from the list and activate the OS simulator's scheduler and make a note of its behavior from their observations. They repeat this for each of the available mechanisms. They are then directed to create new processes with different lifetimes assigned to them. They repeat the above but in each case the same processes are queued in a different order. They make a note of the average waiting times

from the information contained in the OS simulator's event log. Next, pre-emptive and priority scheduling are observed by giving priorities to processes. This is facilitated by simulator's ability to suspend itself when a new process is scheduled to run.

Next, task switching is investigated. The students select round-robin scheduling and create two processes. When one of the processes times out and another is scheduled the OS simulator suspends itself. The students access the PCB of the timed-out process and make a note of the PC value and the saved values of the CPU and the SR registers. When next time round this process is scheduled to run the students get the opportunity to observe the new value of the CPU simulator's PC register, its SR and register file values. They can then repeat this for the other process.

```
program LoopTest
  var g integer

  g = 0
  for n = 1 to 50
    g = g + 1
  next
end % LoopTest
```

Figure 1. Simple loop source code.

Assignment 2: Paging, page table, address translation

The CPU simulator has a maximum free primary memory space equivalent to ten 256 byte pages. When a new process is manually created it can be assigned memory of up to maximum ten pages. The pages of those processes that cannot fit in the primary memory are notionally stored in a secondary memory. The students manually create processes with varying page sizes and make sure that some processes' memory pages are swapped out. This way when processes are scheduled to run their memory pages are swapped in at the expense of some other processes' pages. This way relocation via paging is invoked and observed by the students. This will in turn force the data in a process's page table to change. The students can access the contents of each process's page table and its memory pages. As the data in a process's virtual memory is updated the address translation ensures that the correct memory location is accessed in the primary memory. As pages are swapped in, the page fault count is also incremented. The students have the option of manually invalidating a page table entry thus forcing a page fault when the memory data is next updated. All these activities are visually observable as they occur. The students are asked to comment on the virtual memory activities they observe and are invited to work out the physical addresses from the virtual addresses using a process's page table. They can then verify this by running the simulator.

Assignment 3: Investigating threads

The OS simulator is able to create and schedule threads. The demonstration of this requires the in-built compiler generating the correct system call via soft interrupt code by using special language constructs, the CPU simulator executing it and the OS simulator servicing the soft interrupt. In fact, it is precisely this feature of the integrated simulator that makes it unique and unlike other OS simulators. The student enters or is given the source

code, shown in Figure 2, to compile and load in the CPU simulator's memory.

The students create two instances of the same code as processes. They then activate the scheduler and observe the creation of the threads. The OS simulator provides a process tree view from which the student can observe the parent/child hierarchy as they are formed. Next, they can confirm from the messages displayed on the simulated console that the threads have access to parent's global memory space. The question of what happens to children when parent dies is explored by selecting an option which either kills the children or orphans them when their parents exit. Orphaned children become directly attached to their grandparents or to the root process and this is clearly observable in the tree view.

Finally, the role of the "wait" statement is explored. The students observe the effect of the displayed messages and conclude that the parent process will be suspended when the wait statement is executed until all its child processes terminate. The spawning of the threads and the "wait" construct are implemented via a software interrupt instruction (SWI) that specifies the requested OS action. The students can observe this mechanism from the generated assembly code and can follow its progress in the CPU simulator. This is yet another example of the cooperating interfaces clearly illustrated by the simulator.

#### Assignment 4: Investigating synchronization, critical regions

The built-in teaching language includes constructs that facilitate implementation of thread synchronization as well as critical regions of blocks of code. The thread synchronization is implemented by OS and the special CPU hardware instruction that can atomically test and set a flag. The critical regions are implemented by the OS via the soft interrupt instruction (SWI) that is handled by the OS. In both cases the OS suspends the thread that tries to access the protected region until this region is released by the holding thread.

The students enter source code that uses the "synchronise" keyword to protect two subroutines invoked as threads. This code is then compiled and loaded in CPU simulator's memory. Both subroutines access a global variable, initialize it and increment it in loops. When loops are exited, the value of the variable is displayed in the simulated console.

Figure 3 shows the code used for this investigation. Running this code should display two values of g as 12 and 20 as expected. The students then edit the code and remove the two "synchronise" keywords and run the code again. They now need to explain why the new values observed are different than before and why in this case the task switching frequency (configurable) affects the results.

Next, the students further edit the two subroutines and implement the critical regions of code in both. Figure 4 shows the changes (only one of the subroutines is shown). The two keywords "enter" and "leave" are used to protect the enclosed block of code. Both keywords instruct the compiler to generate the soft interrupt instruction used to enter the OS handlers. As it is the OS that schedules the threads, this method guarantees protection since once the interrupt is served the OS has already flagged the region before the next thread is allowed to continue. The students observe that the results of the modified run yield the correct values for the global variable g.

#### Assignment 5: Investigating deadlocks

The OS simulator can simulate process deadlocks when the right conditions for deadlocking are configured. This enables the students to investigate the conditions necessary for deadlocks and the methods used to detect and resolve deadlocks. The simulator uses two methods: one method requires the use of special constructs in the source code; the other is the manual method. Here the manual method is described. Figure 5 shows the deadlock simulator's interface.

```
program ThreadTest
  var t1 integer
  var t2 integer

  sub Thread1 as thread
    writeln("This is Thread", t1)
    while true wend
  end sub

  sub Thread2 as thread
    call Thread1
    writeln("This is Thread", t2)
    while true wend
  end sub

  t1 = 1
  t2 = 2
  writeln("Start of program")
  call Thread2
  wait
  write("End of program")
end      %ThreadTest
```

Figure 2. Source code for thread creation.

```
program SyncTest
  var g integer

  sub Thread1 as thread synchronise
    g = 0
    for n = 1 to 20
      g = g + 1
    next
    writeln("Thread1 g = ", g)
  end sub

  sub Thread2 as thread synchronise
    g = 0
    for n = 1 to 12
      g = g + 1
    next
    writeln("Thread2 g = ", g)
  end sub

  call Thread1
  call Thread2
  wait
end      % SyncTest
```

Figure 3. Source code for synchronized threads.

```

sub Thread1 as thread
enter
  g = 0
  for n = 1 to 20
    g = g + 1
  next
  writeln("thread1 g = ", g)
leave
end sub

```

Figure 4. Source code for critical region protection.

The deadlock simulator interface presents six different resources available to all the processes. Each resource has only a single instance so once allocated to one process it cannot be allocated to another process at the same time. However it can be requested by one or more processes at the same time. The resources are allocated manually. Each resource is colour-coded reflecting its status. A drop-down list of all processes requesting an allocated resource is displayed. An allocated resource can be manually released. It is also possible to configure various deadlock prevention methods, deadlock recovery methods and detection frequencies. All these are available to enable the students to explore and illustrate the different aspects of deadlocks.

Prior to the students using the deadlock simulator they are given a description of several processes and the resources allocated/requested by each process. They are then asked to produce the resource allocation graph and determine if a deadlock cycle exists. If not, then they are asked to work out what needs doing to create a deadlock. Then they are asked to use the deadlock simulator to verify their solutions. The OS simulator clearly highlights any deadlocked processes and these are suspended until the deadlock is resolved. Next the students are asked to apply a method, e.g. terminating a deadlocked process or pre-empting the release of a resource, in order to resolve the deadlock. They are then asked to configure each one of the three prevention methods in order to assess and demonstrate their relative effectiveness.

Assignment 6: Investigating IO interrupts

The integrated simulator includes a console simulator. The students are asked to enter and compile a source code that can detect a console input event using two methods: vectored interrupt and polled interrupt. The vectored interrupt method requires a special language construct “intr 1” in order to identify a console IO interrupt handler (i.e. interrupt 1). When this code is loaded its starting address is entered in the interrupt vector location 1 corresponding to console IO interrupt vector location. Figure 6 shows a sample source using this method. The students observe that the address of the console input interrupt routine is contained in the list of interrupt vectors shown in a separate window. They can then experiment by manually altering this address and observing its effects.

The code in Figure 6 includes a tight loop in the main body of the program. On pressing a key the CPU executes the interrupt code *IntHandler* which displays the key value. The program terminates if the return key is entered. The students are then asked to modify this code so that the polled interrupt method is used to do the same. They do this by modifying the while loop which continuously monitors the pressing of a key.

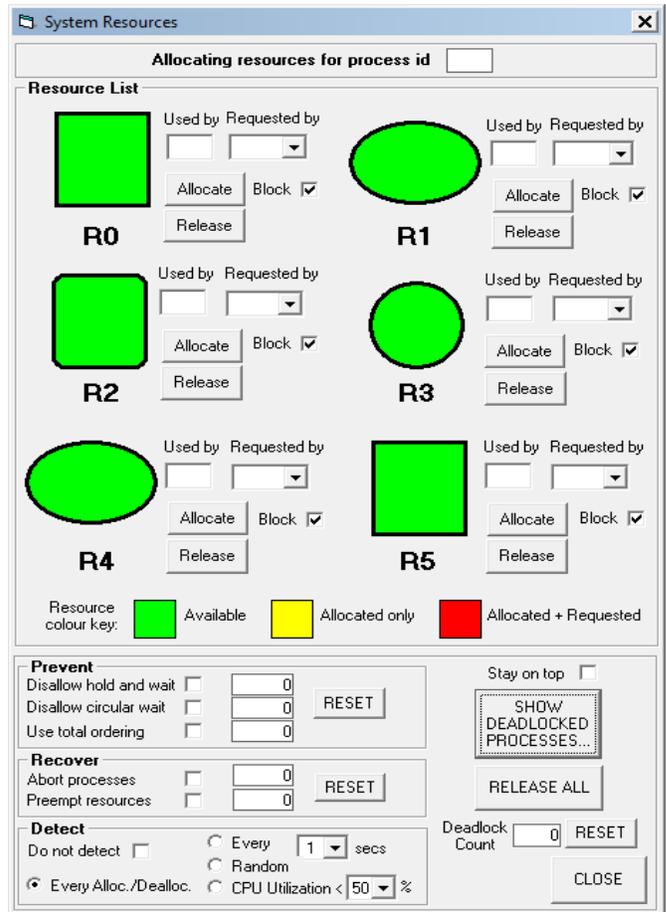


Figure 5. The deadlock simulator window.

```

Program IOInterruptTest
var k integer

sub IntHandler intr 1
  read(nowait, k)
  writeln("Key pressed:", k)
end sub

while k <> 13
  wend
end % IOInterruptTest

```

Figure 6. Source code for IO interrupt handler.

The next two assignments are advanced and are usually suggested as optional assignments as a challenge to more able students.

Assignment 7: Multiple CPUs, load balancing, CPU affinity

The integrated simulator can simulate up to maximum four CPUs. These can be tightly-coupled CPUs, loosely-coupled CPUs or a combination of the two. The tightly-coupled CPUs have duplicate code as they need to simulate the availability of global memory. The students then create multiple processes and start the OS simulator’s scheduler. They observe how different processes are assigned to the CPUs. The graphical representation of CPU

utilization is used to indicate each CPU's utilization level. A well balanced system will show similar utilization levels. The students then manually kill a CPU's processes to observe the loss and the subsequent re-establishment of load balancing. The CPU affinity option can then be used to force processes to stick to the same CPU every time they are re-scheduled. The students are asked to discuss and comment on their observations.

**Assignment 8:** Dynamically-linked vs. statically-linked libraries

The integrated teaching compiler is able to create library code using special language constructs and statically linking them with the main program code. Optionally, the library can be dynamically linked. The students carry out both methods of linking library code in order to illustrate the advantages and disadvantages of each method. The method relevant to OS is the dynamically linked one. The students are asked to write a basic math library code that exports its functions. The main code makes calls to these. The students observe that multiple instances of the calling code share the same library code that is loaded in CPU simulator's memory the first time a function is called and remains in memory as long as instances of the calling program are running. Optionally the OS simulator can be configured to unload the library code if all calling programs terminate. All these activities are graphically observable in different views of the simulator once again clearly demonstrating the interplay between different system interfaces.

**5. AN ASSESSMENT OF PEDAGOGY**

An educational resource is of little value if it does not address the educational needs of the students. It is reasonably well understood that a student's learning experience is greatly enhanced by both the increasing levels of engagement and the depth of learning afforded by a teaching and learning environment. The OS simulator is assessed and evaluated on two fronts: 1) Student engagement and learning support, 2) Qualitative evaluation.

**5.1 THE TWO KEY TAXONOMIES**

The assessment relies on two yardsticks: The proposed Engagement Taxonomy [8] which defines six categories of learner engagement with visualization technology and the Bloom's Taxonomy [1] that identifies a learner's hierarchical depth of understanding. In order to establish the degree of suitability of the OS simulator as an effective educational tool it is assessed against the above two yardsticks. One way of doing this is to map the capabilities and the functionality of the simulator against five of the six categories of the Engagement Taxonomy (the no-viewing category is not considered here) and also to map the kind of learning tasks the simulator is able to support against the levels of Bloom's taxonomy. This assessment should provide some degree of confidence in the tool but by itself may not be sufficient.

Table 1 shows the mapping of the example assignments described in the previous section onto the five categories of the Engagement Taxonomy. Table 2 shows the mapping of the same example assignments onto the five learning levels of Bloom's Taxonomy. As can be seen the assignments included in this paper demonstrate that the OS simulator is able to fully cover all the engagement levels of the Engagement taxonomy and at the same time offer capabilities that can sufficiently support the different learning levels of Bloom's Taxonomy.

**Table 1. Mapping onto the Engagement taxonomy.**

		Assignments							
		1	2	3	4	5	6	7	8
Engagement	View	√	√	√	√	√	√	√	√
	Respond	√	√			√		√	
	Change	√	√		√	√	√	√	√
	Construct			√		√	√		√
	Present					√		√	

**Table 2. Mapping onto Bloom's taxonomy.**

		Assignments							
		1	2	3	4	5	6	7	8
Learning	Knowledge	√	√		√		√		
	Comprehension	√	√	√	√	√	√	√	
	Application			√		√			√
	Analysis	√	√					√	
	Synthesis			√		√			√
	Evaluation	√				√	√	√	√

**5.2 THE EVALUATION METHOD**

In order to establish the degree of effectiveness of the OS simulations a preliminary evaluation using two qualitative surveys were conducted. The surveys used 5-point Likert scale. The evaluations were based on a relatively small sample of students involving 37 first-year and 14 second-year honours undergraduates studying for the computing degrees. Table 3 shows the results of the surveys. The A and B columns represent the aggregated percentages of Strongly Agree and Agree responses and the aggregated Strongly Disagree and Disagree responses of the first-year students. The C and D columns represent the same aggregated responses for the second-year students. The responses are positive and indicate that in the opinion of the majority of the students surveyed the simulations were useful to and supportive of their understanding of the operating systems technology. This survey yielded a high Pearson's correlation coefficient of above 0.78. Interestingly the second-year students appear to be happier in their responses possibly attributable to their familiarity with the software.

Students are required to maintain individual portfolios of their tutorial exercises. The portfolios include student reflections on their learning experiences. Below is a small sample of extracts from student reflections on the exercises using the OS simulations (text unedited):

- “Being able to see scheduling actually working on the simulator made it a lot easier to understand”
- “The lecture confused me slightly about threads however once I had used the simulator I understood it more”
- “Today's session was interesting, it was a change actually opening a simulator and physically loading processes into the operating system”
- “The simulator made the theory much more understandable”
- “The knowledge gained from the lecture and the help from the simulator made this session comfortable and easy to complete”

“Today's session has helped me to develop a greater understanding of process scheduling and this was due to the simulator”

“Today's session was particularly enjoyable as it allowed for group work and communication as well as seeing how the process scheduling works within an operating system by using the simulator”

“I feel that I have learnt the idea of threads much easier by actually seeing and understanding the ways in which they work as well as being able to discuss these findings with group members”

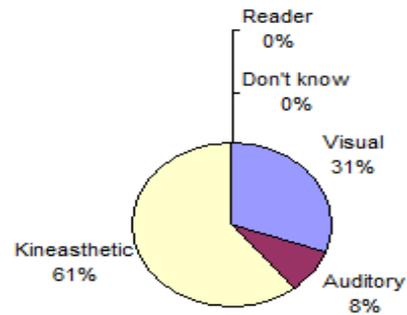
**Table 3. Opinion survey results.**

Survey Questions	A	B	C	D
Overall, the simulator has been a useful tool in understanding some of the more difficult computer architecture concepts	94.6	0	100	0
I found the simulator too complicated to understand and use effectively in most simulator-based tutorials	24.3	56.8	7.1	78.6
The simulator has been more effective in helping me understand the difficult concepts than reading the textbooks or searching on the Internet	75.7	0	100	0
I was encouraged and enjoyed exploring different aspects of computer architecture using the simulator	81.1	8.1	92.9	0
I spent more time learning how to use the simulator than actually doing the tutorial exercises using it	40.5	40.5	7.1	78.6
The simulator encouraged the members of my group to work together in order to solve the tutorial problems	75.7	13.5	92.9	0
I found the simulator exercises supportive of and appropriate for the topics covered in the lectures	94.6	0	100	0
I know of or used other software or method more suitable than the simulator for the understanding of the subjects covered in the lectures	24.3	64.9	0	71.4
The tutorial exercises using the simulator were set at the appropriate level of difficulty and challenged me	86.5	5.4	92.9	7.1

We also surveyed the participants about their learning styles. The results are presented in Figure 7 and show that 92% of the participants regarded themselves as visual and kinesthetic (i.e. learning by doing) type of learners. This is consistent with the results of the qualitative surveys and the student reflections.

## 6. CONCLUSIONS

This paper presented an educational tool for supporting the undergraduate lectures and the practical tutorial sessions in both the introductory and the advanced operating systems teaching modules. The full mapping of the simulator's functionality onto all categories of the engagement taxonomy and its support for different learning levels suggest that the tool has the makings of an effective educational resource. The positive opinion survey results and student reflections add some weight to this suggestion. However, more work needs to be done in this area in order to both confirm and consolidate the findings of the preliminary study. Currently, quantitative evaluations are being carried out.



**Figure 7. Survey of learning styles.**

The integrated system simulator and example tutorials are freely available from the following dedicated link: [www.teach-sim.com](http://www.teach-sim.com).

## 7. REFERENCES

- [1] Bloom, B. S., Krathwohl, D. R. 1956. Taxonomy of Educational Objectives; the Classification of Educational Goals, Handbook I: Cognitive Domain. Addison-Wesley.
- [2] Charles L. A., Nguyen, M. 2005. A Survey of Contemporary Instructional Operating Systems for Use in Undergraduate Courses. JCS21 (1), pp. 183-190.
- [3] Computing Cirricula 2001. 2001. Computing Science, Final Report, December 15 2001. ACM and IEEE Computer Society joint report, USA.
- [4] Computing 2007. 2007. Subject Benchmark Statement. The Quality Assurance Agency for Higher Education, UK, 2007.
- [5] Garrido, J. M. and Schlesinger, R. 2008. Principles of Modern Operating Systems. Jones and Bartlett.
- [6] Maia, L.P. and Pacheco, A.C. 2003. A Simulator Supporting Lectures on Operating Systems. 33rd ASEE/IEEE Frontiers in education Conference, November 5-8, 2003, Boulder, CO.
- [7] Mustafa, B. 2009. YASS: A System Simulator for Operating System and Computer Architecture Teaching and Learning. FISER'09 Conference, Famagusta, Cyprus, Mar 22-24.
- [8] Naps, T.L., et. al. 2002. Exploring the Role of Visualization and Engagement in Computer Science Education. ACM SIGCSE Bulletin 35 (2), June 2003.
- [9] Robbins, S. 2001. Starving philosophers: Experimentation with monitor synchronization. SIGCSE 2001 2/01 Charlotte, NC, USA.
- [10] Robbins, S. 2005. An address translation simulator. SIGCSE'05 February 23-27, 2005, St. Louis, Missouri, USA.
- [11] Silberschatz, A., Galvin, P.B., Gagne, G. 2010. Operating System Concepts. 8th Edition. Wiley.
- [12] Stallings, W. 2009. Operating Systems: Internals and Design Principles. Sixth edition. Pearson Education.