MOSS based Methodology to Building an Operational System for MIPS Simulator

Silvio Roberto Fernandes
orcid.org/0000-0001-7300-8423
silvio@ufersa.edu.br

Leiva Casemiro Oliveira
orcid.org/0000-0001-9147-4604
leiva.casemiro@ufersa.edu.br

Antonio V. T. Costa
antonio.costa@ufersa.edu.br

Departamento de Computação
Universidade Federal Rural do Semi Árido – UFERSA
Mossoró/RN, Brasil

Abstract— **The teaching of Operating Systems (OS) has the challenge of joining theory and practice, at the same time keeping students stimulated. But how to do this without drowning students in millions of lines of code from the real OS or abstract too much just using animations without giving them the experience to create their algorithms and seeing them in action? In this paper, we present the MOSS methodology, in which students develop their OS completely integrated with the traditional MARS simulator. We have been applying this methodology since 2015, and the impact on the grades and evasion rate has been very satisfactory.**

Keywords—MOSS, MIPS, MARS, Operating System, teaching methodology

INTRODUCTION

The Operating System (OS) courses are the majority present at all kinds of Computer Science undergraduate schools. The content of these courses presenting varieties of algorithms strategies in favor of the hardware and software interchanges. The main functions of the OS include resources management especially to access the processor, memory, I/O devices, and file system. The management solves in a fairly way the contest for physical and logical resources shared by the programs, which are called process in the OS context. For such functions, optimized solutions must be implements in the OS in terms of hardware point of view, as well as provide abstract interfaces for both programmers and final users [1].

In this scenario, the challenges of teaching OS involve the theoretical concepts detailing practice tasks, to allow experiences as close as possible of the real situations, with efforts in a short time course. Furthermore, the typical solutions of an OS are very specific[2]–[4], and build for one or a small set of processors, which skills and acknowledgment of the architecture and assembly language expertise. Thus, an interdisciplinary methodology between Computer Architecture and Operating System courses, which uses standardized solutions for a processor, can benefit the learning and reduce the gap inter theoretical and practical teaching.

This work presents an implementation of instructional MIPS Operating System Simulated (MOSS), built through an interdisciplinary methodology employed at OS courses in UFERSA, for a MIPS processor simulation environment. The open-source solution is extensible and enables the construction of distinct educational activities.

RELATED WORKS

One of the most referenced authors in the Operating Systems (OS) area is Andrews S. Tanenbaum. In his books are presents the concepts and the implementation of OS MINIX [1]–[4], and argues that to understand the concepts in practice, a computer student needs to "dissect" the code of an OS like biology students need to do with a frog. The MINIX evolved, but its textbook failed to follow, being currently desynchronized, making difficult a teaching methodology that intends to use both. In [5] is proposed a virtual lab environment that uses a hybrid MINIX and Linux infrastructure to teach security-related topics. The MiniOS [6] is another attempt to use a real OS, during a course, introducing modified activities to implement in a virtual kernel together with a Linux kernel.

In [7] is presented a customizable framework for OS teaching at different levels, which combines theoretical and practical aspects. The authors proposed a methodology presented in two books (one experimental and one theoretical). However, the paper does not present details of the framework but focus on the books generated from it. The work report in [8] aims to improve learning OS concepts using a gamification methodology based on “crosswords” and the “Q & A” game called “Jeopardy!”. They also propose “battle of threads”, inspired by the naval battle game and the “game of process state transition”. Using the problem-based learning methodology, [9] reports the procedures for teaching process scheduling and the deadlock problem. The work [10] implements a virtual core of Linux that can be modified, debugged and restarted without affecting the OS native installation in the computer or other users, and propose its educational uses, emphasizing on distance education. A teaching methodology using Navix OS is present in [11]. Navix is a complete Unix-based operating system, built from scratch, and is simple enough to be easily modified by students. The authors also report on extensive OS documentation.

The use of simulators is particularly interesting in the OS courses, to reduce abstraction of the concepts. The SOSim [12], [13] addresses the main management functions of an operating system in a visual and simplified way. In [14] is presented the S2O dedicated to simulating process scheduling. A Java-based simulator for some aspects of scheduling and memory management is present in [15]. Among the more traditional and more accurate simulators for OS functions, such as RCOS [16], [17], it is noteworthy the Nachos [18], which is a functional OS framework with embedded architecture simulator for MIPS, and the Kaya OS [19], a multi-functional OS project accompanied with a MIPS R3000 processor emulator.

TABLE 1 presents the main aspects of the works most similar to proposed in this paper, MOSS. The features we consider are: if the solution is a real or simulated OS; whether the students can visualize the internal OS information; the presence of didactic material; whether and how it is programmable; and if allowed for students to extend the existing functionalities. Except for S2O and RCOS, all the works present teaching material, but the MINIX material (book) is outdated concerning the last version of the OS.

TABLE 1 Features of instructional OS proposals

Name	***Conception / Architecture***	***Internal Status Display***	***Programmable / Language***	***Extensible / Language***
MINIX	Real OS / x86	No	Yes / C	Yes / C
Nanvix	Real OS / x86	No	Yes / C	Yes / C
SOsim	Simulated / N/A	Yes, GUI	No	No
S2O	Simulated / N/A	Yes, GUI	No	No
RCOS	Simulated / P-Machine	Yes, GUI	No	Yes / Java
Nachos	Simulated / MIPS	No	Yes / C/C++	Yes / C
MOSS	Simulated / MIPS	Yes, GUI	Yes / Assembly	Yes / Java

We highlight that MOSS, despite it is simulated, presents the internal state of all managers through a graphical user interface (GUI), which makes understanding easier and quicker. It provides didactic material, also allows students to write their processes using MIPS assembly, their internal structure can be easily modified or extended using the Java language and coupled to traditional simulator MARS.

MOSS: Methodology And Building Aspects

The proposal educational or instructional OS is a result of the teaching methodology employed at the Computer Science undergraduate program in UFERSA (Universidade Federal Rural do Semi Árido). It consists of practical activities development, focused on MIPS processor at Organization and Computer Architecture (OAC) and Operating System (OS) courses. The methodology efforts strongly integrate both courses since 2014 [20].

The methodology can be applied in the OS course once the background knowledge required in terms of architecture, organization, and in the OAC course, the assembly language of the MIPS processor are intensely studied, which are prerequisite of OS course. In the beginning, students learning about the MARS environment [21]. During the simulation, MARS presents, in color and animated GUI, the memory information updates, regards address, data, and content; the access to registers and floating-point coprocessor; and system call simulations (syscall) to emulated services and tasks during execution time. All of that makes MARS an appropriated platform for teaching OS concepts.

The possibility to create and connect new toolset (*tools*), new syscalls, new instructions, and pseudo instructions, using abstract classes in Java [22], enables instructors to meet better their course schedule due to OS practical activities with MARS and break the gap of practical examples offers at the courses.

A. Steps of Learning

At UFERSA undergraduate programs, the OS courses are divided into three evaluating units, and each one has a score. The methodology proposes three implementation tasks, leading to MOSS building in an incremental process. The tasks orders are:

- **Unit I:** process and threads, inter-process communication (IPC), and IPC classical problems.
- **Unit II:** the study of algorithms for processes Scheduler and memory management concepts.
- **Unit III:** input/output devices management and study regard file system.

Following the methodology guidelines, at the end of a unit, the student will have developed one manager of the MOSS. For this, Java classes will serve as internal structures for the didactical OS and will communicate with MARS. Unpack MARS, source code becomes accessible, and the abstract class *mars.tools.AbstractMarsToolAndApplication* can be extended, and the interface "*mars.tools.MarsTool*" [23] included at the students' implementation files, allowing them to create their functionalities and *tools*. A *tool* on MARS is a graphical interface element that exhibits information about the execution of the simulator. The manager created by the student can access information generated by the simulator, due to the execution of specific assembly code, for example, and show the environment states in simulation time.

In addition to the creation of tools, the methodology guides the students to think and creates news *syscalls*. The *syscalls* work as interfaces between the assembly codes and the *tools*. Extend the class *AbstractSyscall* present in the MARS package, and the students can create their system calls.

Passed a few semesters, the initial version of the MIPS Operating System Simulator, so-called MOSS 0.1 [24], was developed based on the methodology guidelines. This version already included tools and syscalls to simulate the manager of process, memory, and files. The validation results of the first version reveal great improvement needs, and then, we develop the second version called MOSS 0.2. For details of the validation process, includes target audience, profile, survey and discussion see ref. [24].

B. MOSS 0.2 Description

The didactical OS MOSS 0.2 provides an interface for the programmer user through *syscalls* and an interface for the final user (student) through graphical elements as a *tool* of MARS, as illustrated in Fig. 1. Both interfaces make viable, practical activities and classroom experiments in OS courses.

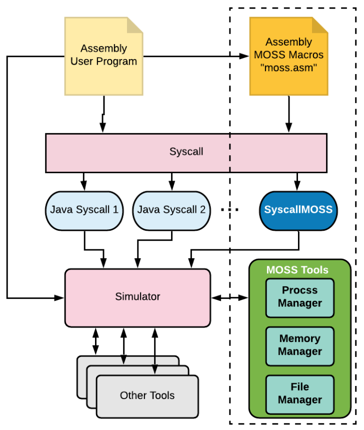
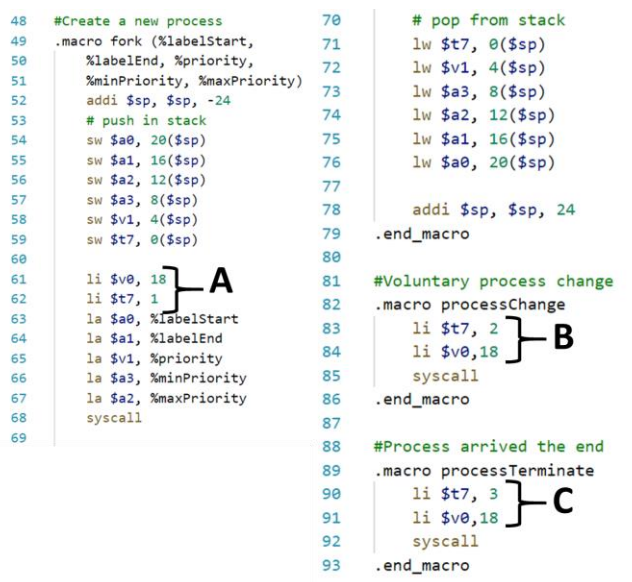


Fig. 1 MOSS Block diagram. Details for MOSS syscall and tools.

1) *Syscalls interface:* The main decision in the project of the MOSS conception is to simplify all system call using a one master syscall, labeled as SyscallMOSS. SyscallMOSS has a unique code, 18, and three functions (*fork*, *processChange* and *processTerminate*). The parameter in the register $t7 selects the function. In MIPS, one system call is selected by passing it specific code through register $v0, and the complementary parameters if required by the syscall. Thus at MOSS+MARS environment, the SyscallMOSS is invoked passing value 18 to register $v0, and the selected function parameters at register $t7 must be: 1 to create a new process (*fork*); 2 to spontaneous process schedule (*processChange*); and 3 to end a process (*processTerminate*). Each function can also use other registers for specific usage.



```
#Create a new process
.macro fork (%labelStart,
    %labelEnd, %priority,
    %minPriority, %maxPriority)
    addi $sp, $sp, -24
    # push in stack
    sw $a0, 20($sp)
    sw $a1, 16($sp)
    sw $a2, 12($sp)
    sw $a3, 8($sp)
    sw $v1, 4($sp)
    sw $t7, 0($sp)

    li $v0, 18
    li $t7, 1
    la $a0, %labelStart
    la $a1, %labelEnd
    la $v1, %priority
    la $a3, %minPriority
    la $a2, %maxPriority
    syscall

    # pop from stack
    lw $t7, 0($sp)
    lw $v1, 4($sp)
    lw $a3, 8($sp)
    lw $a2, 12($sp)
    lw $a1, 16($sp)
    lw $a0, 20($sp)

    addi $sp, $sp, 24
.end_macro

#Voluntary process change
.macro processChange
    li $t7, 2
    li $v0,18
    syscall
.end_macro

#Process arrived the end
.macro processTerminate
    li $t7, 3
    li $v0,18
    syscall
.end_macro
```

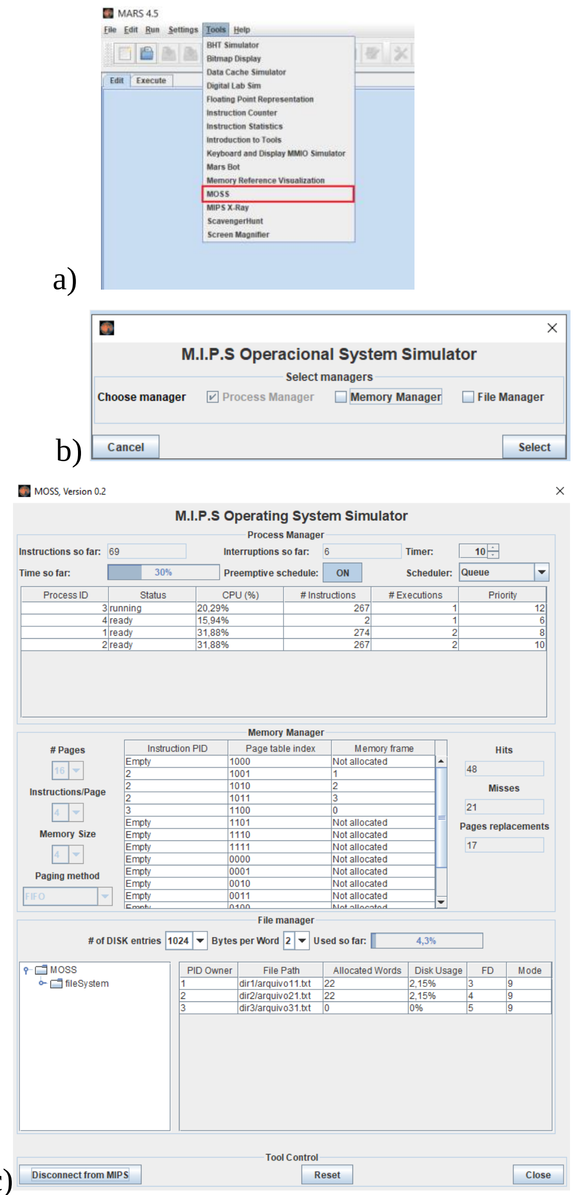
Fig. 2 Macro file "moss.ams" packaged in MOSS 0.2. Details for SyscallMOSS invoked (18) and function selection (1, 2, or 3).

For easy uses, sharing, and convenience, the macro file "MOSS.asm" packaged in MOSS 0.2. As present in Fig. 2, the "moss.asm" contents the SyscallMOSS functions. As indicated in the cases A, B, and C in the figure, each macro has specific parameters, saves the appropriate values at the registers, and invokes the SyscallMOSS passing 18 to register $v0.

2) *Tool interface:* The MOSS can be accessed through MARS at the menu Tools. One of the three managers (process, memory, and file) can be selected. The MOSS interface is present in Fig. 3. In MOSS project conception, the process manager is hierarchy high, so that it cannot be disabled. Furthermore, in the next section, we will present details of the MOSS 0.2, the managers' contents and descriptions, and the practical activities and implementation.

To exemplify the operation of the MOSS Syscalls and Tool interfaces, it is necessary for the creation and execution of some processes. In this environment, a single file with the assembly code of all processes must be assembled. The macro "MOSS.asm" can also be included to enable the processes to use the SyscallMOSS functions more abstractly, through the macros. After that, MOSS should be connected to the MIPS.

Fig. 4 illustrates a single file that included the macro "moss.asm" and creates three processes Program1 (P1), Program2 (P2) e Idle (P3). Labels are used to define the beginning and the end of one process. The labels are passed to MOSS as parameters of the macro *fork*.



b) Choose manager: [✓ Process Manager] Memory Manager File Manager

c) Process Manager

Instructions so far: 69 Interruptions so far: 6 Timer: 10
Time so far: 30% Preemptive schedule: ON Scheduler: Queue

Process ID	Status	CPU (%)	# Instructions	# Executions	Priority
3	running	20,29%	267	1	12
4	ready	15,94%	2	1	6
1	ready	31,88%	274	2	8
2	ready	31,88%	267	2	10

Memory Manager

Pages: 16 Instructions/Page: 4 Memory Size: 4 Paging method: FIFO
Hits: 48 Misses: 21 Pages replacements: 17

Instruction PID	Page table index	Memory frame
Empty	1000	Not allocated
2	1001	1
2	1010	2
2	1011	3
3	1100	0
Empty	1101	Not allocated
Empty	1110	Not allocated
Empty	1111	Not allocated
Empty	0000	Not allocated
Empty	0001	Not allocated
Empty	0010	Not allocated
Empty	0011	Not allocated
Empty	0100	Not allocated

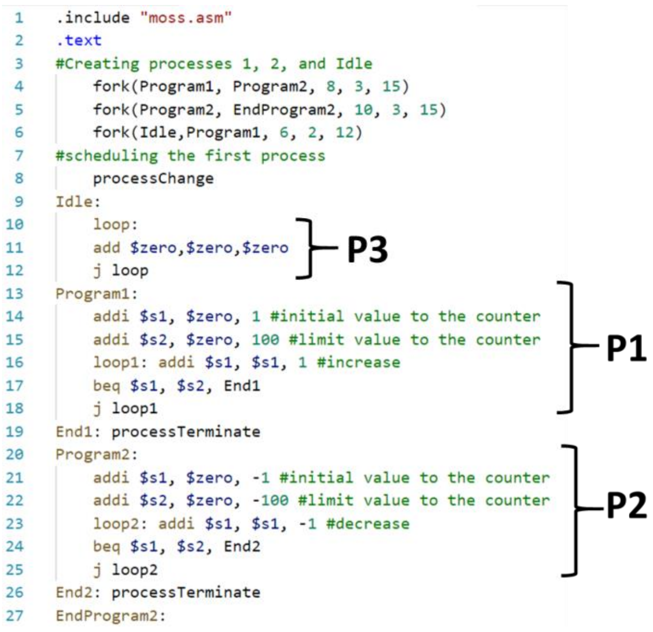
File manager

# of DISK entries 1024 Bytes per Word 2 Used so far: 4,3%

PID Owner	File Path	Allocated Words	Disk Usage	FD	Mode
1	dir1/arquivo11.txt	22	2,15%	3	9
2	dir2/arquivo21.txt	22	2,15%	4	9
3	dir3/arquivo31.txt	0	0%	5	9

Fig. 3 a) Menu tools of the MARS, with detais for MOSS managers as a tool. Details of interface for choosing manager. b) MOSS 0.2. interface to choose the managers; c) MOSS 0.2. interface with the three mangers for process, memory and files.

The values for initial, minimum, and maximum priorities of each process are also informed. The processes are set in the running state. Then when processChange is called, the Schedule (process manager) chooses a process, and the management can be done by MOSS interface. Process P1 performs a sum within a loop until it reaches 10 and call processTerminate. P2 performs a subtraction within a loop until it reaches -10. Note that P1 and P2 use the same registers. The MOSS process manager must be able to ensure the context switch. Already P3 performs an "infinity" loop without syscalls, so that at least one process still executes even the others end.



```
.include "moss.asm"
.text
#Creating processes 1, 2, and Idle
    fork(Program1, Program1, 8, 3, 15)
    fork(Program2, EndProgram2, 10, 3, 15)
    fork(Idle,Program1, 6, 2, 12)
#scheduling the first process
    processChange
Idle:
    loop:
    add $zero,$zero,$zero
    j loop
Program1:
    addi $s1, $zero, 1 #initial value to the counter
    addi $s2, $zero, 100 #limit value to the counter
    loop1: addi $s1, $s1, 1 #increase
    beq $s1, $s2, End1
    j loop1
End1: processTerminate
Program2:
    addi $s1, $zero, -1 #initial value to the counter
    addi $s2, $zero, -100 #limit value to the counter
    loop2: addi $s1, $s1, -1 #decrease
    beq $s1, $s2, End2
    j loop2
End2: processTerminate
EndProgram2:
```

Fig. 4 Assembly example for creating and monitoring three processes in MOSS+MARS environment.

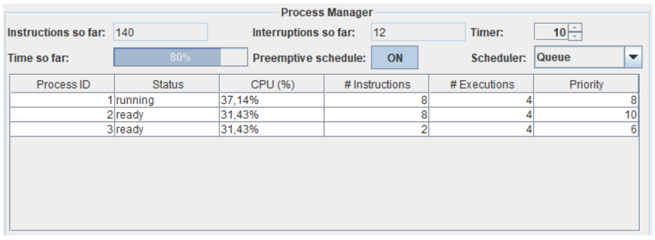
C. Methodology aspects

Follow the approach of evaluating units at UFERSA, and the methodology presents practical activities and implementations for each one.

1) *Activity and Implementation at Unit I*: The first activity consists in the development of the process manager. Therefore, the students are being instructed to create Java classes that represent the Process Control Block (PCB), Process Table and Scheduler. A review of the concepts about PCB and its purpose of storing process information (identifier, registers contents, and process state) and information of the other managers, which will be implemented in the future Units, has considerable allocated time, around 40% of the course schedule.

The Process Table allows the creation and removal processes PCB as well as the movement of the created PCBs, within the lists according to the processes state. The Scheduler class implements the algorithms for process scheduling. This class should be able to implement several algorithms, which can be chosen at project or simulation time, considering or not the preemption. For the preemptive algorithms, the activity required that the timer interruption must be developed, simulated the process change using the amount of executed instructions so far. As an interface to the programmer, students are instructed to develop 3 syscalls or 3 functions in the same syscall: fork (create process), processChange (the scheduler is explicitly called by the process to switch context) and processTerminate (the process is terminated and no longer scheduled). Students are also instructed to develop a file of macros to facilitate the use of generated syscalls, easily for passing parameters through specific registers, as the “moss.asm”.

Fig. 5 shows the target Process Manager (PM) interface of the MOSS 0.2 as a result of the students’ implementation. A button selects the preemptive schedule approach. When preemption is active (*Preemptive schedule button is “ON”*), the user can determine the amount of executed instruction (*Timer option*) before the interruption occurs. The default value for the Timer is 10 instructions. It is important to mention that when the non-preemptive option is active, the only way to simulated process context switch is through processChange explicitly called in assembly code. The scheduling algorithms available in MOSS 0.2 are queue (*default*), dynamic priority and lottery (*random choice*).



Process Manager

Instructions so far: 140 Interruptions so far: 12 Timer: 10

Time so far: 80% Preemptive schedule: ON Scheduler: Queue

Process ID	Status	CPU (%)	# Instructions	# Executions	Priority
1	running	37,14%	8	4	8
2	ready	31,43%	8	4	10
3	ready	31,43%	2	4	6

Fig. 5 Process Manager (PM) of the MOSS 0.2. Configuration parameters and process information are indicated.

After a call of *fork* syscall the PM creates the entire data structures for the new process and exhibits in a new row at the interface table. The interface table contents the follow information: process ID (unique identifier); Status (ready or running); CPU (%) the ration of the CPU utilization by the process; # Instructions represents the number of instructions of the process; # Execution represents how many time the process was run; and Priority (actual priority of the process). The PM interface still presents a counter for the amount of executed instructions (Instructions so far) of the existing processes; the counter for the next interruptions in preemptive mode (Interruptions so far) and the progress bar (Time so far) which exhibit the time percent in number of instructions until the next system interruption.

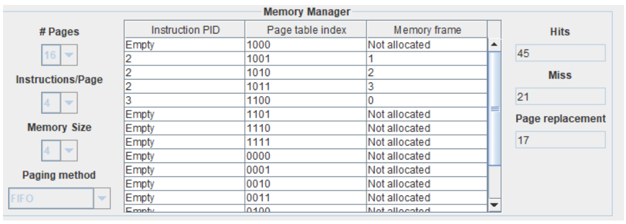
During process execution, when the timer interruption occurs, or an explicitly call for *processChange*, the PM is responsible for the process switch. The student’s implementation must change the states of the actual running process, save the associate PCB and apply the policy of the scheduling algorithm. The PCB should be removed when process call *processTerminate* and finish execution. With preemption mode active based on the Fig. 4 example, is possible to observe at MOSS interface, the update of the $s1 and $s2 register values concomitant with the execution and switch the context of processes P1 and P2. With this scenario, it is possible to accomplish activities related to processes synchronization, develop solutions such as semaphores and mutex, implements new schedule algorithms or measure performance of the already existing algorithms.

2) *Activity and Implementation at Unit II:* The second activity consists in the development of the memory manager (MM). In the beginning, the students are lead to modify their PCB class and fork syscall to include the memory address to indicate the maximum and minimum memory space limits. Any improper access to memory address space of another process should interrupt the process execution, and the user should be alerted.

Besides, students also must construct the MOSS interface for MM. The MM should store global attributes for all processes, among which: virtual page size (in terms of MIPS amount of instructions); the number of pages; the maximum quantity of pages by a process, and algorithms for page replacement. These attributes can be adjusted through the MM interface.

Fig. 6 presents the MM interface of the MOSS 0.2. The user can configure the number of virtual pages of the OS (*# Pages*); the amount of instructions contained on a page

(*Instructions/Page*); the number of frames to represents the physical memory storage (*Memory Size*); and the page replacement algorithms (*Page method*). The students should feasible the FIFO, Second change and NRU algorithms [1].



Memory Manager

# Pages	Instructions/Page	Memory Size	Paging method
16	4	4	FIFO

Instruction PID	Page table index	Memory frame
Empty	1000	Not allocated
2	1001	1
2	1010	2
2	1011	3
3	1100	0
Empty	1101	Not allocated
Empty	1110	Not allocated
Empty	1111	Not allocated
Empty	0000	Not allocated
Empty	0001	Not allocated
Empty	0010	Not allocated
Empty	0011	Not allocated
Empty	0100	Not allocated

Hits	Miss	Page replacement
45	21	17

Fig. 6 Memory Manager (MM) of the MOSS 0.2 Configuration parameters and memory information are indicated.

For memory management, MM keeps the necessary information for every process like bits for the indication if a page was modified or refer; bits for protection (Read, Write, Execute) and bits for Presence/Absence of page on physical memory. During the process execution, the MM also exhibits a mapping table with the process using a page (*Instruction PID*); the paging table index (*Page table index*), and the allocated memory frame (*Memory frame*). Furthermore, the quantities of virtual addresses that are already in memory frame (*Hits*), address those requires paging from disk (*Miss*) and the number of page replacement (*Page replacement*).

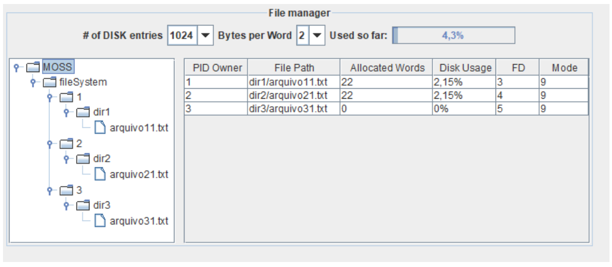
In this activity, no new syscall is required. The MM maintains the accounting of which memory part is in use or not, just for codes fragment. All memory allocated to the code segment is starting to track immediately after the process creation. So that, any process attempts of memory access beyond space address boundaries interrupt their execution. Free memory allocation is automatically performed after the process end.

In MOSS environment, the MM considers that any processes are created without any allocated memory. While the process advances on execution, the memory manager paging the virtual addresses from the disk into physical addresses and updates into the virtual memory table. At the same time, counting miss and hits, and invoking the page replacement algorithm when necessary.

3) *Activity and Implementation at Unit III:* The third activity consists in the development of the file system manager (FM). The students must be lead to modify syscall that already in MARS. The syscalls with code 13 to 16 [23] allow manipulation of files (open, read, write, and close).

The design of the file manager must be planning by the student. The information necessary for management as the block size of a disk, the accounting of free blocks and inode data structure must be implemented. The FM associates an inode to a file and should allocate dynamically block units of the disk as the file grow or shrink. The list of open files, file owners, and permission credentials for file manipulation should also be presented in the FM design.

Fig. 7 shows the FM interface of the MOSS 0.2. It is possible to set the amount of disk allocation unit (*# of Disk entries*) and the quantity of byte used by allocation unit (*Bytes per Word*); see the statistic about disk usage (*Used so far*); a tree-view of the directory and open files for file manipulation. In addition for each open file an information table, content the owner process (*PID Owner*), the absolute file path (*File Path*), the amount of words allocated by the file (*Allocated Words*), statics of disk usage by the file (*Disk Usage*), file identifier (*FD or File Descriptor*) and the file open mode (*Mode*) is updating.



File manager

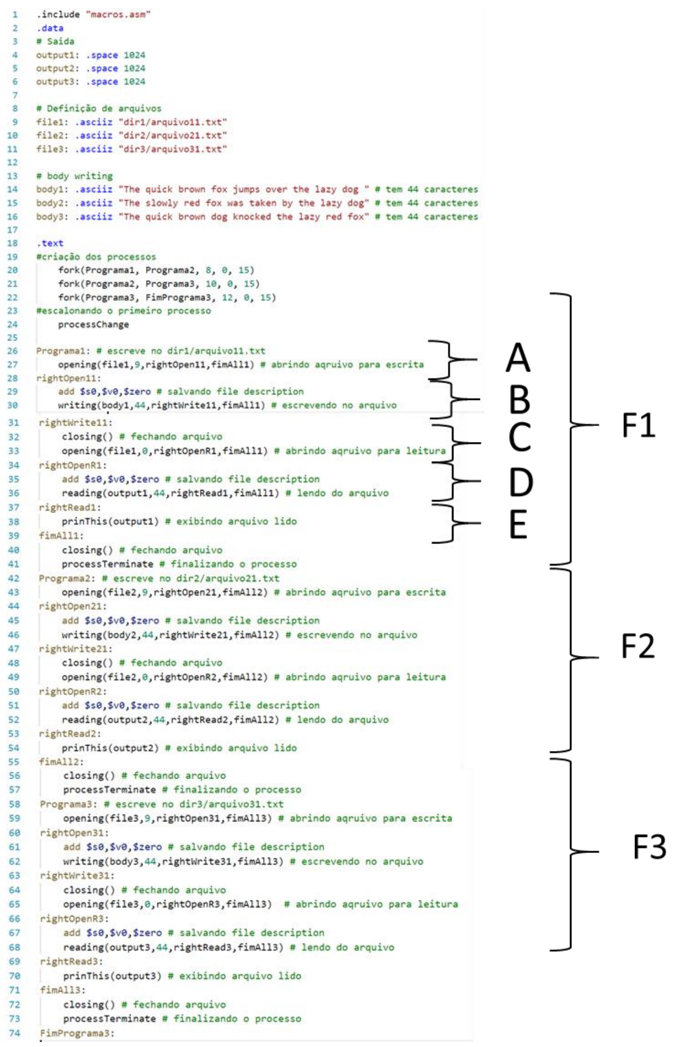
# of DISK entries: 1024 Bytes per Word: 2 Used so far: 4,3%

```
MOSS
  fileSystem
    1
      dir1
        arquivo11.txt
    2
      dir2
        arquivo21.txt
    3
      dir3
        arquivo31.txt
```

PID Owner	File Path	Allocated Words	Disk Usage	FD	Mode
1	dir1/arquivo11.txt	22	2,15%	3	9
2	dir2/arquivo21.txt	22	2,15%	4	9
3	dir3/arquivo31.txt	0	0%	5	9

Fig. 7 File Manager (FM) of the MOSS 0.2. Configuration parameters and process infromations are indicated.

To test FM interface, students writing an assembly code, as illustrated in Fig. 8.



```
.include "macros.asm"
.data
# Saida
output1: .space 1024
output2: .space 1024
output3: .space 1024

# Definição de arquivos
file1: .asciiz "dir1/arquivo11.txt"
file2: .asciiz "dir2/arquivo21.txt"
file3: .asciiz "dir3/arquivo31.txt"

# body writing
body1: .asciiz "The quick brown fox jumps over the lazy dog " # tem 44 caracteres
body2: .asciiz "The slowly red fox was taken by the lazy dog" # tem 44 caracteres
body3: .asciiz "The quick brown dog knocked the lazy red fox" # tem 44 caracteres

.text
#criação dos processos
    fork(Programa1, Programa2, 8, 0, 15)
    fork(Programa2, Programa3, 10, 0, 15)
    fork(Programa3, FimPrograma3, 12, 0, 15)
#escalonando o primeiro processo
    processChange

Programa1: # escreve no dir1/arquivo11.txt
    opening(file1,9,rightOpen11,fimAll1) # abrindo aqruivo para escrita
rightOpen11:
    add $s0,$v0,$zero # salvando file description
    writing(body1,44,rightWrite11,fimAll1) # escrevendo no arquivo
rightWrite11:
    closing() # fechando arquivo
    opening(file1,0,rightOpenR1,fimAll1) # abrindo aqruivo para leitura
rightOpenR1:
    add $s0,$v0,$zero # salvando file description
    reading(output1,44,rightRead1,fimAll1) # lendo do arquivo
rightRead1:
    prinThis(output1) # exibindo arquivo lido
fimAll1:
    closing() # fechando arquivo
    processTerminate # finalizando o processo
Programa2: # escreve no dir2/arquivo21.txt
    opening(file2,9,rightOpen21,fimAll2) # abrindo aqruivo para escrita
rightOpen21:
    add $s0,$v0,$zero # salvando file description
    writing(body2,44,rightWrite21,fimAll2) # escrevendo no arquivo
rightWrite21:
    closing() # fechando arquivo
    opening(file2,0,rightOpenR2,fimAll2) # abrindo aqruivo para leitura
rightOpenR2:
    add $s0,$v0,$zero # salvando file description
    reading(output2,44,rightRead2,fimAll2) # lendo do arquivo
rightRead2:
    prinThis(output2) # exibindo arquivo lido
fimAll2:
    closing() # fechando arquivo
    processTerminate # finalizando o processo
Programa3: # escreve no dir3/arquivo31.txt
    opening(file3,9,rightOpen31,fimAll3) # abrindo aqruivo para escrita
rightOpen31:
    add $s0,$v0,$zero # salvando file description
    writing(body3,44,rightWrite31,fimAll3) # escrevendo no arquivo
rightWrite31:
    closing() # fechando arquivo
    opening(file3,0,rightOpenR3,fimAll3)  # abrindo aqruivo para leitura
rightOpenR3:
    add $s0,$v0,$zero # salvando file description
    reading(output3,44,rightRead3,fimAll3) # lendo do arquivo
rightRead3:
    prinThis(output3) # exibindo arquivo lido
fimAll3:
    closing() # fechando arquivo
    processTerminate # finalizando o processo
FimPrograma3:
```

Fig. 8 Assembly example for testing FM of MOSS 0.2. Files F1, F2, and F3 are open (A) for writing (B), close (C), and open for reading (D). Then the content of the file is shown in the console MARS environment (E).

To communicate with MOSS, the macro file "moss.asm" must be included. In the example, three processes are created (P_i), and each one opens a file (F_i), writing the content of the variable body ($body_i$) and close the respective file. Next, the process reopens the same file in reading mode, reads it and exhibits the content in MARS console. Finally, the file is closed, and the process ended.

To perform the proposed activity, the students should modify the code of syscall *open* to receive the parameters: filename in register $a0 and the number which represents the open mode through the register $a1, assuming the values 0-read, 1-write, and 9-append. The FM verifies all constraints when the specific mode is selected. The opening syscall returns the file descriptor, a unique number to representing the new open file. The file descriptor is passing as a parameter for *read*, *write* and *close* syscalls.

The *write* syscall request a buffer containing the data and the number of bytes to be written. The FM checks if there is enough disk space. The selected option on FM interface determines the number of bytes per word. If there is enough space in disk, the file is written, the number of stored disk blocks is subtracted, and returns the number of bytes written. The syscall *read* request the number of bytes to read and the buffer where the read data will be stored. The successful operation returning the number of bytes read in the $v0 register. To close an open file, the user must only inform the file descriptor when performing the system call.

RESULTS AND DISCUSSIONS

The use of the MOSS based methodology has resulted in positive impacts in the course schedule and practical activities supplies, as in student's engagement, motivation, and performance. In this section, we will present an impact analysis of the proposed methodology that resulted in the emergence of MOSS, which in turn improved the application of the methodology and opened a new perspective.

The analyzed aspects are the students' grades and the evasion rate in each semester. The periods analyzed are before the methodology (from 2010.2 to 2014.2) and after the beginning, the methodology (since 2015.1). The first version of MOSS was created in 2017.2. Then, in the last two semesters, 2018.1 and 2018.2, MOSS started to be used as an activity platform, as an illustrative example and as inspiration for students to develop their OS simulators.

First, we will analyze the impact of the methodology on the grades and evasion rate, utilizing Fig. 9.

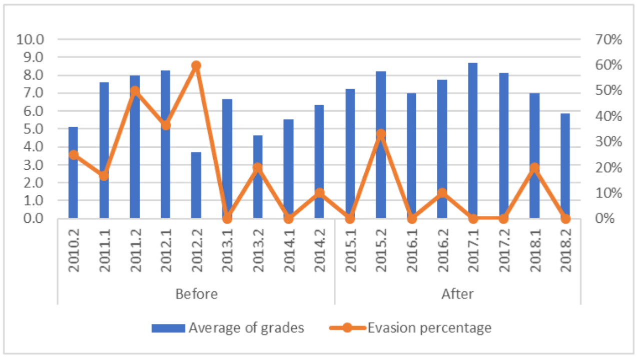


Fig. 9 Values of grades average and evasion rate before and after the uses of the proposed methodology.

In this figure, we present the grades averages for each semester as well as the percentage of dropout (or evasion) rate. Before the methodology, there was a greater lack of students' interest in the OS course, which is reflected in the behavior instability of the grade averages and high dropout rates. In 2012.2, the MINIX was used for practical activities. However, the students encountered a lot of difficulties, expressed in the notes and in the evasion rate. After using the methodology, the grade averages became more stable with a minimum of 6, while evasion has decreased, with a maximum of 30% and generally below 10%.

The value of the averages in each semester can not exactly reflect the phenomenon of improvement. Therefore, we present in Fig. 10 the frequencies of the notes (from 0 to 10), "before" and "after" the methodology, through the bar graph. In the "before" period, the highest frequency was already between grades 6 and 7, but there were also great frequencies of grades less than 1. In the "after" period, the frequency of the lowest grades decreased, and the highest frequency became around 8.

To verify the distribution of these grades and to understand the trend caused by the methodology, we also show in Fig. 10 the graph of the probability mass density function (PMF) calculated for 100 points for both "Before" and "After". A feature of PMF is that its peak is located in the mean of the values, and its slope represents how the values are distributed. Thus, it can be seen that the "before" PMF was more spread with a peak around 6, while the "after" PMF peaks between 7 and 8 with more abrupt decreases for the extremes, especially for notes below 5.

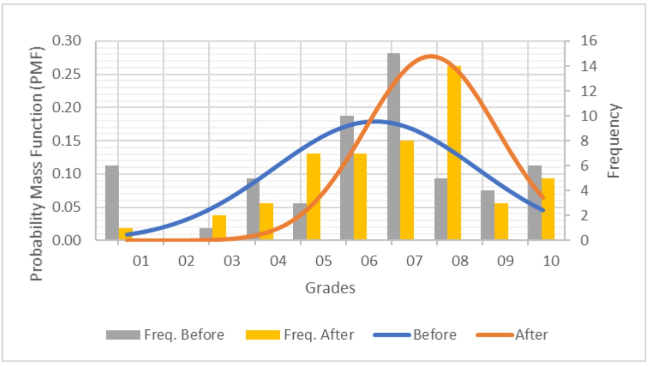


Fig. 10 Frequency of each student grade before and after the uses of the proposed methodology. Details for the tendency line.

Through data analysis, we believe that the methodology was effective and presents positive efforts in the issues related to students' performance and interest. As a result of the methodology, we develop the simulated educational operating system MOSS, which has been used to improve the methodology, allowing the development of practical activities even in a short period as a semester.

CONCLUSIONS AND FUTURE WORKS

In this work, we present the MOSS 0.2, an educational operating system for MIPS; the proposed methodology that generates and can be used with it; and an evaluation of this methodology which was applied in the UFERSA OS courses, that used it during eight semesters.

The methodology is introduced for a short OS course with strong integration with Organization and Computer Architecture course based on MIPS. It is adopted the incremental three-unit method present in UFERSA. In the first unit activities and implementations regards process manager; in the second unit lectures and activities for memory manger; and finally lectures about I/O devices and activities/implementations for file manager.

The methodology is already be used in UFERSA, and MOSS serves as inspiration for students to develop their own "like-MOSS" simulators. In future works, we intend to expand and improve the methodology with other courses like

Compiler, microprocessors & microcontrollers or embedded systems and basic software. The improvement methodology to OS course still includes I/O manager and new guidelines to all functionalities embedded in MOSS.

References

[1] A. S. Tanenbaum, *Modern Operating Systems*, 4th ed. Pearson PLC, 2014.

[2] A. S. Tanenbaum, “MINIX 3”. [Online]. Disponível em: http://www.minix3.org/.

[3] A. S. Tanenbaum, “Lessons Learned from 30 Years of MINIX”, *Commun ACM*, vol. 59, nº 3, p. 70–78, fev. 2016.

[4] A. S. Tanenbaum e A. Woodhull, *Operating Systems: Design and Implementation: The MINIX Book*, 3rd ed. Pearson Education International, 2009.

[5] W. Du e R. Wang, “SEED: A Suite of Instructional Laboratories for Computer Security Education”, *J Educ Resour Comput*, vol. 8, nº 1, p. 3:1–3:24, mar. 2008.

[6] R. Román Otero e A. A. Aravind, “MiniOS: An Instructional Platform for Teaching Operating Systems Projects”, in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, New York, NY, USA, 2015, p. 430–435.

[7] J. Ge, B. Ye, X. Fei, e B. Luo, “A Novel Practical Framework for Operating Systems Teaching”, in *2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing*, 2009, p. 596–601.

[8] J. Hill, C. K. Ray, J. R. Blair, e C. A. Carver Jr, “Puzzles and games: addressing different learning styles in teaching operating systems concepts”, in *ACM SIGCSE Bulletin*, 2003, vol. 35, p. 182–186.

[9] H. Yi-Ran, Z. Cheng, Y. Feng, e Y. Meng-Xiao, “Research on teaching operating systems course using problem-based learning”, in *2010 5th International Conference on Computer Science & Education*, 2010, p. 691–694.

[10] J. Nieh e C. Vaill, “Experiences teaching operating systems using virtual platforms and linux”, *SIGCSE Bull*, vol. 37, nº 1, p. 520–524, 2005.

[11] P. H. d M. M. Penna, M. B. Castro, H. C. d Freitas, J. Méhaut, e J. Caram, “Using the Nanvix Operating System in Undergraduate Operating System Courses”, in *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2017, p. 193–198.

[12] F. B. Machado e L. P. Maia, *Arquitetura de Sistemas Operacionais*, 4º ed. LTC, 2007.

[13] L. P. Maia, F. B. Machado, e A. C. Pacheco Jr., “A Constructivist Framework for Operating Systems Education: A Pedagogic Proposal Using the SOsim”, in *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, New York, NY, USA, 2005, p. 218–222.

[14] D. S. de Carvalho, G. da R. Balthazar, C. R. Dias, M. A. P. Araújo, e P. H. R. Monteiro, “Simulador para a Prática de Sistemas Operacionais”, *Rev. Eletrônica FMG*, vol. 3, 2006.

[15] G. A. Tonini, e S. C. Lunardi, “Simulador para o Aprendizado de Sistemas Operacionais”, apresentado em Simpósio de Informátia, 2006.

[16] D. Jones, “RCOS.Java: An Animated Operating System for Computer Science Education”, in *Proceedings of the 1st Conference on Integrating Technology into Computer Science Education*, New York, NY, USA, 1996, p. 233–.

[17] D. Jones, e A. Newman, “A Constructivist-based Tool for Operating Systems Education”, in *Proceedings of ED-MEDIA 2002--World Conference on Educational Multimedia, Hypermedia & Telecommunications*, Waynesville, 2002, p. 882–883.

[18] W. A. Christopher, S. J. Procter, e T. E. Anderson, “The Nachos Instructional Operating System”, EECS Department, University of California, Berkeley, UCB/CSD-93-739, nov. 1992.

[19] M. Goldweber, R. Davoli, e M. Morsiani, “The Kaya OS Project and the μMPS Hardware Emulator”, in *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, New York, NY, USA, 2005, p. 49–53.

[20] S. Fernandes e I. S. Silva, “Relato de Experiência Interdisciplinar Usando MIPS”, *IJCAE Int. J. Comput. Archit. Educ.*, vol. 6, nº 1, p. 52–61, 2017.

[21] K. Vollmar e P. Sanderson, “MARS: An Education-Oriented MIPS Assembly Language Simulator”, in *ACM SIGCSE*, 2006, p. 239–243.

[22] Oracle, “Java”. [Online]. Disponível em: https://www.java.com.

[23] P. Sanderson e K. Vollmar, “An Assembly Language I.D.E. To Engage Students Of All Levels”, apresentado em 2007 CCSC, 2007.

[24] A. Costa, S. Silva, T. Macedo, e S. Fernandes, “MOSS - Uma Ferramenta para o Auxílio do Ensino de Sistemas Operacionais”, in *Anais do XXIX Simpósio Brasileiro de Informática na Educação (SBIE 2018)*, Fortaleza, 2018.