ASTES

Advances in Science, Technology and Engineering Systems Journal Vol. 5, No. 6, 96-107 (2020)

www.astesj.com

Special Issue on Multidisciplinary Innovation in Engineering Science & Technology

ASTESJ

ISSN: 2415-6698

Using TOST in Teaching Operating Systems and Concurrent Programming Concepts

Tzanko Golemanov*, Emilia Golemanova

Department of Computer Systems and Technologies, University of Ruse, Ruse, 7020, Bulgaria

A R T I C L E I N F O

Article history:
Received: 30 July, 2020
Accepted: 15 October, 2020
Online: 08 November, 2020

Keywords:
Operating Systems
Concurrent Programming
Teaching Tools

A B S T R A C T

The paper is aimed as a concise and relatively self-contained description of the educational environment TOST, used in teaching and learning Operating Systems basics such as Processes, Multiprogramming, Timesharing, Scheduling strategies, and Memory management. TOST also aids education in some important IT concepts such as Deadlock, Mutual exclusion, and Concurrent processes synchronization. The presented integrated environment allows the students to develop and run programs in two simple programming languages, and at the same time, the data in the main system tables can be monitored. The paper consists of a description of TOST system, the features of the built-in programming languages, and demonstrations of teaching the basic Operating Systems principles. In addition, some of the well-known concurrent processing problems are solved to illustrate TOST usage in parallel programming teaching.

1. Introduction

This paper is an extension of work originally presented in 29th Annual Conference of the European Association for Education in Electrical and Information Engineering (EAEEIE) [1].

Operating Systems (OS) is one of the most fundamental courses in Computer Engineering, Computer Science, and Information Systems curricula. At the same time, students are faced with a lot of problems in learning OS basics, due to not fully understanding the wide range of techniques, strategies, and architectures involved in OS modules. The complexity and depth of concepts in this area requires a careful and detailed explanation from the educator in order to reach a better apprehending. Developing concurrent programs in an operating system environment (Unix, Linux, etc.) asks for students to already have prerequisite skills in both the C-programming and the operating system. Usually, such level of experience is difficult to be obtained at an undergraduate degree, thus a preferable approach would be to use a convenient and effective teaching tool that helps the students in their uneasy endeavor to apprehend and master the OS concepts.

We have assigned the widely spread tools for studying OS into the following two groups:

- Instructional Operating Systems
- Visual OS simulators

The systems from the first group (MINIX [2], Nachos [3], Xinu [4], Pintos [5], GeekOS [6]) run on real hardware and are too complicated to install and use. The main disadvantage of the instructional OS is the requirement students to be familiar with system-level programming in C, C++, Java or assembly language, and to be able to study large program codes (more than 4K program lines of Geek-OS and 15K of MINIX). Most of these systems do not have graphical visualizations, which makes them not very suitable for teaching purposes. In such cases, students spend a lot of time learning the systems themselves and how to use them properly.

In contrast, the visual tools from the second group (SOsim [7], SchedulerSim [8], CPU Sim [9]) are easier to use and convenient, however, they present rather restricted and pre-specified abilities (e.g. simulation of just one OS module or subsystem). For example, the TROJAN [10] simulator helps in teaching multiprocessor organization, the cache utilization, and the network traffic issues, while SIME [11] visualizations focus on memory management, but without shared memory concepts. Additionally, some of these systems (Proc OS [12], Alg OS [13]) do not connect theoretical concepts to a running code. Others present simulations of only one example problem (e.g. the Producer/Consumer problem [14]).

The authors propose a newly-developed by them OS teaching system named ***TOST*** *(Teaching in Operating Systems Tool)*. It has

*Corresponding Author: Tzanko Golemanov, Ruse University, Ruse, Bulgaria, tgolemanov@uni-ruse.bg

been created in response to the requirement of the educational tool to be straightforward to use, yet powerful enough, allowing the students to not only experiment with OS settings, architectures, and strategies but also to create and run any concurrent programs they want. This imposes the need for TOST to be developed as an integrated environment consisting of a code editor, a high-level language compiler, a virtual processor emulator, an interface for setting and modifying OS parameters, and a visualization of the content of the main OS tables.

The structure of the paper is as follows: Section 2 is an introduction to TOST giving an overview of the TOST interface, the grammars of the built-in programming languages, the manners to develop and execute concurrent processes, as well as the ways to set and modify system options, and to monitor the system tables' data. Section 3 is devoted to teaching Operating System basics, such as Process Management, Process Scheduling, and Memory Management. Section 4 describes the use of TOST in teaching Concurrent Programming concepts, specifically Mutual Exclusion (Dekker's algorithm, Peterson's algorithm, Test_And_Set instruction, and Semaphores), Interprocess Synchronization (Sleeping Barber problem), and Deadlock (Dining Philosophers problem). In Section 5, a student evaluation is presented. We conclude in Section 6 and discuss some further improvements to TOST.

2. The TOST System

The main objective of an educational tool in OS is to give students an inside view of the operating system. At the same time, in our opinion, the tool should meet the following basic requirements:

- to operate on an accessible hardware and software platform
- to support multiprogramming and concurrent processes
- to be with a well-known interface
- to support well-known programming languages with concurrency features
- to allow students the experimentation with the basic OS parameters and options.

TOST has been developed with exactly this idea in mind. It allows students to develop and execute concurrent processes, while at the same time they can change the basic settings of the OS (CPU scheduling, Quantum size, Memory management, Virtual memory strategies, etc.) and to monitor the main system tables (Processes, Blocked, Ready, Semaphores, etc.).

The proposed integrated environment includes:

- a multi-tasking, and a multi-window operating system
- editors and compilers for two simple programming languages
- a virtual processor emulator.

The last TOST version is developed with Embarcadero® Delphi within Embarcadero RAD Studio 10.3.3 Rio. It has no installation procedure, the executable file of the tool is very small in size (less than one MB), and it operates on any MS Windows environment. TOST is easy to learn and use, and allows students to become familiar with the capabilities and the limitations of the built-in programming languages within a short time (about an hour).

The first TOST environment is introduced in [15], while the focus of [1] is on using TOST in teaching mutual exclusion, synchronization, and deadlock. This paper expands the application area of TOST, describing its usage for teaching and learning OS concepts.

2.1. Programming in TOST

A general view of the current TOST main menu is depicted in Figure 1.

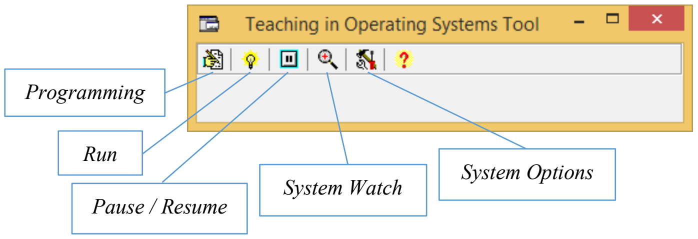


Figure 1: TOST interface

Using TOST, students can develop and execute concurrent programs in two simple programming languages - PASCAL-style and C-style. The grammars of these languages, consist of roughly twenty grammar productions only, which are presented in Figure 2 and Figure 3 in Extended BNF (ISO/IEC 14977).

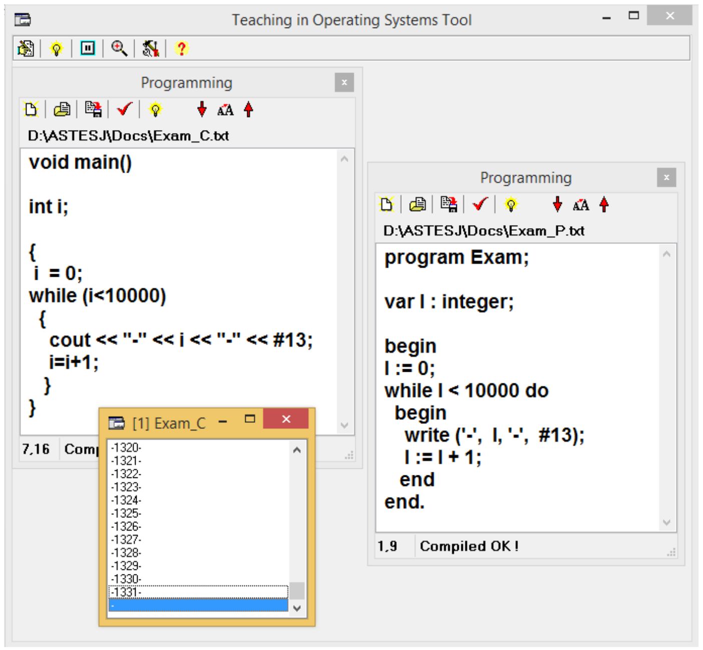
```
program    = "void" "main" "(" ")" body
body       = [ { type identifier { "," identifier } } ] block
type       = "int" | "char" | "bool" | "shared" | "semaphore"
block      = "{" statement { ";" statement } "}"
statement  = block | assignment | if | while | read | write | lock | unlock |
             init | wait | signal
assignment = identifier "=" expression
expression = expr [ ( "<" | ">" | "==" ) expr ]
expr       = term { ( "+" | "-" | "or" ) term }
term       = factor { ( "*" | "/" | "and" ) factor }
factor     = [ "not" ] ( identifier | constant | random | "(" expression
             ")" )
constant   = number | "'" char "'" | "#" ascii_code | "true" | "false"
random     = "random" "(" expression ")"
if         = "if" "(" expression ")" statement
while      = "while" "(" expression ")" statement
read       = "cin" ">>" identifier
write      = "cout" "<<" expression { "<<" expression }
lock       = "lock" "(" identifier ")"
unlock     = "unlock" "(" identifier ")"
init       = "init" "(" identifier "," expression ")"
wait       = "wait" "(" identifier ")"
signal     = "signal" "(" identifier ")"
```

Figure 2: C-style language grammar productions

```
program    = [ "program" identifier ";" ] body "."
body       = [ "var" { identifier { "," identifier } ":" type ";" } ] block
type       = "integer" | "char" | "boolean" | "shared" | "semaphore"
block      = "begin" statement { ";" statement } "end"
statement  = block | assignment | if | while | read | write | lock | unlock |
             init | wait | signal
assignment = identifier ":=" expression
expression = expr { ( "<" | ">" | "=" ) expr }
expr       = term { ( "+" | "-" | "or" ) term }
term       = factor { ( "*" | "/" | "and" ) factor }
```

```
factor     = [ "not" ] ( identifier | constant | random | "(" expression
             ")" )
constant   = number | "" char "" | "#" ascii_code | "true" | "false"
random     = "random" "(" expression ")"
if         = "if" expression "then" statement
while      = "while" expression "do" statement
read       = "read" "(" identifier ")"
write      = "write" "(" expression { "," expression } ")"
lock       = "lock" "(" identifier ")"
unlock     = "unlock" "(" identifier ")"
init       = "init" "(" identifier "," expression ")"
wait       = "wait" "(" identifier ")"
signal     = "signal" "(" identifier ")"
```

Figure 3: PASCAL-style language grammar productions


Teaching in Operating Systems Tool
Programming
D:\ASTESJ\Docs\Exam_C.txt

```
void main()

int i;

{
 i = 0;
while (i<10000)
  {
   cout << "-" << i << "-" << #13;
   i=i+1;
  }
}
```

```
-1320-
-1321-
-1322-
-1323-
-1324-
-1325-
-1326-
-1327-
-1328-
-1329-
-1330-
-1331-
-
```

Programming
D:\ASTESJ\Docs\Exam_P.txt

```
program Exam;

var I : integer;

begin
I := 0;
while I < 10000 do
  begin
    write ('-',  I, '-',  #13);
    I := I + 1;
   end
end.
```

Figure 4: Process editing and executing in TOST

In the text bellow we will use C-style only for the programs' definition.

The programming languages in TOST are exclusively for learning purposes and while they are quite simple, there are some features that need to be addressed:

2.1.1. Types of variables

Only a few standard simple data types *integer*, *char* and *boolean* are available for local variables, nevertheless, some data types for concurrent programming purposes are also provided. Variables from the ***shared*** type can be declared in a TOST program. These variables become common for all concurrent processes running in the system. A shared variable is unified, i.e. it can accept any integer, char or boolean values, as well as any data can be extracted from it. As a result, passing through such variable can also be used in order to convert data from one type to another.

Another type of data used in parallel programming is the ***semaphore***. In TOST we implement the original Dijkstra's semaphores concept [16]. A semaphore variable ***S*** consists of two properties - ***S.value*** and ***S.queue***. The property S.value is a non-negative integer, while S.queue is a FIFO list of identifiers of the blocked processes (process ID).

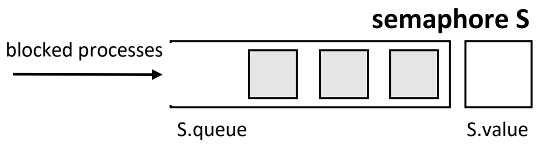


Figure 5: Semaphore's structure

2.1.2. Statements

Although TOST languages maintain only basic statements like assignment, conditional, loop and input/output, some statements for concurrent programming purposes are also available.

The semaphore operations ***init***, ***wait***, and ***signal***, which are defined below, are functions from the OS kernel:

init (S, value)

```
S.value = value
```

wait(S)

```
if (S.value > 0)
  decrement S.value;
else
  block this process and add it's ID in S.queue
```

signal (S)

```
if (S.queue is empty)
     increment S.value;
else
     unblock the first process from S.queue
```

Figure 6: Semaphore operations

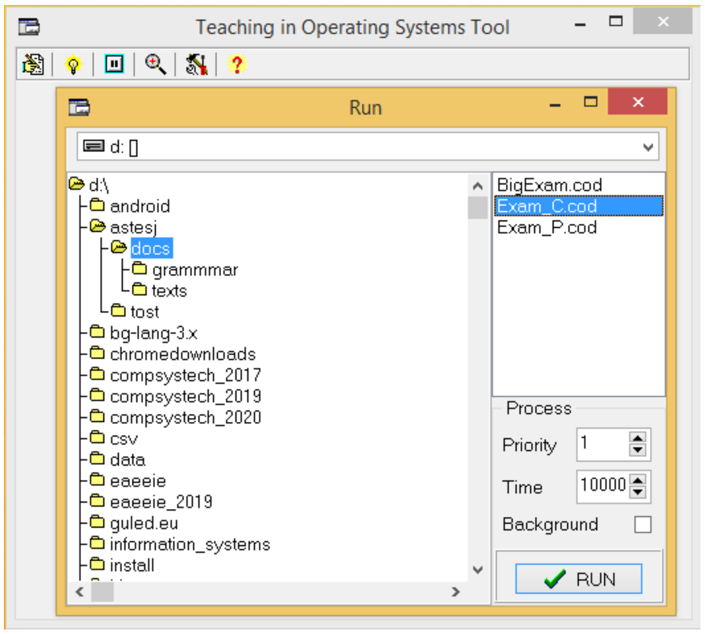
Students can investigate the advantages and disadvantages of using the special assembly language instruction ***Test_and_Set*** [17] through the statements ***Lock*** (CS) and ***Unlock*** (CS), where CS (*Critical Section*) is a boolean shared variable.

Function ***random***(*expression*) which returns a random integer value from *0* to *expression-1* can also be used in some synchronization primitives' implementations.

2.2. Processes Execution Management in TOST

After the successful compilation, an object code file is generated (with COD extension) and it is ready to be run in the environment. The compiled processes can be started from the ***Run*** at the main menu (as well as from the editing windows).

Figure 7 presents the general system view where in the ***Run*** dialog box students can select an object file to be executed, and set the initial parameters of the running process.



Priority 1
Time 10000
Background ☐

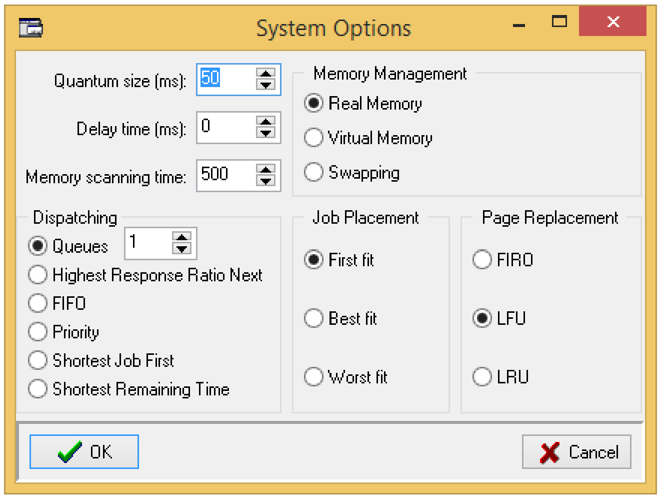
Figure 7: Process running

Priority spin-edit box sets the initial process priority while the total process execution time (used in some scheduling strategies) can be entered in ***Time***. By checking ***Background*** the students can specify the process visual mode (execution in a separate window or behind the scenes). Pressing the ***RUN*** button multiple times will cause the start of several concurrent processes.

If detailed monitoring of running processes is required, a temporary "freeze" can be performed via ***Pause / Resume*** at the main menu.

2.3. TOST System Options

Some basic OS parameters can be set and modified while several processes are executed in a concurrent mode.



Quantum size (ms): 50
Delay time (ms): 0
Memory scanning time: 500

Memory Management: [✓ Real Memory] Virtual Memory Swapping

Dispatching: [✓ Queues] 1 Highest Response Ratio Next FIFO Priority Shortest Job First Shortest Remaining Time

Job Placement: [✓ First fit] Best fit Worst fit

Page Replacement: FIRO [✓ LFU] LRU

Figure 8: System Options dialog box

In System Options dialog box (Figure 8) students can modify:

- ***Quantum Size***: system time sharing quantum size in ms
- ***Delay time***: processes speed control
- ***Memory scanning time***: time interval of RAM scanning
- ***Dispatching***: current scheduling strategy
- ***Memory Management***: current memory management strategy – *Real Memory*, *Virtual Paging* or *Swapping*
- ***Job Placement***: jobs placement strategy in real memory mode
- ***Page Replacement***: page replacement strategy in virtual memory mode

These modifications of system parameters allow real-time monitoring of how each of them affects the processes executed.

2.4. TOST System Watch

For achieving an OS inside view, while several processes are executed in parallel mode, students can monitor the information dynamics in selected system tables (Figure 9): ***Processes***, ***Ready***, ***Blocked***, ***Memory Allocation*** map and ***Semaphores*** info. The information from each of the system tables is presented in a separate window, as it is shown in Figure 10.

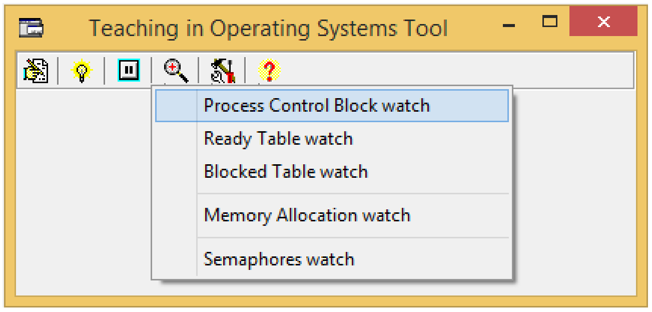
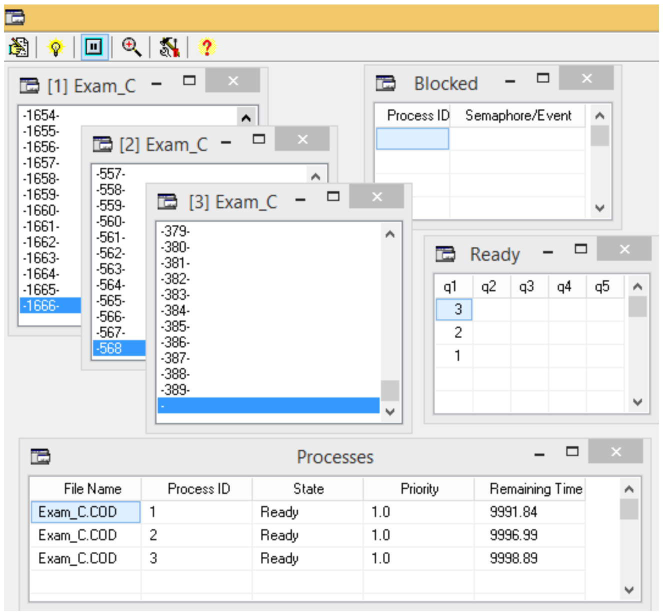


Figure 9: System tables Watch

2.4.1. Processes

The ***Processes*** table (Figure 10) contains the current ***Process Control Block*** of each of the existing concurrent processes.



```
-1654-
-1655-
-1656-
-1657-
-1658-
-1659-
-1660-
-1661-
-1662-
-1663-
-1664-
-1665-
-1666-
```

```
-557-
-558-
-559-
-560-
-561-
-562-
-563-
-564-
-565-
-566-
-567-
-568
```

```
-379-
-380-
-381-
-382-
-383-
-384-
-385-
-386-
-387-
-388-
-389-
-
```

Process ID	Semaphore/Event

q1	q2	q3	q4	q5
3				
2				
1				

File Name	Process ID	State	Priority	Remaining Time
Exam_C.COD	1	Ready	1.0	9991.84
Exam_C.COD	2	Ready	1.0	9996.99
Exam_C.COD	3	Ready	1.0	9998.89

Figure 10: Processes execution and system tables monitoring

The information is presented in five fields:

- ***File Name***: the name of the compiled object file
- ***Process ID***: an integer unique identifier of the process
- ***State***: current process status – *Ready*, *Running* or *Blocked*
- ***Priority***: a real value determining the process priority
- ***Remaining Time***: the time remaining until the process completion.

In this table, students can manipulate the priority or the remaining time of the selected process.

2.4.2. Ready table

The ***Ready*** table contains IDs of all processes with status "Ready". It is possible (according the settings in System Options) to have up to 5 priority queues. Thus, students can experiment with scheduling strategies like ***Round Robin*** (one queue only) or ***Multilevel Feedback Queue*** [18] for the separation of the processes with different behavior (with more I/O operations or with more computations).

2.4.3. Blocked table

The identifiers of blocked processes and the names of semaphores/events, related to their blocking are presented in the ***Blocked*** table. In fact, these are FIFO-queues with blocked processes of all semaphores used. The information in this table is especially helpful when students have to monitor semaphores usage in processes ***synchronization***, ***mutual exclusion*** and ***deadlock*** prevention cases.

2.4.4. Memory Allocation map

Important information about memory allocation and referencing is given by ***Memory Allocation Watch***. When this submenu item is selected, the system periodically scans the TOST RAM and displays results in a separate window. This feature allows students to monitor a map of the memory blocks, where the occupied blocks are marked with the process' ID. The detailed referencing of the pages in the virtual memory is also displayed.

2.4.5. Semaphores information

Semaphores Watch submenu item can be selected for monitoring all semaphore variables' current values in a separate window. This helps students to master the mechanism of using both binary and counting semaphores.

3. Teaching Operating System Basics

3.1. Teaching Process Management

A process [2] is basically a program in execution. During its existence, the process goes through several different states, and the basic three are ***Ready***, ***Running***, and ***Blocked***.

Once students run several processes, they can monitor (as shown in Figure 10) their life cycles in the *Processes* (ID, the current State and the state changing), *Ready* (ID list of all ready processes), and *Blocked* (ID list of all blocked processes, and the event each process is waiting for) tables.

3.2. Teaching Process Scheduling

The act of choosing which of all the ready processes should be moved to the running state is not a trivial task [19]. It is known as Process Scheduling and is performed by OS Dispatcher.

There are several goals of the process scheduling system:

- to maximize the fairness according to the processes priorities
- to keep the CPU busy at all time
- to maximize throughput and to deliver minimum response time for all processes
- to minimize resource starvation
- to support a lot of interactive users
- to minimize system overheads (OS CPU time, OS RAM).

In fact, very often these goals contradict with each other, so the Dispatcher should implement an appropriate compromise [20]. Through experimentation, students are expected to gain understanding in the scheduling strategies and to be able to choose the most appropriate one, depending on the user's needs and objectives. In TOST students can change (at any time) the current scheduling strategy using ***Dispatching*** radio-buttons in ***System Options*** (see Figure 8).

Scheduling disciplines of the two general categories are supported in TOST:

Non-Preemptive Scheduling: once the OS assigns the CPU to a process, the process does not release the CPU until it has finished:

- ***FIFO:*** Simplest scheduling algorithm with only one queue (see queue q1 in *Ready*). New processes are appended in q1 where context switches only occur upon running process' termination. Then the top placed process in q1 becomes *Running*. Students can monitor:
  - scheduling overheads are minimal
  - throughput can be low
  - no starvation
  - waiting time and response time can be high
  - no prioritization occurs
  - interactive users are not allowed
- ***Priority:*** Whenever a scheduling event occurs, the *Processes* table will be searched for the process with the maximum priority number (the most important one), which will be the next one to be scheduled for execution. Students can monitor:
  - scheduling overheads are low
  - throughput can be low
  - starvation can occur in a busy system with many high-priority processes in the Ready table

- waiting time and response time can be high
- interactive users are not allowed

- ***Shortest Job First (SJF)***: Dispatcher selects the process with the least estimated processing time to be executed. The estimated processing time in TOST can be set while process is initialized (Figure 7). Students can monitor how the algorithm increases throughput, and how the starvation can become a problem, in a system with many short processes in the *Ready* table.
- ***Highest Response Ratio Next (HRRN):*** is a dynamic priority discipline proposed by Brinch Hansen to mitigate the problem of the large processes starvation. When CPU becomes free, Dispatcher calculates the priorities of all ready processes according to:

  Priority = 1 + waiting time / processing time

  and selects the high priority process to be executed. Students can monitor the priority calculations (in *Processes* table) which gives preference to short processes, but with each subsequent recalculation the priority of the large ones grows and they can be selected as well.

Preemptive Scheduling: Dispatcher can interrupt a running process any time in middle of the execution and the process is returned back in the *Ready* table:

- ***Shortest Remaining Time (SRT):*** is a modification of SJF with a preemption. When a new shortest process arrives Dispatcher interrupts the current process and assigns the CPU to the new one. Students can monitor that this discipline achieves maximum throughput in most cases, and the starvation is possible for large processes where a lot of small processes arrive.
- ***Queues:*** By specifying the number of queues (Figure 8), students can set two strategies:
  Queues (1): Round-robin (RR): is a FIFO modification with the process preempting where *Ready* table is a queue. Dispatcher assigns a small time interval (Quantum) per process, then interrupts and appends it at the end of the *Ready* queue. When students set *Delay time* in *System Settings* they can monitor processes preempting in the *Ready* table in detail. When students run processes with different priorities, they can monitor how the more important processes are with proportionally larger quantum and get a better service, and the starvation can never occur.
  Queues (2-5): Multilevel feedback queues: In this case, processes are grouped according to their behavior - interactive processes vs batch processes, and processes with more I/O vs. computational processes. In order to obtain a balance in system's resources utilization Dispatcher has to provide better service for interactive and I/O processes. Students can monitor in detail how processes pass into the multilevel set of priority queues (from 1 to 5) in *Ready* table.

3.3. Teaching Memory Management

In general, when discussing the topic of memory management [2], [19] the focus is on the virtual memory organization. However, before reaching this point, students need to understand in detail what the issues are in using the real memory organization. Only then they can comprehend why this kind of organization has been replaced everywhere by the virtual one.

3.3.1. Real Memory

Real memory organization is a classical (one of the oldest) memory allocation model according to which the process is placed in a consecutive memory blocks [21]. Each process takes up exactly as much memory as it needs, and due to the fact that the memory blocks have sequential addresses, process loading, executing and the memory releasing are extremely fast. Then *"Where is the problem?"* comes to be the next instructor's question towards the students.

Students are instructed to develop and try to run several large processes. Later, they run five small processes and with using the TOST feature ***Memory Allocation Watch,*** monitor the memory map with real memory organization. Then students kill the first, second and fourth process, so these memory blocks are freed. Figure 11 shows how the memory blocks occupied by a process are marked with its identifier.

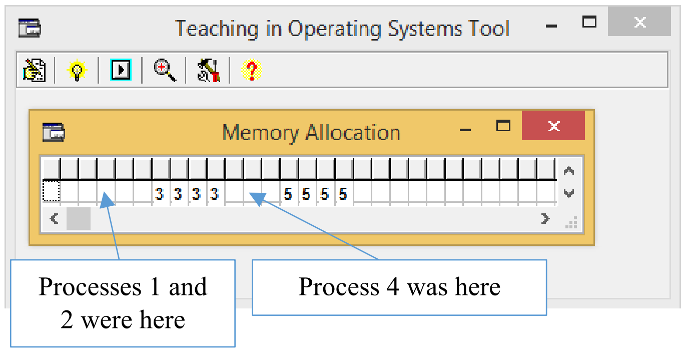

Teaching in Operating Systems Tool
Memory Allocation
3 3 3 3
5 5 5 5
Processes 1 and 2 were here
Process 4 was here

Figure 11: Memory allocation map

Following these instructions, students quickly realize the deficiencies of the real memory:

- A process cannot be started if it is even one byte larger than the available physical memory.
- There is a low CPU utilization, due to the low degree of multiprogramming (number of ready processes loaded in physical memory).
- The occurrence of memory fragmentation: In high dynamics of started and finished processes, the free memory area is broken into small parts, in which none of the ready processes can be allocated. Thus, a lot of memory blocks can remain unusable.

In order to minimize the problem of memory fragmentation, the placement strategies ***First-fit***, ***Best-fit*** and ***Worst-fit*** can be applied (Figure 8). To assess their usefulness students can run a new process and monitor its placement in the memory.

3.3.2. Virtual Memory

Although the problem of memory fragmentation can be solved, other serious problems of real memory remain: small maximum process size and low degree of multiprogramming.

Virtual memory organization allows each process to be divided into blocks (***pages***) while only a small part of the code and data (that are currently needed) are located in the physical memory. The physical memory is divided into blocks, which are called ***frames***. Figure 12 illustrates a moment in the execution of a large process where ***Virtual Memory*** as *Memory management* is set (Figure 8).

At that stage, students can monitor the occupation and references to the blocks (pages) in progress, with the memory scanned at a certain interval (0.5s for example). When the process starts, only a small part of its blocks are loaded - those that are needed at the beginning. Later, the OS loads additional blocks into the memory, those which are needed by the process but are not available. The blocks that the process refers to during the scan interval are colored red.

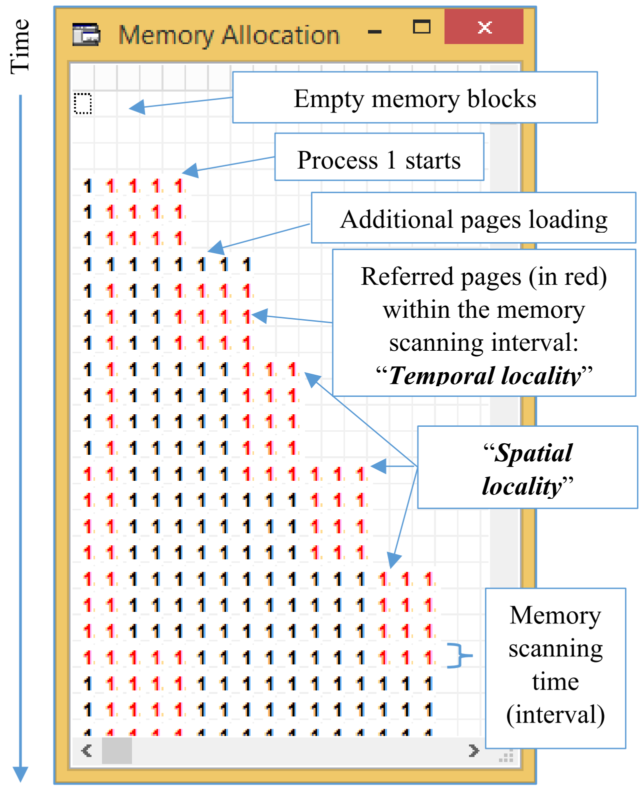

Time
Memory Allocation
Empty memory blocks
Process 1 starts
Additional pages loading
Referred pages (in red) within the memory scanning interval: "Temporal locality"
"Spatial locality"
Memory scanning time (interval)

Figure 12: Memory allocation and pages referencing

Memory allocation map allows the instructor to clarify the ***principles of the locality*** [22] - ***Temporal locality*** and ***Spatial locality***.

Principles of the locality can be used for anticipatory paging (swap prefetch) to increase processes' execution speed. These principles are useful when the optimal replacement strategy has to be chosen.

In TOST students can set (Figure 8) and examine various replacement strategies such as *Least Recently Used* (***LRU***), *Least Frequently Used* (***LFU***) and ***FIFO***.

4. Teaching Concurrent Programming Concepts

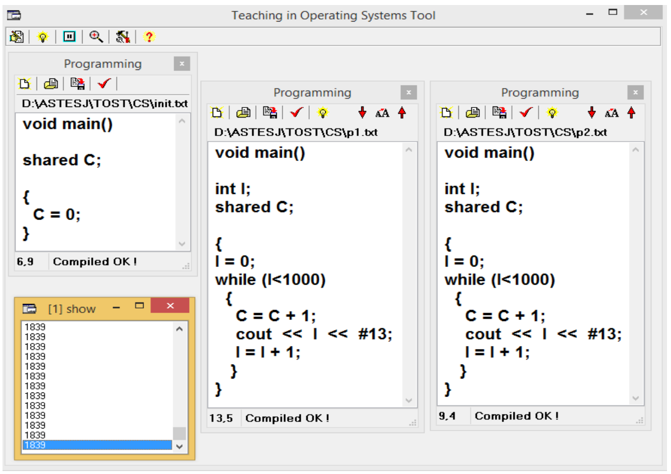
Even if the students are familiar with the theoretical aspects of parallel programming, in order to gain intuition about the subject, it is vital to be provided with examples of incorrect operation of competitive processes. Therefore, our approach is to firstly develop concurrent programs without any protections and then, analyze the faulty actions of the processes. Consequently, modify the programs in order to achieve correct results. In addition, to strengthen this knowledge, we solve and program some of the well-known toy-problems in concurrent processing.

Representative examples of using TOST in teaching ***Mutual Exclusion***, ***Interprocess Synchronization*** and ***Deadlock*** are considered in the next sections.

4.1. Teaching Mutual Exclusion

The parts of the concurrent processes where they access a shared resource are called ***Critical Sections*** (CS) [23]. Critical Sections must be protected by synchronization primitives, assuring only one of the processes can be in the CS.

Figure 13 presents programs with no protection of the critical section. Students develop and execute an initialization process (init.txt), and two concurrent processes (p1.txt and p2.txt), incrementing the shared variable C 1000 times. After finishing the processes, the students can see the final result value of C = 1839, instead of the correct value of 2000.



D:\ASTESJ\TOST\CS\init.txt
```
void main()

shared C;

{
  C = 0;
}
```
6,9 Compiled OK !

D:\ASTESJ\TOST\CS\p1.txt
```
void main()

int I;
shared C;

{
I = 0;
while (I<1000)
  {
    C = C + 1;
    cout  <<  I  <<  #13;
    I = I + 1;
  }
}
```
13,5 Compiled OK !

D:\ASTESJ\TOST\CS\p2.txt
```
void main()

int I;
shared C;

{
I = 0;
while (I<1000)
  {
    C = C + 1;
    cout  <<  I  <<  #13;
    I = I + 1;
  }
}
```
9,4 Compiled OK !

```
1839
1839
1839
1839
1839
1839
1839
1839
1839
1839
1839
1839
1839
```

Figure 13: Concurrent processes with Critical Section

After examining the reasons for the wrong result, the students transform the programs gradually, seeking the correct solution for synchronization primitives. Thus, they examine program versions with one, two, and three shared variables, and at last, they come to the consideration of the ***Dekker's algorithm*** (Figure 14).

The ***Peterson's algorithm*** [24] is another well-known algorithm that students can try and compare with Dekker's algorithm. A simple solution of CS protection by the Peterson's algorithm is given in Figure 15.

In Figure 16 a simple example of CS protection, using primitives ***Lock*** and ***Unlock***, based on the hardware instruction ***Test_And_Set***, is presented.

The last example considered is the use of semaphores to protect the critical section (Figure 17).

```
void main()
shared C, ReqP1, ReqP2, Turn ;
{
 C = 0;
 ReqP1 = false;
 ReqP1 = false;
 Turn = 1;
}
```

```
void main()
int I;
shared C, ReqP1, ReqP2, Turn ;
{
I = 0;
while ( I < 1000 )
  {
   ReqP1 = true;
   while ( ReqP2 == true )
    If  (Turn  == 2)
     {
      ReqP1 = false;
        while (Turn  == 2)  ;
      ReqP1 = true;
     };
         C = C + 1;
   ReqP1 = false;
   Turn  = 2;
   I = I + 1;
  }
}
```

(a) Initialization process (b) Process 1

Figure 14: Critical Section protection by Dekker's algorithm

```
void main()
shared C, ReqP1, ReqP2, Turn;
{
 C = 0;
 ReqP1 = false;
 ReqP1 = false;
}
```

```
void main()
int I;
shared C, ReqP1, ReqP2, Turn ;
{
I = 0;
while ( I < 1000 )
  {
   ReqP1 = true;
   Turn  = 2;
   while (ReqP2 and (Turn == 2)) ;
         C = C + 1;
   ReqP1 = false;
   I = I + 1;
  }
}
```

(a) Initialization process (b) Process 1

Figure 15: Critical Section protection by Peterson's algorithm

```
void main()
shared C, CS;
{
 C = 0;
 CS = false;
}
```

```
void main()
int I;
shared C, CS ;
{
I = 0;
while ( I < 1000 )
  {
   Lock (CS);
         C = C + 1;
   Unlock (CS);
   I = I + 1;
  }
}
```

(a) Initialization process (b) Process 1

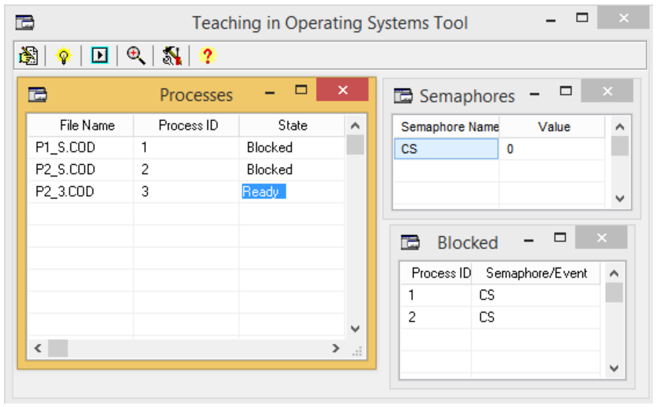
Figure 16: Critical Section protection by Test_And_Set instruction

```
void main()
semaphore  CS;
shared  C;
{
 C = 0;
 init (CS, 1);
}
```

```
void main()
semaphore  CS;
shared  C;
int  I;
{
 I = 0;
 while (I<1000)
  {
   wait (CS);
      C = C + 1;
   signal (CS);
   I = I + 1;
  }
}
```

Figure 17: Critical Section protection by a semaphore

Figure 18 presents the contents of two main OS tables (Processes and Blocked) during the concurrent execution of three identical processes with the source code of Figure 17. Processes *P1_S* and *P2_S* are blocked, while the ready process *P2_3* is inside the critical section. Students also have the possibility of monitoring the semaphore variable CS in a separate window



File Name	Process ID	State
P1_S.COD	1	Blocked
P2_S.COD	2	Blocked
P2_3.COD	3	Ready

Semaphore Name	Value
CS	0

Process ID	Semaphore/Event
1	CS
2	CS

Figure 18: Monitoring TOST system tables

After finishing the processes, the value of the shared variable C is correct and equal to 3000.

4.2. Teaching Interprocess Synchronization

A good illustration of the Interprocess Synchronization is the. ***Sleeping Barber problem*** [25] depicted in Figure 19. As soon as the Barber or Client process reaches a certain point in the execution, it must stop and wait for an event to occur.

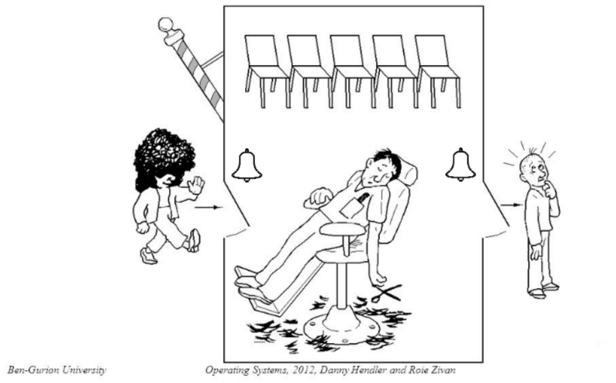


Figure 19: Sleeping Barber problem

The problem can be stated as follows: In a barbershop, there is a barber chair and an area for the waiting clients with five chairs. The algorithms of the concurrent processes of the Barber and the Client include the following steps:

Barber:

1) *The barber signals that he is ready to work (free).*
2) *The barber sits on the barber's chair and sleeps, waiting for a new client (indoor bell).*
3) *The barber gets up and waits for the client to sit on the barber's chair.*
4) *The barber begins the haircut and signals when he completes the work.*
5) *The barber waits until the client leaves the barbershop (outdoor bell).*
6) *Go to 1.*

```
void main()
semaphore Chairs, Barber,
            Client, Sit, Finish,
            Out;
{
   init ( Chairs, 5 );      There are 5 chairs available
   init ( Barber, 0 );      The barber is not available
   init ( Client, 0 );      No new client
   init ( Sit, 0 );         No one is sitting in barber's chair
   init ( Finish, 0 );      The barber has not finished work
   init ( Out, 0 );         The client has not come out
}
```

(a) Initialization

```
void main()
semaphore Chairs, Barber,
            Client, Sit, Finish,
            Out;
{
   while (true)
   {
    signal ( Barber );
    wait ( Client );
    wait ( Sit );
    signal (Finish );
    wait ( Out );
   }
}
```

(b) Barber

```
void main()
semaphore Chairs, Barber,
            Client, Sit, Finish,
            Out;
{
   wait ( Chairs );
   signal ( Client );
   wait ( Barber );
   signal ( Chairs );
   signal ( Sit );
   wait ( Finish );
   signal (Out );
}
```

(c) Client

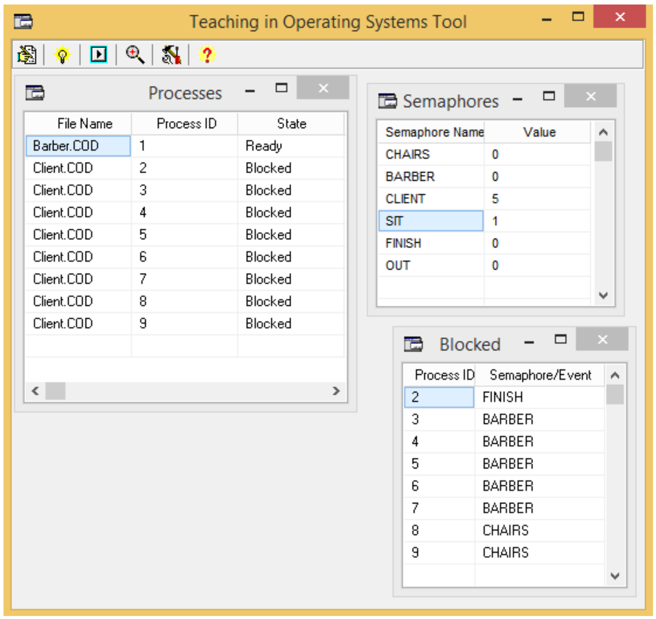
Figure 20: Sleeping Barber solution

Client:

1) *The client checks if there is a free chair in the waiting room and eventually waits until a chair is available.*
2) *The client signals that there is a new client (input bell).*
3) *The client is waiting for the barber to be free.*
4) *The client frees the chair in the waiting room.*
5) *The client sits on the barber's chair and signals that he is ready for a haircut.*
6) *The client is waiting for the barber to complete the haircut.*
7) *The client gets out of the barber's chair and leaves the barbershop (output bell).*

The first thing students have to do is to identify processes synchronization events and implement one semaphore variable for each of the controlled events. Then they develop three programs - Initialization (for setting semaphore values), Barber, and Client (Figure 20).

The implementation of the synchronization of one barber's process and eight clients' processes are depicted in Figure 21.



File Name	Process ID	State
Barber.COD	1	Ready
Client.COD	2	Blocked
Client.COD	3	Blocked
Client.COD	4	Blocked
Client.COD	5	Blocked
Client.COD	6	Blocked
Client.COD	7	Blocked
Client.COD	8	Blocked
Client.COD	9	Blocked

Semaphore Name	Value
CHAIRS	0
BARBER	0
CLIENT	5
SIT	1
FINISH	0
OUT	0

Process ID	Semaphore/Event
2	FINISH
3	BARBER
4	BARBER
5	BARBER
6	BARBER
7	BARBER
8	CHAIRS
9	CHAIRS

Figure 21: Sleeping Barber execution monitoring

According to the information in TOST system tables:

- the barber is busy (Semaphores: *Barber = 0*)
- a client is seated in the barber chair (Semaphores: *Sit = 1*) and waiting for the barber to complete the haircut (Blocked: *Finish*)
- five new clients are in the waiting room (Semaphores: *Clients = 5*) and all the chairs are occupied (Semaphores: *Chairs = 0*)
- all five new clients are waiting for the barber (Blocked: *Barber*)
- two clients are waiting for a chair outside of the barbershop (Blocked: *Chairs*).

4.3. Teaching Deadlock

The ***Dining Philosophers*** toy-example [26], depicted in Figure 22, is often used to analyze the synchronization and deadlock problems in the concurrent processes execution, as well as to demonstrate the approaches for solving them.

The students solve this problem by developing five programs (for the philosophers) while the correct access (mutual exclusion) to the shared objects (forks) is implemented by semaphores. The example programs for the first two philosophers are presented in Figure 23.

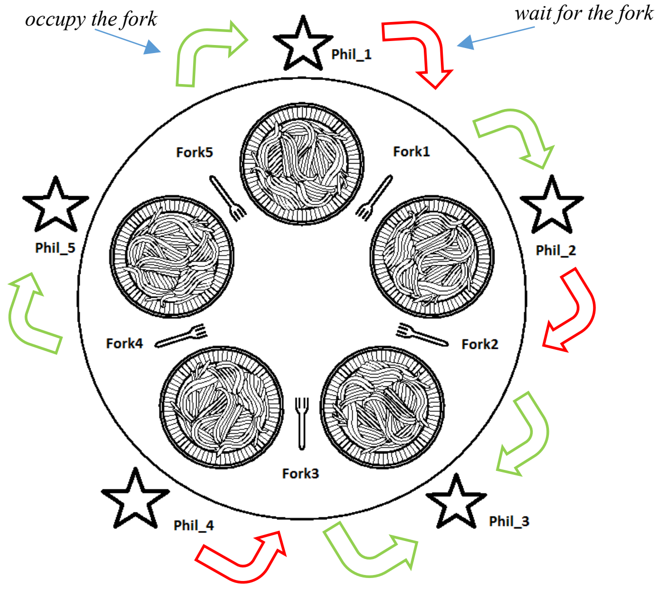
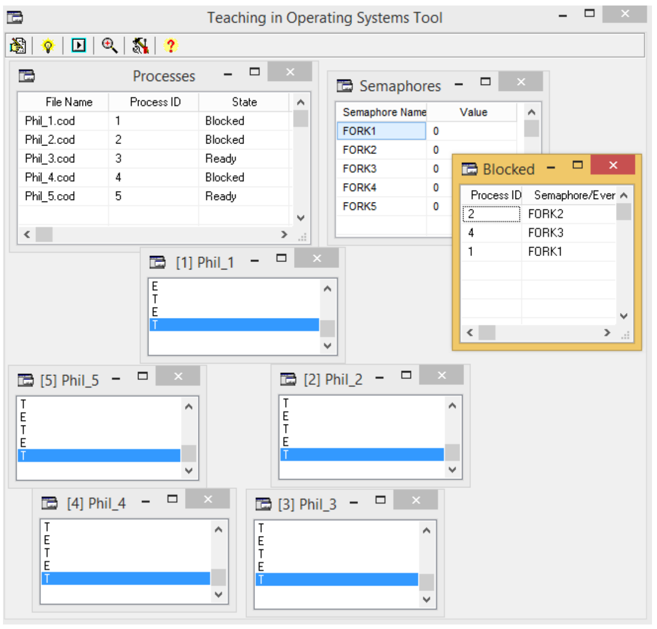


Figure 22: Dining Philosophers problem

```
void main()
semaphore FORK5, FORK1 ;
{
while ( true )
  {
      cout << "T" << #13;
     wait ( FORK5 );
     wait ( FORK1 );
      cout << "E" << #13;
     signal ( FORK1 );
     signal ( FORK5 );
  }
}
```

```
void main()
semaphore FORK1, FORK2 ;
{
while ( true )
  {
      cout << "T" << #13;
     wait ( FORK1 );
     wait ( FORK2 );
      cout << "E" << #13;
     signal ( FORK1 );
     signal ( FORK2 );
  }
}
```

Figure 23: Phil_1 and Phil_2 program samples



Teaching in Operating Systems Tool

Processes

File Name	Process ID	State
Phil_1.cod	1	Blocked
Phil_2.cod	2	Blocked
Phil_3.cod	3	Ready
Phil_4.cod	4	Blocked
Phil_5.cod	5	Ready

Semaphores

Semaphore Name	Value
FORK1	0
FORK2	0
FORK3	0
FORK4	0
FORK5	0

Blocked

Process ID	Semaphore/Ever
2	FORK2
4	FORK3
1	FORK1

[1] Phil_1
```
E
T
E
T
```

[5] Phil_5
```
T
E
T
E
T
```

[2] Phil_2
```
T
E
T
E
T
```

[4] Phil_4
```
T
E
T
E
T
```

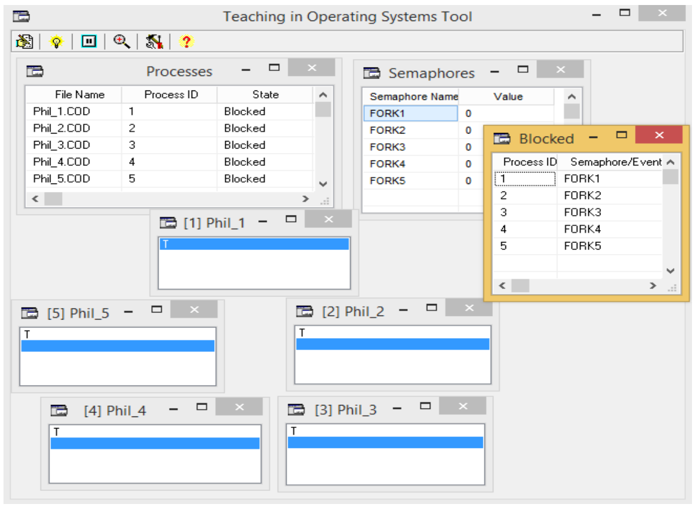
[3] Phil_3
```
T
E
T
E
T
```

Figure 24: Dining Philosophers execution

TOST allows students to monitor (Figure 24) how philosophers are waiting for the forks (in *Processes* and *Blocked*) and how they take and put the forks on the table (in *Semaphores*).

Another aim of the Dining Philosophers problem is to demonstrate the prevention of ***Deadlock***.

Students can monitor how in the case of all philosophers taking a fork with their right hand, while expecting the release of a second fork, a *Deadlock* occurs (Figure 25 and Figure 26).



Teaching in Operating Systems Tool

Processes

File Name	Process ID	State
Phil_1.COD	1	Blocked
Phil_2.COD	2	Blocked
Phil_3.COD	3	Blocked
Phil_4.COD	4	Blocked
Phil_5.COD	5	Blocked

Semaphores

Semaphore Name	Value
FORK1	0
FORK2	0
FORK3	0
FORK4	0
FORK5	0

Blocked

Process ID	Semaphore/Event
1	FORK1
2	FORK2
3	FORK3
4	FORK4
5	FORK5

[1] Phil_1
```
T
```

[5] Phil_5
```
T
```

[2] Phil_2
```
T
```

[4] Phil_4
```
T
```

[3] Phil_3
```
T
```

Figure 25: Dining Philosophers Deadlock example

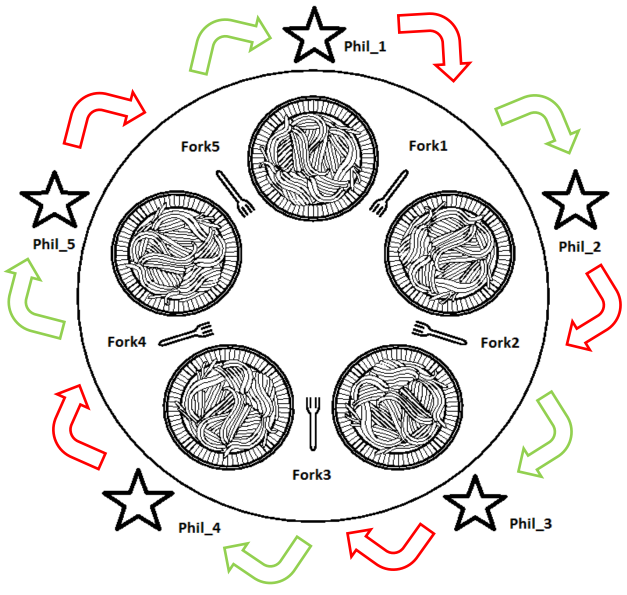


Figure 26: Dining Philosophers Deadlock example (Circular Wait)

Students have to find a solution and save philosophers from starvation dead. One idea is to break the fourth necessary condition for Deadlock - ***Circular Wait*** [27]. Students apply this approach by reprogramming Phil_1 (Figure 27) and introducing the general requirement for all processes to request resources in **ascending order only**.

```
void main()
semaphore FORK5, FORK1 ;
{
while ( true )
  {
     cout << "T" << #13;
     wait ( FORK5 );
     wait ( FORK1 );
      cout << "E" << #13;
     signal ( FORK1 );
     signal ( FORK5 );
  }
}
```

```
void main()
semaphore FORK5, FORK1 ;
{
while ( true )
  {
     cout << "T" << #13;
     wait ( FORK1 );
     wait ( FORK5 );
      cout << "E" << #13;
     signal ( FORK1 );
     signal ( FORK5 );
  }
}
```

Figure 27: Phil_1 program – **before** and **after** changes

5. Students' Opinion

Trying to improve the quality and usefulness of the integrated environment TOST, moreover to get better results from its usage, we also rely on end-users' opinions. For this reason, in a span of two consecutive years, we conduct qualitative surveys with students at the end of the course. The survey used a 5-point Likert scale and is based on a relatively small sample of 131 bachelor students. The summary data for user acceptance of the presented tool are in Table 1.

Table 1: Opinion Survey Results

Survey questions	**A**	**B**	**C**	**D**	**E**
I like TOST environment	2	13	30	55	31
TOST is easy to learn	0	3	15	40	73
I think that programming in TOST is complex	75	29	11	12	4
TOST is visually very attractive	7	22	26	61	15
Overall, I am satisfied with the TOST approach to studying Operating systems	5	11	15	41	59
Overall, I am satisfied with the performance of the TOST environment	9	9	12	21	80
The installation process of TOST is quick and easy	0	0	3	5	123
TOST often freeze, crash, or does not behave as expected	106	9	11	4	1
Overall, I am satisfied with the experimenting with TOST example solutions in understanding concepts related to Mutual Exclusion, Synchronization, Deadlock, and Operating system in general	4	2	3	111	11
I think the interface of the TOST environment is user-friendly	8	5	39	57	22
TOST is successful and effective in stimulating Operating system algorithms and strategies	0	0	0	95	36

A: *strongly disagree*
B: *disagree*
C: *undecided*
D: *agree*
E: *strongly agree*

Based on the data presented, it can be assumed that the responses are positive. According to the majority, the TOST environment is easy to learn and use, user-friendly, useful, and supportive for their understanding of the fundamental concepts of operating systems and the problems concerning concurrent programming.

6. Conclusions and Further Research

Learning the concept and basic principles of Operating systems poses a great challenge to Computer Systems and Technologies students. In this paper, the authors present an integrated environment named TOST, specially designed to be used in Operating systems and Concurrent programming courses at Ruse University. A description of the TOST environment and a demonstration of how it can be very effectively used in teaching Operating systems principles are discussed, as well as some of the TOST features that address the learning of Mutual Exclusion, Interprocess Synchronization, and Deadlock.

The TOST includes a time-sharing operating system with compilers for two simple and easy to use programming languages. There are shared variables and semaphores variables for concurrent algorithms development. The ability to monitor the real-time information from basic OS tables allow students to gain a better apprehending of the algorithms, concepts, and theories behind the design, construction, and insights of operating systems.

The presented environment might be extended in the following directions:

- developing a built-in file system with a set of file allocation methods and management strategies;
- developing a spooling system for mediation between a process and slow peripheral devices;
- implementing complex data types and subroutines for the programming languages integrated.

We have been using TOST for several years in the Department of Computer Systems and Technologies at Ruse University and the student's feedback and results are positive. The lack of any installation procedure, the small size, and the 'copyleft' license allow TOST to be obtained and used without limitation and with full inclusivity.

Conflict of Interest

The authors declare no conflict of interest.

Acknowledgments

This work/paper is supported by project 20-FEEA-01 "Methods and tools for multimedia content analysis and automated document and big data processing", funded by the Research Fund of the "Angel Kanchev" University of Ruse.

References

[1] T. Golemanov, E. Golemanova, "Using TOST in Teaching Mutual Exclusion, Synchronization, and Deadlock," in 29th Annual Conference of the European Association for Education in Electrical and Information Engineering (EAEEIE) 2019, Ruse, Bulgaria, 2019, doi:10.1109/EAEEIE46886.2019.9000441.
[2] A.S. Tanenbaum, A.S. Woodhull, Operating Systems Design and Implementation (Second ed.), 2011.
[3] W. Christopher, and T.A. S. Procter, "The Nachos instructional operating

system," in 1993 Winter USENIX Conference, San Diego, California, 1993, doi:10.5555/1267303.1267307.

[4] D. Comer, Operating System Design: The XINU Approach, Prentice-Hall, 1984.

[5] B. Pfaff, G. Romano A. and Back, "The Pintos Instructional Operating System Kernel," in SIGCSE 2009, Chattanooga, TN, 2009, doi:10.1145/1539024.1509023.

[6] D. Hovemeyer, J.K. Hollingsworth, B. Bhattacharjee, "Running on the bare metal with GeekOS," ACM SIGCSE Bulletin, **36**(1), 2004, doi:10.1145/1028174.

[7] L.P. Maia, F.B. Machado, P.J.A. C., "A Constructivist Framework for Operating Systems Education: a Pedagogic Proposal Using the SOsim," in ITiCSE'05, Monte de Caparica, Portugal, 2005, doi:10.1145/1067445.1067505.

[8] T.W. Chan, "A Software Tool in Java for Teaching CPU Scheduling," Journal of Computing Sciences in Colleges, **19**(4), 257–263, 2004, doi:10.5555/1050231.1050269.

[9] D. Skrien, "CPU Sim 3.1: A tool for simulating computer architectures for computer organization classes," Journal on Educational Resources in Computing (JERIC), **1**(4), 2001, doi:10.1145/514144.514731.

[10] D. Park, R.H. Saavedra, "Trojan: A high-performance simulator for shared memory architectures," in Proceedings of the 29th Annual Simulation Symposium. IEEE, 1996, doi:10.5555/829528.831210.

[11] A.R. Lopes, D.A. de Souza, J.R.. de Carvalho, W.O. de Silva, S.V.L. P., "SIME: Memory simulator for the teaching of operating systems," in Computers in Education (SIIE), 2012 International Symposium on. IEEE, 2012.

[12] P. Gayet, B. Bradu, "Procos: A real-time process simulator coupled to the control system," in Proceedings of ICALEPCS, Kobe, Japan: 794–796, 2009.

[13] A. Garmpis, "Design and development of a web-based interactive software tool for teaching operating systems," Journal of Information Technology Education, **10**(10), 1–17, 2011, doi:10.28945/1357.

[14] B. Mustafa, "Visualizing the modern operating system: simulation experiments supporting enhanced learning," in Proceedings of the 2011 Conference on Information Technology Education. ACM, 209–214, 2011, doi:10.1145/2047594.2047650.

[15] T. Golemanov, E. Golemanova, "A Teaching in Operating Systems Tool," in CompSysTech'06, 2006.

[16] E.W. Dijkstra, "Cooperating sequential processes (EWD-123)," Center for American History, University of Texas at Austin, 1965, doi:10.5555/1102034.

[17] M. Herlihy, "Wait-free synchronization," ACM Trans. Program. Lang. Syst. **13**(1), 1991, doi:10.1145/114005.102808.

[18] A.C. Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Operating Systems: Three Easy Pieces, Arpaci-Dusseau Books, 2014.

[19] A. Silberschatz, P.B. Galvin, G. Gagne, Operating System Concepts. 9., John Wiley & Sons, Inc., 2013.

[20] P. Krzyzanowski, Process Scheduling: Who gets to run next?, 2015.

[21] D. Samanta, Classic Data Structures, Prentice Hall India Pvt, 2004.

[22] S. William., Computer organization and architecture : designing for performance (8th ed.), Prentice Hall, Upper Saddle River, NJ, 2010.

[23] M. Raynal, Concurrent Programming: Algorithms, Principles, and Foundations, Springer Science & Business Media, 2012.

[24] G.L. Peterson, "Myths About the Mutual Exclusion Problem," Information Processing Letters, **12(3)**, 115–116, 1981, doi:10.1016/0020-0190(81)90106-X.

[25] H.M. Deitel, P.J. Deitel, D.R. Choffnes, Operating systems, Pearson/Prentice Hall, 2004.

[26] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall International, 2004.

[27] J. Coffman, Edward G., M.J. Elphick, A. Shoshani, "System Deadlocks," ACM Computing Surveys, 1971, doi:10.1145/356586.356588.