Teaching Real Time OSs with DORITOS

Jae C. Oh and Daniel Mossé
Department of Computer Science
University of Pittsburgh, Pittsburgh, PA 15260.
{joh, mosse}@cs.pitt.edu
http://www.doritos.cs.pitt.edu

Funded by National Science Foundation, Grant Number DUE-9652953

Abstract

We are developing a teaching package that can be used in a college course that would fill a gap among current science majors and teach senior-level undergraduate students theory and practice of real-time operating systems, including their requirements, characteristics, internals, and specification. This course has two components: (1) a theoretical part, and (2) a practical hands-on implementation component achieved with DORITOS (Distributed Object-Based Real-time InsTructional Operating System) as the implementation environment. DORITOS' design is based on UC-Berkeley's NACHOS. The DORITOS package will be distributed with DKaffe (a modified version of Kaffe JVM) and a basic system which allows students to run simple threads.

In this paper, we focus on the practical, hands-on system that allows students to learn the internals of a Real-time Operating Systems (RTOS). Throughout the term, assignments require students to use and modify DORITOS to implement real-time elements as well as to analyze the performance of implemented algorithms.

1 Introduction

Computer operating system (OS) courses need to teach students both theory and practice. The theory of operating systems can be taught using textbooks and other publications, while teaching the practical aspects of operating systems requires hand-on implementations of the concepts being taught. Several instructional systems (e.g., MINIX [1], NACHOS [3], and Tunis [4]) attempt to provide practice. The above systems can be used to supplement undergraduate operating systems courses that teach fundamental concepts of computer operating systems. Although supplements to "regular" operating system courses exist, there are no such counterparts for teaching real-time OS courses.

We are developing a teaching package that can be used in a college course that would fill a gap among current computer science curricula and teach senior-level undergraduate and graduate students theory and practice of real-time operating systems (RTOS); these include requirements, characteristics, internals, and specification of such systems. This course has two components: (1) a theoretical part, and (2) a practical hands-on implementation component achieved with DORITOS (Distributed Object-Based Real-time InsTructional Operating System).

DORITOS' design is based on UC-Berkeley's NACHOS in the following ways:

- both systems use Object-Oriented (OO) design,
- the syntax for the invocation of DORITOS is similar,
- the code structure is similar, and
- both systems contain a MIPS emulator to support execution of user programs.

DORITOS package comes with a DORITOS basic system which allows students to run simple threads. Throughout the term, assignments will require students to use and modify DORITOS to implement real-time elements as well as to analyze the performance of implemented algorithms.

A RTOS is one that considers finishing a task on time as important as finishing it correctly. Examples of such systems are pervasive in industrial process control and space applications. Consider, for instance, a process submitted to an OS during the landing of a space shuttle. If the process is delayed due to background tasks and the parachutes or retro-rockets are not activated, a catastrophe may ensue. Therefore, RT tasks/processes MUST be completed correctly and timely. These concepts are becoming even more popular as of late, with the insertion of such technology in run-of-the-mill products such as anti-lock break systems and cruise control systems in modern automobiles.

This paper is organized as follows: Section 2 presents an overview of DORITOS, including (1) design decisions made, (2) timing mechanisms developed to support real-time aspect of the operating system, and (3) changes made to a standard, freely-distributed Java virtual machine – Kaffe [9]. We call the modified virtual machine *DORITOS Kaffe (DKaffe)*. Section 3 discusses student assignments that accompany DORITOS. Section 4 discusses the project status. Section 5 presents our classroom experience and future work.

2 DORITOS Overview

2.1 Developing DORITOS

Our positive experience with NACHOS heavily inspired us in the design and implementation of DORITOS. Therefore, DORITOS shares some design decisions, such as OO design, with NACHOS. However, we chose Java as language for developing DORITOS while NACHOS is implemented with C++. There are pedagogical as well as practical reasons for choosing Java:

- *Portability of DORITOS:* Java is platform independent. Therefore, students in various institutions will have less problems in installing and using DORITOS in fast changing computing environment.
- *Accessibility:* There exist free Java Virtual Machines (JVM) – for example, Kaffe by T.J. Wilkins & Assoc. [9] and Sun Microsystem's Java Development Kit (JDK) [6] for educational institutions – with source code that can be modified; it was necessary to modify the JVM to support mechanisms of DORITOS. Kaffe comes with "Berkeley-type license" so that it can be modified and can be distributed freely.
- *Object-Oriented Nature of Java:* Java supports strongly-typed language and makes it easy to reuse software components. Students using DORITOS will utilize their time more wisely, with less debugging of unnecessary mistakes and, therefore, spend more time in learning principles of real-time systems [7]. In particular, our experiences from teaching operating systems courses in the past five years have shown that students struggle more with the correct usage of the language constructs rather than with the operating system concepts being taught. In particular, our experiences reveal that the most common problem is the misuse of pointer types. Java intentionally does not support pointer types so that it eliminates programmers' inadvertent misuse of pointers.
- *Better Network Support of Java:* Network classes and libraries are well supported in Java and easier to use than in other conventional languages. Future DORITOS assignments that implement a RT network will take advantage of the network capabilities of Java.
- *Popularity and widespread use of Java:* Supporting evidences include: (1) JVMs have been ported to most OSs. (2) popular Internet browsers such as Netscape and MicroSoft Internet Explorer support *Java scripts* and *Applets*, and (3) in the JavaOne conference – a well-known Java conference – the attendance has been steadily increased from 6,000, 10,000, and 40,000 in the years 1996, 1997, and 1998, respectively.

2.2 Components in DORITOS System

Figure 1 shows basic DORITOS components: a JVM (DKaffe), DORITOS Kernel, and a device (DISK in this case) thread. DORITOS Kernel is further divided into DISK Handler thread and DORITOS Kernel thread.

DKaffe is the modified Java Virtual Machine for the DORITOS system, the DISK thread emulates the hardware of hard disk, and DORITOS Kernel emulates the operating system. The DISK handler thread is a part of the DORITOS Kernel that handles disk I/O requests and interacts with DISK thread and DORITOS Kernel thread; it is similar to a DISK device driver. There can be many other device threads and device handler threads such as timer thread, timer device thread, network thread, network device thread, etc.; however, for illustration purposes, we'll only show the DISK and its handler.

In the figure, the DORITOS Kernel thread shows a few main components of the kernel: Disk Request Queue, Scheduler, and Ready Queue; the small circles in Disk Request Queue and Ready Queue represent DORITOS user threads. The arrows in the figure are used to explain the dynamics of a disk I/O request (see Section 2.3).

Note that the DISK thread, DISK Handler thread, and DORITOS Kernel thread are Java threads (i.e., they run on top of DKaffe), while DORITOS user threads are emulated threads within DORITOS Kernel thread; they are user programs that runs by the MIPS emulator (not shown in the figure), which is a part of DORITOS Kernel thread.

2.3 A Reliable and Precise Timing Mechanism

An important feature in DORITOS is the timing mechanism. The main purpose of real-time systems is to guarantee that each job is completed before its deadline. If there is no precise timing mechanism, real-time scheduling does not mean much. UNIX time measures such as `gettimeofday()`, `time()`, etc., are not precise with respect to the JVM, and therefore cannot be used for counting time in DORITOS. This is because the JVM is but a process running on top of an OS and may be of varying priority. We needed an emulated time that is precise, consistent and self-contained within the JVM. In such case, DORITOS time will not be affected by external conditions such as clock drift and changes in CPU loads in the host machine.

DORITOS's time tick (or unit time) is one Java byte-code instruction executed by DKaffe. The actual time (that is, the real world time) it takes to execute each byte-code instruction is irrelevant in DORITOS; as far as DORITOS is concerned, one-byte code executed is equivalent to one time tick.[1] For each Java instruction executed by DKaffe, one DORITOS unit time is passed by incrementing a global integer variable – `TotalTicks`.

In addition to precise and consistent scheme for measuring time within DORITOS, we need to emulate parallel (asynchronous) devices. In real systems, a CPU does not typically wait for I/Os to complete; after a process makes an I/O request, the process requesting an I/O is blocked and the CPU continues to work by executing the next job in the ready state. Meanwhile, the disk performs the I/O request and interrupts the CPU when the I/O is completed. This requires emulation of asynchronous device. Figure 1 shows an overview of DORITOS organization.

Since DORITOS runs on a single CPU machine (i.e., a JVM), there can only be one Java thread running at any given time. To make the time for running device threads (e.g., DISK thread and TIMER thread) transparent to that of running the CPU, we stop the DORITOS time when a device thread is running. To implement this, the JVM purposely does not count the byte-code instructions while running device threads. Therefore, as far as DORITOS Kernel thread is concerned, no time is passed during device I/Os. Figure 2 describes the sequence of events of a simple scenario as in Figure 1.

[1] This is a reasonable assumption for simplifying the implementation; it is akin to a machine instruction which does not take a single clock cycle to execute.

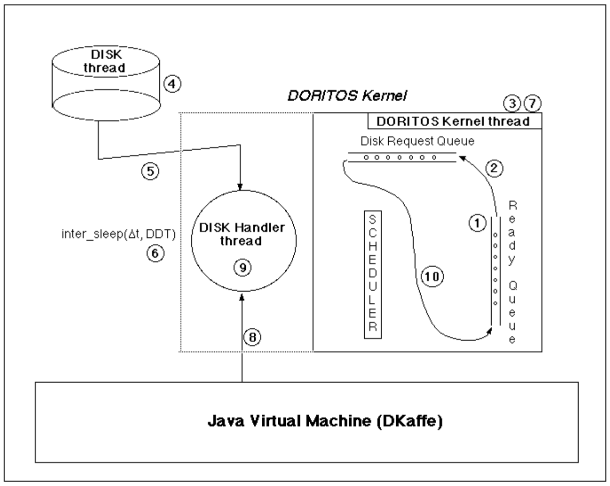

Figure 1: DORITOS System Overview

Since standard JVMs do not support such mechanics described above, Kaffe [9], a JVM, has been modified for such asynchronous device operations. Figure 3 shows the list of new instructions developed for DKaffe.

3 Assignments

Several assignments that focus on teaching real-time systems concepts have been developed to accompany DORITOS. Figure 4 shows the dependency graph for the assignments. In the figure, the *Multi-programming and System calls* assignment is not required but it is recommended; completion of this assignment will make subsequent projects to run user programs requesting variety of system calls. As shown in the figure, the projects are reasonably independent giving instructors flexibility in choosing the material to be taught in the class.

3.1 Measuring Execution Time for User programs

This assignment will familiarize students with some basic issues of real-time systems and DORITOS (how to run it and understanding its interface). This assignment is a stepping stone to later projects.

Students are to instrument DORITOS so that it can measure execution times of user programs (e.g., worst-case, average-case, etc.) as accurately as possible. The DKaffe instructions explained in Figure 3 can be used to measure the execution times of user programs; use of `Thread.getTime()` function should suffice.

Students are also to create and run several characteristically different user programs on DORITOS and measure their execution times. In measuring the execution time, students should attempt to estimate the worst-case execution time of each program by running the user programs with different inputs. Worst-case time is important in RTOS since it is required by an *admission control scheme*. An admission control scheme checks whether each submitted program can be finished within given deadline, then accordingly "admits" (i.e., put it in the ready-queue) or "rejects" the program.

1. At time t_d, a disk request is made by DORITOS user thread T_1.
2. DORITOS user thread T_1 is suspended and moved to the Disk Request Queue. DORITOS time stops by stopping the counting of the Java byte-code instructions executed into DORITOS time counter.
3. DORITOS Kernel thread is suspended in the JVM's ready queue. This is because the Disk device thread has higher priority than DORITOS Kernel thread and a disk request has been made.
4. DISK thread runs. Assume Δt is the disk access time.
5. When DISK thread is done, it signals the DISK handler thread.
6. DISK handler goes to sleep for Δt, the amount of disk access time. At the same time, DORITOS time restarts.
7. For Δt, DORITOS Kernel thread runs.[a]
8. When Δt has been passed, DISK handler thread is waken up by DKaffe.
9. Since DISK handler thread has higher priority, it will preempt DORITOS Kernel thread. DISK handler thread performs necessary DMA operations and interrupts the emulated CPU (i.e., DORITOS MIPS emulation)
10. The suspended thread T_1 is inserted into the DORITOS ready queue.

[a] There can be higher priority threads such as Timer device thread that is ready to run. In this case, the Timer device thread will preempt DORITOS Kernel thread.

Figure 2: A Simple Disk I/O Scenario

3.2 Simple Real-time Scheduling: Preemptive Deadline Scheduling

RT tasks can be classified according to their timing requirements, as *hard real time* (HRT), *soft real time (SRT)*, and *not real time* (NRT). A HRT task is a task that absolutely demands timely and logically correct execution while a SRT task is a task that prefers timely completion of the task. Finally, NRT tasks are tasks that require no real-time demands. Tasks can be further categorized as *periodic* and *aperiodic*. Periodic tasks are those tasks that enter execution state at regular time-interval while aperiodic tasks are tasks whose execution starting time cannot be anticipated.

The basic DORITOS scheduler simply fetches a task to run in non-preemptive fashion without considering temporal constraints of the task. Students will implement a preemptive real-time scheduler that honors the deadline of a given task in terms of the task classification above. This is called deadline-based scheduling in which the priority of a thread depends on the thread's deadline. When a task is submitted, the estimated execution time and desired deadline for the task are also submitted along with the task.

Students will add an admission control scheme by implementing the scheduler such that tasks that are guaranteed

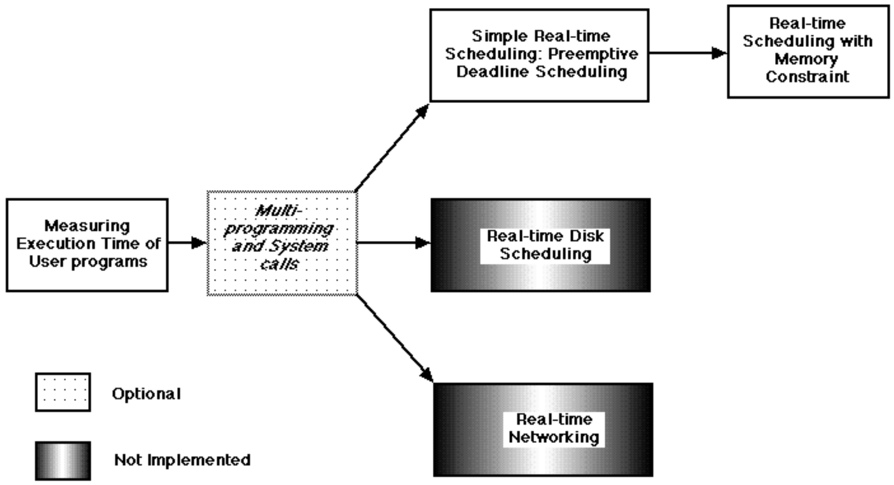

Figure 4: DORITOS assignment dependency graph; shaded box is optional

- `Thread.instrSleep(jlong num)`: Sleep the current Java thread for `num` Java instructions. Run the next ready Java thread.
- `Thread.instrSleep2(Thread* t, jlong num)`: Sleep the Java thread `t` for `num` Java instructions. Run the next ready Java thread.
- `Thread.countThisThread(Thread* t)`: Count the Java instructions used for the Java thread `t` into DORITOS time. Java instructions for executing device threads are generally not counted toward DORITOS time.
- `Thread.startTimer()`: Start counting Java instructions into DORITOS time.
- `Thread.stopTimer()`: Stop DORITOS time. Therefore, after this command has been issued, as far as DORITOS is concerned, the time is not passing.
- `Thread.getTime()`: Get current DORITOS time.
- `Thread.setTime(jlong ctime)`: Set current DORITOS time to `ctime`.

Figure 3: DKaffe Instructions

to complete before their deadlines will be accepted and inserted in the ready queue. Tasks that cannot be guaranteed will not be allowed into the system, and a warning should be issued by the operating system. Furthermore, an accepted task may miss its deadline due to a miscalculation on the worst-case execution time. Two possible solutions exist: (1) when a task finishes execution check whether the task actually completed in time; if not, issue a warning, (2) when the deadline arrives, terminate the executing job. Students should test their scheduler with various mixtures of the task types discussed above.

3.3 Real-time Scheduling with Memory Constraint

This assignment introduces memory requirements of threads in addition to the deadline requirements discussed above. When a job is submitted, students will make DORITOS check for enough CPU time and memory available to run the job within its deadline. Compilers generate the memory requirement for the user program. Students can get this information from file `Addrspace.java`, which is part of the DORITOS source code. `Addrspace.java` contains code that reads the size of binary executable file upon reading the file into DORITOS memory.

The scheduler must not simply deny a job just because it doesn't have enough memory at the time the job is submitted. The scheduler has to consider the deadline of the submitted job and the possible future memory availability to see if the deadline of the submitted job can be satisfied.

3.4 Multi-programming and System calls

This assignment has been converted from the NACHOS user program assignment to fit DORITOS environment. This assignment is optional. Students are to implement several system calls such as disk I/O requests, console I/O requests, `exec(), join(),` and `exit()` calls as in UNIX. For details, refer to DORITOS distribution package or NACHOS [3].

4 Project Status

The first experimental version of DORITOS system was completed in June 1998. It supports threading, synchronization, execution of multiple user threads, memory management including a primitive form of paging, system call handling, and timer device. Also the modifications for the Java Virtual Machine (called DKaffe – DORITOS Kaffe) has been completed. We still need to implement the RT file system and RT network capabilities for DORITOS. We expect to be able to take advantage of relatively simple-to-use Java socket libraries in developing the network capabilities.

The programming assignments that accompany DORITOS are to implement various real-time system components including scheduling, file system, and networking. Three assignments that teach the following concepts are completed: real-time scheduling algorithms (deadline, and memory), and measuring execution times of user programs.

So far, DORITOS has been tested on Linux-Intel machines and Solaris machines. However, since DORITOS runs on DKaffe, a Java Virtual Machine, it should run on other platforms once DKaffe has been compiled for the target platform. Currently, Kaffe, and therefore DKaffe, has been ported to numerous platforms including FreeBSD 2.x, Linux, NetBSD 1.1R, NeXTStep 3.x, SunOS 4.x, Solaris 2.x, OSF/1, MkLinux, and Irix 5 & 6.

5 Classroom Experience and Future Work

DORITOS has been tested in a real classroom (CS1651) in Fall term 1998 in the Computer Science Department at the University of Pittsburgh. The CS1651 (Advanced Systems Software) class is a follow-up to the CS1550 (Introduction to Operating Systems). DORITOS was distributed both as on-line and as a stand-alone CD, which was burned in house. Several students preferred to run DORITOS on their home Linux systems, and therefore used the CD version. During the semester, three times student questions prompted us to discover bugs in the DORITOS distribution, and the need to redistribute several files within DORITOS; DKaffe remained the same throughout the semester.

Four DORITOS projects have been assigned to students: (1) Measuring Execution Time of User Program, (2) System Calls and RT-File System, (3) Simple RT-Scheduling: Preemptive Deadline First Scheduling, and (4) Real-time Scheduling: Earliest Deadline First (EDF) with Periodic Tasks. Notice that we divided the assignment discussed in Section 3.2 into two parts – Assignments (3) and (4) – to allow students more time and separation of concerns.

1. *Measuring Execution Time of User Program:* This assignment was quite an easy assignment. Students spent the most of the time in reading and understanding relevant source codes. It served as an excellent introduction to the DORITOS system.

2. *System Calls and RT-File System (RT Disk Scheduling):* The students implemented some system calls in this part of the project. This assignment served to give us some experience, to prepare for the RT File System assignment.

3. *Simple RT-Scheduling: Non-Preemptive Deadline Scheduling:* Java's `Vector Class` simplified the assignment; students were able to concentrate on implementing the EDF without wasting the time with data structure algorithms for list, queue, etc.

4. *Real-time Scheduling: EDF with Periodic Tasks:* In this assignment, each task was submitted with a deadline as well as with a period. When a task finished, the task was resubmitted to the ready queue for the next period. The assignment teaches the students how hardware timer and associated interrupt service routine work (required for context switching) and how to reinitialize tasks' contexts so they can be re-started in each period. Some students simply forked the threads for each period creating redundant address space for the existing threads; since the basic DORITOS system does not provide a memory management scheme, the main memory is soon overflowed. The instructor should warn and discourage students re-forking threads for each period.

In summary, DORITOS provided a successful opportunity for combined learning of RT and OS system principles. Many informed lessons are learned from the first class ever taught with DORITOS; we defer the detailed discussions of our experience until quantitative data has been fully collected and analyzed. Lessons learned from the course will be used to improve DORITOS and to write a laboratory manual. We plan to release a beta test version for external testing sometime in early Spring 1999.

Acknowledgments

The authors would like to thank to Tim Wilkins, the author of the Kaffe JVM, for valuable input on how to carry out the modifications of Kaffe for DORITOS project and for making Kaffe freely available, Dan Bauman for implementing the file system part of DORITOS system, Dave Broudy for testing DORITOS system and commenting on its installation procedure, and CS1651 students in Fall 1998 at the University of Pittsburgh for their patience in using the pre-release version of DORITOS.

References

[1] G. Aguirre, M. Errecalde, R. Guerrero, C. Kavka, G. Leguizamon, M. Printista, and R. Gallard, *TExperiencing MINIX as a Didactical Aid for Operating System Courses*, Operating Systems Review, 25:32-39, July 1991.

[2] S. Chen, J. A. Stankovic, J. F. Kurose and D. Towsley, *Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems.* UM-CS-1990-077, The UMass-Amherst Dept. of Computer Science. 1990

[3] W. A. Christopher, S. J. Procter, and T. E. Anderson, *The Nachos Instructional Operating System*, available via ftp `sprite.berkeley.edu`, file `nachos/nachos-3.0.tar`.

[4] R. Holt, *Concurrent Euclid, the UNIX System and TUNIS*, Addison-Wesley, 1983.

[5] McKusick, Joy, Leffler and Fabry, *A Fast File System for UNIX.* ACM Transactions on Computer Systems 2, 3. pp 181-197, August 1984.

[6] Sun Microsystems, *Java Development Kit*, request for a copy can be made by sending an email via `http://www.javasoft.com`

[7] W. F. Tichy, *Programming-in-the-large: Past, Present, and Future*, Proceedings of the 14th International Conference on Software Engineering, 362–367, 1992.

[8] Rosenblum and Ousterhout, *The Design and Implementation of a Log-Structured File System.* ACM SIGOPS Operating Systems Review, No. 5, Vol. 25, 1991.

[9] T. J. Wilkins & Associates, *Kaffe: A free virtual machine to run Java code*, available via `http://www.kaffe.org`