Proceedings of the 2014 Federated Conference on
Computer Science and Information Systems pp. 819–826
DOI: 10.15439/2014F237
ACSIS, Vol. 2

Benu: Operating System Increments for Embedded Systems Engineer's Education

Leonardo Jelenković, Domagoj Jakobović, Stjepan Groš
University of Zagreb
Faculty of Electrical Engineering and Computing,
Unska 3, 10000 Zagreb, Croatia
Email: {leonardo.jelenkovic, domagoj.jakobovic, stjepan.gros}@fer.hr

***Abstract*—Most of today's computer systems, including rapidly emerging embedded ones, rely on an operating system. Consequently, the development of embedded systems and related software often requires a deeper understanding of operating systems. This paper presents a new incrementally built operating system and a learning course formed around it. Each increment builds on the previous one and introduces new system elements, new concepts and solutions, and a new set of assignments for improving or extending operations or simply demonstrating its use. Increments and assignments are designed to extend theoretical and practical knowledge in the operating system domain, give experience with non-trivial software systems and their development tools, familiarize the learner with basic computer hardware components and demonstrate device driver construction. The audience targeted by this operating system and course materials includes advanced students with (basic) knowledge of computer architecture, programming and operating systems. In addition, materials may be used individually as part of a lifelong learning process.**

I. Introduction

EMBEDDED computer systems are ubiquitous and have become increasingly integrated into our environment, thus increasing the need for engineers with appropriate skills. Developing and maintaining such systems involves hardware and software considerations. The software complexity inherent in embedded systems may vary from a simple controlling program running on a small micro-controller to a complex distributed system. Furthermore, complexity and logical correctness are not enough. For embedded systems, software (as well as hardware) must conform to additional requirements, such as deterministic behavior, dependability, security, low power consumption, long term operation and connectivity. To fulfill such requirements, an engineer must have appropriate skills and expertise. Computer science is a fast-developing domain, and it is common for computer science engineers to expand their skills long after getting their degrees, qualifying them for new challenges. Some of the challenges for the embedded system software developer are discussed in the following.

Software development for a new project may start from scratch or may reuse code from existing projects. Some simple systems may be built from scratch in a short period, but to achieve the required functionality in shorter periods, it is usually better to start with existing elements, e.g., a similar system or a collection of components. If there are no similar systems or useful components to reuse, one may start by selecting the operating system and appropriate development tools that will be used as a base. The operating system and development tools might be selected from the available commercial off the shelf tools (COTS), such as μC/OS-II [1], QNX [2], VxWorks [3], or from freely available tools, such as Linux [4] and and FreeRTOS [5]. Possible disadvantages of COTS systems may include price and possible problems with customization because all system details and internal component sources might not be available. The main problem with freely available systems lies in the absence of technical support (except in the form of free community forums). Whether we use COTS, freeware or a custom-built system, at some point of development, customizing the core system is likely to be required. The reason for customization might be a change in system requirements or a hardware change (e.g., a new or changed device, part or system, prior to the existence of official support). When system customization is required, even if we have the complete source code, an operating system (as for the core of any system) is a complex system, and changes are difficult to implement. Deep knowledge of operating systems is required if the desired change is to be made without compromising the existing functionality.

To be a successful embedded system engineer, one must understand computer architecture, have programming skills and have a deeper understanding of operating systems. The basics of those skills are usually acquired through education. Improvement of these skills is possible with practice and experience. While knowledge of computer architecture and programming language skill is usually improved through courses, operating system expertise is harder to attain. Depending on the instructor and the available options, operating system exercises usually concentrate only on using operating system operations through its interface (and improving understanding in this way). Knowledge obtained in this fashion may be adequate for a regular software engineer but is not sufficient for an embedded systems engineer who might be asked to customize some (operating system) core component.

A deeper knowledge of operating systems might be acquired individually, using the literature and resources from the Internet or through special seminars or courses. In this paper, we present the educational system, Benu, whose source code is freely available [6]. Benu was built primarily for

978-83-60810-58-3/$25.00 © 2014, IEEE

education on operating systems in embedded and real-time systems (RT), but it is generic enough that it can also be used for education on operating systems in other areas. The source code is accompanied by a textbook (currently only in Croatian), as Benu is used in the course “Operating systems for embedded computers” (OSFEC) [7].

Remainder of this paper is structured as follows. A comparison with similar systems is made in Section II. Section III presents the basic concepts and ideas behind Benu. The main part of the paper details the contents of the Benu increments, which are presented in Section IV. Benu usability is discussed in Section V. Conclusions are presented in Section VI.

II. Related work

Creating an operating system for education is an old idea and has been performed many times in the past because examples are the best educational tools, especially those examples that the teacher is comfortable using. A review of many such systems, often called “instructional operating systems,” has already been presented in several papers, e.g., [8] and [9]. A large number of such educational systems indicate the complexity in teaching this subject and the possibility of many different approaches, emphasis on different aspects of operating systems, different abstraction levels, influences of teacher preferences and the number of students in groups and their competences, i.e., their previous education. In this section, we compare only a few instructional operating systems, primarily highlighting features that are important for those systems when comparing them to Benu.

MINIX [10] is a well-known instructional operating system, modeled on UNIX, which was first introduced in the early 1980s and has since evolved to version 3. In addition to educational use, mostly for understanding UNIX system architecture and micro-kernel concepts, MINIX was later developed for systems with minimal resources, embedded systems, and systems requiring high reliability. A classical operating system textbook [11] details MINIX internals, providing its usage for education.

Linux [4] is not built to be an educational tool. However, because it was free from initial release, it has become fairly popular in academia, and it is often used as a base for student assignments in operating system courses. Because of the magnitude of the Linux source code, the assignments are mostly concentrated on utilizing operations that Linux provides, not on modifying its internal coding. Linux is a complete operating system, built to be effective and used on a variety of computers, primarily targeting personal computers, workstations and servers. Therefore, Linux source code, while freely available, in authors view is too vast and complicated to be used as an educational system for begginers. Only highly persistent students can master Linux complexity and use its internals as part of assignments.

The operating system μC/OS-II [1] is a small system, designated for embedded and real-time systems, with very limited hardware resources. It comes with a companion book, and it is free for educational use. μC/OS-II was primarily created for use in real-time systems, but because it is simple, it is also adequate for education, perhaps more for individual use by enthusiasts than in coursework.

Nachos [12] is a system skeleton prepared for student assignments (in C++) that complete some of the Nachos functionalities. Topics covered by Nachos include thread and process management, paging, file systems and network subsystems. Because solutions from previous assignments are used in the next ones, students are highly motivated to do their best in every assignment. Nachos comes with a MIPS processor emulator for the UNIX environment, which somewhat limits its use. Similar ideas used in Nachos are behind PortOS [13]. PortOS runs in an emulated environment, as a process in a Windows operating system. Assignments prepared for PortOS include multithreading, network and file subsystems.

Nachos and PortOS start with threads and are therefore on a higher abstraction level than Benu. In addition, Benu highlights the building process, building tools and advanced features using C. MINIX, and especially Linux, are complete operating systems, made to be used in more than one role, while Benu is built just for education and research. μC/OS-II is specifically designed for embedded and real-time systems and has many mechanisms for low-level system control exposed directly to user programs. For example, a user program may temporarily disable some kernel components, such as interrupts and scheduling. Although a few of such mechanisms can be found in Benu, we do not encourage their usage; we prefer accomplishing all such operations through the kernel.

III. Benu basic concepts

Benu is a collection of increments that uses a step-by-step presentation of the core operating system operations, data structures and algorithms, where each new increment introduces only a few new subjects. Other educational operating systems, while presenting a single topic, still use the complete system, highlighting related elements from it. In Benu, using increments, the student can focus better on only the subjects introduced in that increment, thus simplifying the learning process of an otherwise very complex system. The evolution of operating system components is roughly presented through increments, starting with the basic functionality in one increment and adding extended functionalities as they are needed. In addition to operating system topics, using Benu in education might improve the other skills required for embedded system development. These skills include the advanced use of the C programming language, experience with development and debugging tools and methods, and familiarization with POSIX for real-time and embedded systems.

For education, Benu can be used without supervision, simply progressing through prepared assignments. Better results can be achieved faster if assignments are preceded with some theoretical introduction, e.g., that presented in [7], with topics covered, such as those presented in [14], or any operating system textbook, e.g., [15], [11] or [16]. Source code dissection, assignments and other experiments should follow theoretical

introductions to broaden students' understanding. Every increment in Benu is associated with example assignments. New components, methods or principles are carefully chosen and added such that each new major increment brings the proper number of new elements for ease of understanding. Some concepts that are more radical and complex, such as threads and processes, are introduced in several smaller increments. Changes from one increment to the next can be easily tracked using text or graphical tools, such as Meld [17].

The current version of Benu is prepared for both Intel i386 and ARM platforms. Although the i386 platform is not typical for embedded systems, it has the advantage of providing educators with access to development tools, emulators, computers and documentation. Adding support for other processors is supported by the layered architecture of Benu, with a separate hardware abstraction layer. As a development platform, Linux is selected because all of the required tools, i.e., GNU development tools [18], are freely available and easy to install on Linux. Linux itself may be run as a development platform in an emulated environment, requiring only the emulator to be installed on the host computer (if the host computer is not already Linux-based).

Based on our experience in teaching computer architecture, programming languages and operating system basics as well as using Benu in an advanced operating system course, we observed that using Benu accomplished the following:

- A deeper understanding of the operating system, its components, data structure, operations, algorithms and limitations,
- Improvement of advanced programming skills, which in turn produces shorter, more efficient, extendable and more readable (and thus reusable) code,
- An understanding of the capabilities of developing and debugging tools and computer emulators,
- The ability to build embedded system software from scratch, not relying on any operating system interface, and preparing images to be loaded into systems with variety of memory configurations,
- Expertise with POSIX interfaces for timers, threads and signals for real-time and embedded systems,
- The ability to navigate within and use larger source code projects (written by others), the discovery of usual concepts and practice in source code naming, file structure and management tools, and
- An improvement in the student's problem-solving skills.

We do not claim that Benu is the best choice in the field of embedded operating system education, but it may be among the easiest for beginners. Starting increments are simple and do not require preparation. Students can start early, familiarize themselves with the development environment and be prepared for the more demanding increments that follow.

IV. Contents of increments

Benu is created using basic operating system principles as a base [15], [16] and is modified to better suit the embedded system environment, simplified for educational purposes, and built for incremental topic introduction. Benu contains eight major increments in the source code, which, for the rest of this section, are just "increments" or "chapters". Each chapter has several minor increments, depending on the complexity of the subjects presented in the chapter, as shown in Table I. Details about each chapter, its purpose, the components it presents, and possible assignments, are presented in the following sections.

TABLE I
Major and minor increments in Benu

Major increments – chapters	Minor increments
Chapter_01_Startup	01_Startup 02_Example_clock
Chapter_02_Source_tree	01_Source_tree 02_Console 03_DEBUG 04_Debugging
Chapter_03_Interrupts	01_Exceptions 02_PIC 03_Dynamic_memory 04_Interrupts
Chapter_04_Timer	01_Time 02_One_timer 03_Timers
Chapter_05_Devices	01_Devices 02_Keyboard 03_Serial_comm
Chapter_06_Shell	01_Shell 02_Arguments 03_Programs 04_Makepp
Chapter_07_Threads	01_User_threads 02_Threads 03_Ext_context 04_Synchronization 05_Messages 06_Signals 05_Sched2
Chapter_08_Process	01_Syscall 02_User_mode 03_Programs_as_module 04_Programs_as_process 05_Static_processes 06_Processes

A. *Chapter 01 – Development environment*

The goal of Chapter 01 (with related materials from the textbook) is to present the environment used for developing system software, i.e., Benu, which will run on bare-bone hardware (real or emulated). Compiling and running system software requires special steps during the compilation and linking phases, supported with appropriate configurations, and is thus significantly different from compiling and running application programs. Because of this straightforward goal, the code is purposely simple; it just displays the "Hello World" message on the console.

There are only three files in Chapter 01: two with source

code (one in assembly and one in C) and one with shell script used for compiling. The assembly code is small, but it is required for processor initialization (stack pointer and status register). The assembler then transfers control to a function written in C. A single function is placed in a C file, i.e., a function that writes a text string into video memory, thus displaying it on the console. While low-level operations implemented in assembly code and device drivers might be interesting to some, they are not essential for using and understanding Benu and the principles it describes. It is possible to learn most operating system concepts without a knowledge of assembly language or device driver details; therefore, assembly and device drivers are placed into a separate directory (from the next chapter) to isolate them from the other increments and lessons. To compile and link the source codes into an executable and run it, a shell script is used only in this chapter. A shell script better reflects the necessary steps involved and therefore better serves the purpose of this chapter, which is to show how the appropriate tools are used. The script illustrates how to start the compiler and linker and how to create the system image and start emulator, using all necessary flags and parameters. Beginning with the next chapter, Benu uses standard build tools, i.e., `make` with appropriate definitions across `Makefiles`.

The second minor increment of the first chapter demonstrates how very simple systems can be implemented without having an OS in a traditional sense. For that purpose, a simple clock is implemented that uses hardware timers and displays a counter on the screen.

Assignments for the first chapter should include only the preparation of the development environment, e.g., installing the required tools, downloading Benu, and compiling and running it in an emulated environment. The first few assignments should also focus on using a revision control tool, such as Git [19] or Subversion [20].

B. Chapter 02 – Layered structure

The instructional goal of the second chapter is to demonstrate the modular and layered approaches to operating system design. To reduce complexity, systems are often developed modularly, using the "divide and conquer" approach. A single module or component is simpler than an entire system, and it is thus easier to develop and test. Furthermore, modules can be concurrently developed by different programmers, reducing development time.

Operating systems are designed and built in a modular manner; each module is a "subsystem". Typical subsystems include input-output, memory management, thread and process management, network, security and file subsystems.

Operating systems are also layered, and the layers communicate with one another using strictly defined interfaces in a top-to-bottom fashion, i.e., a higher layer uses the services of the immediately lower layer only. One of the purposes of introducing layers is to separate smaller, architecture-dependent code that sits immediately above the hardware, often called the "hardware abstraction layer" (HAL), from the larger architecture-independent code in the layers above. This separation makes it easier to port the operating system across different architectures because only the HAL has to be modified or rewritten. The architecture-independent code consists of several layers, one built upon another. The first layer, called the "kernel", is immediately above the HAL and provides the core operations for system resource management. System resources include hardware components, such as devices, and software components, such as synchronization objects and the task scheduler. The kernel implements fine-grained operations that manipulate system resources and provide an interface for accessing them from the layer above, i.e., the application programming interface (API) layer. The API layer (henceforth referred to as "api") uses the kernel layer to implement any designed interface (e.g., POSIX [21]) for user programs, i.e., the next higher layer. User programs use the "api" to implement operations that are required by the user (which is the highest layer of the operating system).

Benu adheres to the described layered approach with four layers, named after directories where their code is placed: arch (HAL), kernel, api and programs. There is also an additional pseudo-layer, the library layer ("lib"), which captures the utility functions used by the other layers, e.g., string manipulation and list operations. A list of all of the layers as well as their containing directories in Benu is shown in Fig. 1.

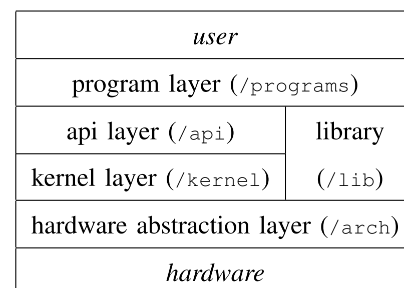

Fig. 1. Layers in Benu

In Chapter 02, the layers contain no functionality: the corresponding directories are mostly empty, and they are just placeholders for the components that will be implemented in subsequent chapters.

Each layer has a clearly defined interface for the higher layers. Fig. 2 shows an interface example introduced in Chapter 04, when the user program uses the `clock_nanosleep` operation. The function `clock_nanosleep` is defined in api as a system call to the kernel function `sys__nanosleep`. The kernel function uses the HAL interface `arch_timer_set` which calls `i8253_set_time_to_counter` from the device driver (via the timer interface function `set_interval`).

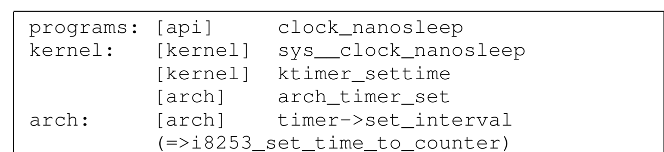

Fig. 2. Interface chain for operation `clock_nanosleep` (in Chapter 04)

Interfaces that one layer provides for the next one are defined in header files placed in the `include` directory (placed in top level, same as those from Fig. 1).

The naming convention used throughout Benu follows certain prefix rules. Kernel functions that provide the interface to the higher layer (i.e., to api) have the prefix `sys__`; internal kernel functions (i.e., the ones used only within the kernel) are prefixed with `k_` or just `k`; functions that are part of HAL are prefixed with `arch_`; and the device driver's functions are prefixed with a short device name.

Devices and subsystems are used through a predefined interface. Chapter 02 defines the interface for printing characters on the console, i.e., `console_t` (defined in `include/types/io.h`) with the following elements:

```
int (*init)(void *p);
int (*clear)();
int (*gotoxy)(int x, int y);
int (*print)(int attr, char *text);
```

Each structure element is a function that implements a specific operation. The same functionality may be achieved by different devices or components. For example, for simple console display, a graphics card can be used, as in Chapter 02, or a serial port connected to a terminal, as in Chapter 05. Switching from one console to another is accomplished using different objects (variables) that implement the same interface. In Chapter 02, in `kernel/startup.c`, consoles for kernel and user programs are selected using variables `k_stdout` and `u_stdout`, both referencing variable `vga_text`, defined in the device driver file `arch/devices/vga_text.c`.

The same principle for defining an interface is frequently used in Benu, i.e., using the structure with functions and parameters like `console_t`. Interfaces are defined for the interrupt device (Chapter 03), timer devices (Chapter 04), general devices (Chapter 05), and dynamic memory allocators (Chapter 03). Interfaces make separations easier, replacing one device or component with another simpler and source code more readable.

Chapter 02 also presents possibilities for tracing and debugging. Because the system being created uses (emulated) hardware directly, debugging is harder than debugging a traditional program which can be paused and inspected at any moment. One primitive debugging method is to insert print commands in the source code, e.g., with the `printf` operation or with `LOG` and `ASSERT` macros that will be executed only in the `DEBUG` mode. Another method is to use the appropriate tools that enable the developer to stop (and examine) the system while executing, such as the GNU debugger and the QEMU emulator [22], which are used in the demonstration example.

Assignments for Chapter 02 can be focused on layered architecture and on debugging. For example, the assignment can be to divide the console into two parts, one for kernel messages and the other for program output. Implementation through two different `console_t` objects, almost the same as the one provided, will require little coding but will need implementation in nearly all layers. Debugging can be practiced by requesting stops and system inspections at defined points or by discovery of covertly inserted bugs that cause system failures.

C. *Chapter 03 – The interrupt subsystem*

Interrupts are very important mechanisms, not only for managing devices but also for other purposes, e.g., protection from system calls, thread scheduling (timer interrupts), memory management and program failure detection. The instructional goal of this chapter is the presentation of interrupt handling methods in the operating system. The presented material is, by necessity, simplified and therefore has some limitations, but it also offers space for possible extensions using student assignments.

The primary function of the interrupt subsystem is to provide an interface for connecting the interrupt handler functions (which might be part of other subsystems) with interrupts, i.e., the functions that will be used as interrupt handlers. The interface for registering the interrupt handler function `hnd` with an interrupt identified by number `id` is defined in HAL as follows:

```
arch_register_interrupt_handler(id, hnd);
```

For every source of interrupts identified by an interrupt number, at least one handler function can be defined. When the interrupt occurs, all handlers for that interrupt are called sequentially.

Benu uses an Intel 8259 programmable interrupt controller (PIC) in HAL through the `arch_ic_t` interface, making future replacements with other interrupt controllers (e.g., APIC) easier. Chapter 03 defines the interrupt subsystem but handles only the processor's interrupts because no other device driver is used in Chapter 03. Device drivers are added in ensuing chapters, i.e., the timer in Chapter 04 and the keyboard and UART in Chapter 05.

Registering more interrupt handler functions for a single interrupt number can be accomplished using a static data structure (i.e., an array with predefined size for each interrupt number) or a dynamic data structure, such as a list. A list provides more flexibility and less overall memory consumption, but requires dynamic memory management. Because dynamic memory management will also be required for other subsystems, it is introduced in this increment. Two algorithms for dynamic memory management are presented: the simple "first fit" (FF) method with a "last in, first out" list of free blocks and the more complex "two level segregated first" (TLSF) method [24]. FF is simpler, and, on average, faster than TLSF. However, TLSF provides reduced fragmentation; and more importantly, the execution time complexity of TLSF is $O(1)$, while the worst case for FF is $O(n)$, where n equals the number of free blocks. Therefore, TLSF is a candidate for use in real-time systems.

Student assignments for Chapter 03 include improvements to the interrupt subsystem, such as adding priorities to existing interrupt handlers. Then, when multiple interrupts overlap,

they can be handled according to their priorities. Other assignments can be focused on implementing additional dynamic memory management algorithms, such as “best fit”.

D. Chapter 04 – Time management

Most program activities, especially in embedded systems, must be executed in a timely manner. Therefore, the operating system must provide support for time management through a timer subsystem. Most required operations include system time control (“set” and “get” system time), thread execution delays and programmable future actions, i.e., alarms. The operating system also uses time for managing input/output devices, scheduling and maintenance.

The timer subsystem implemented in Benu consists of two components: a lower-level component implemented in HAL and a higher-level component implemented in the kernel. The component implemented in HAL uses an Intel 8253 programmable interrupt timer (PIT) through the `arch_timer_t` interface. The primary operations provided by that interface include keeping system time and a single alarm, which, upon alarm expiration, forwards a call to the kernel. The component implemented in the kernel extends capabilities to multiple alarms available to the kernel and programs and provides operations for program delays.

Assignments for Chapter 04 may include extensions of the timer subsystem. For instance, absolute times can be changed to relative times or the sorted list of alarms may be replaced with some more efficient structure; monotonic clock can be added to the system, a clock that can’t be changed with `*set*` interface (as current real-time clock can); a software watchdog timer can be implemented; other more advanced hardware devices than the Intel 8253 can be used to achieve better resolution for time management.

E. Chapter 05 – Device interface

Every device in a computer has its own specifications. To simplify device management, devices are grouped into classes, and an interface is defined for each class. When creating a device driver for a device, the appropriate interface must be implemented. The simplest device driver interface must include functions for sending data to the device and functions for reading data from the device. Such an interface may not be as efficient as a more complicated interface that, for example, uses direct memory access capabilities of the devices, but it is a good starting point for illustrating the integration of device control into an operating system. For that reason, Benu uses a simple interface defined by the structure `device_t` (defined in `include/arch/device.h`) with the following functions:

```
int  (*init)(uint flags, void *parm, device_t *dev);
int  (*destroy)(uint flags, void *parm, (...) );
int  (*send)(void *data, size_t size, (...) );
int  (*recv)(void *data, size_t size, (...) );
void (*irq_handler)(int irq, void *dev);
int  (*callback)(int irq, void *dev);
```

The usage of the `device_t` interface is demonstrated on three devices for which the device driver is prepared within Benu: the display driver (replacing `console_t`), a keyboard driver using the Intel 8042 controller, and a serial port using the 16550 UART device. The interface `device_t` is intended for use only within the kernel. The kernel allows programs to access these interfaces indirectly through `sys__device_*` system calls (`*open`, `*close`, `*read`, and `*write`).

Assignments for this chapter may include introducing the program (thread) blocking state to read/write operations until they are completed on a device. Other assignments include adding device drivers for other devices or improvement to current devices, e.g., adding scroll history capabilities to the console display driver.

F. Chapter 06 – Command shell

The interfaces offered to users on today’s computer systems range from graphical interfaces, with buttons or menus, to console-oriented interfaces, such as the command line interface, where the user types commands to be executed. Because Benu only has a text-based console, a command-line interface is implemented and presented. From an educational point of view, the implementation of a command line user interface is useful for two reasons: parsing the command line and sending parameters to programs (as strings, without interpretation).

The second novelty introduced in Chapter 06 is in the compile script (`Makefile`). Every program from the `programs` folder may be independently included in or excluded from compilation. This change further distances the program layer from the kernel, making the kernel (and thus HAL and api) potentially usable for many purposes.

Assignments for this chapter may include improvements to the shell program, e.g., adding history and auto-complete features.

G. Chapter 07 – Thread management

The systems presented in the previous chapters or the systems based on them (e.g., created as assignments) are very simple. Still, they may be sufficient for numerous embedded systems. More complex systems require additional features, such as multithreading and processes. Introducing those features has a strong impact on all of the system components, making the system significantly more complex and larger, and thus it is not recommended if those features are not required by the embedded system.

Multithreading support simplifies complex system implementation. Independent tasks may be run independently as threads, with their own timings and resource requirements that are more easily coded and satisfied at runtime.

Based on our teaching experience, we believe that multithreading programming is one of the most difficult subjects in computer science education. Thinking “in parallel” is required, and any shared resource must be considered and properly protected. Synchronizations via semaphores and monitors have to be carefully designed to achieve desired sequences and avoid deadlocks or simultaneous changes on

any shared resource. Multithreading is increasingly important because modern processors are multicore and manycore and require multithreading for using all of the processing power the processors can provide. Thus, many operating system and programming courses emphasize multithreading with the other subjects.

The multithreading covered in Benu includes both lower-level kernel operations, such as thread creation with resource allocation and context switching, and higher-level operations, such as scheduling, synchronization and communication. A priority scheduler is used with "first in, first out" as the second level scheduling criteria (for threads with equal priority). Semaphore and monitor synchronization mechanisms are included, and communication is provided through messages and signals.

Many assignments can be created for Chapter 07. Threads can be used for kernel operations, e.g., in an interrupt subsystem for processing individual interrupts. Existing thread operations can be improved or extended. Semaphores and monitors may be extended with "try-wait" and "timed-wait" operations or improved with priority inheritance protocols. New synchronization and communication mechanisms could be added, such as barrier, read/write locks and pipes. Program assignments that solve some synchronization problems can also be created.

H. Chapter 08 – Process Management

Programs, especially more complex ones, may have bugs that might compromise the system. In a multitasking environment, there should exist mechanisms that protect the kernel and other tasks from a malfunctioning task. The usual protection mechanisms include processor operation modes and memory protection, both requiring hardware support from the processor. If the system is running in unprivileged processor mode, the thread may not be able to execute instructions that could compromise the entire system. For example, if a thread cannot disable interrupts but must instead use a synchronization function for a critical section, the eventual error that leads to an infinite loop in a critical section will only affect that thread and other threads that use the same critical section object, while the rest of the system will be unaffected. The same reasoning is true with memory separation methods, such as memory protection and virtual memory. If a thread cannot change memory locations outside its defined boundaries, it cannot compromise kernel data or other programs.

Grouping threads that work on the same operation into a single process (i.e., threads that are created within the same instance of a single program) will isolate them from other threads, and vice versa. An error in one thread will usually have only a local effect on threads in the same process. Errors that compromise a shared system resource, like a device, will, however, still be an issue for the entire system.

Chapter 08 brings a further separation of programs and the kernel by introducting the privileged and unprivileged processor modes, forcing software interrupts as mechanisms for calling kernel functions (syscalls). Additionally, memory protection is introduced using segment registers of the Intel 80386 processor family for simple virtual memory implementation. Threads use logical addresses and cannot reach outside the boundaries of their processes. Any attempt to do so will trigger an interrupt, and the thread will be terminated. Compiling programs using logical addresses complicates the building process. Since kernel and programs must be prepared for different locations (physical and logical) they are separated into different objects after compilation. To simplify emulation in those increments GRUB was used as boot loader where program objects can be prepared and loaded as modules.

Assignments for Chapter 08 might be same as those for Chapter 07 because major differences in the chapters are in the implementation of the syscall mechanism via a software interrupt with the address space changing from the user to the kernel, from logical (process) to physical address space. Both changes require special syscall parameter handling, which provides a sufficient challenge to adopt.

V. USING BENU

Even in the last increment (that includes all) there are 56 source code files (.c) and 73 header files (.h) (including the 16 example programs). Furthermore, about half of them are only for layering purposes (parameter checking and forwarding call to lower level function). The combined source code of the kernel, HAL, library and api (all except headers and example programs) have approximately 7,500 lines of code (as counted by the `cloc` program). A system this small cannot have advanced components, such as paging, file systems and networking, which are required in more complex systems. However, some simpler systems, such as the ones used in embedded computers, may use an operating system like Benu because they may not need advanced components at all. Future work on Benu includes developing those advanced components, though in some minimalistic form that has yet to be devised. Otherwise, such complexity will significantly reduce its educational value. The components present in Benu are built on basic principles, avoiding too many complexities that may have better properties. From an educational viewpoint, this approach leaves the basic course straightforward and allows for advanced student assignments and projects.

Although Benu is built on somewhat different ideas than Linux and MINIX, it can be a good prelude to studying them. Because Benu is not a complete operating system, it can provide a simpler example for embedded systems that do not need advanced components. Due to their simplicity, Benu source codes do not have as complex interconnections in their kernel as real systems have (e.g., Linux). Compared to other systems, the components in Benu are easier to change and replace, and new components are easier to integrate and evaluate, thus making Benu also usable in operating system research. For example, current synchronization mechanisms can be changed and extended with other models (e.g., [23]), priority inheritance and priority ceiling protocols can be embedded, deadlock detection can be implemented, thread scheduling can be upgraded, and interrupts can be processed

with threads [24]. As example extensions, “round robin” (RR) and “earliest deadline first” (EDF) schedulers are implemented and presented in Benu.

In addition to Benu source code, a companion textbook is available to students (currently only in Croatian). The textbook is tightly coupled to Benu but includes theoretical explanations of operating system topics, excluding advanced components such as networking and file systems. A quick start with Benu is possible with a basic knowledge of computer architecture and operating systems, and a moderate knowledge and experience in the C programming language. Therefore, a Benu course should best be placed after courses that cover fundamentals. In addition, Benu could be learned as a post-graduate, as part of the lifelong education process. Benu targets students interested in operating system internals and experimentations with it and students interested in software development for embedded computers.

Benu has been used as a teaching tool since the 2009/2010 academic year, when OSFEC was offered as an elective course in master computing science studies. Most students who take this course show much interest, and most of them successfully complete the exams. However, for some students, the assignments and exams are harder to complete. After an analysis, it was found that the students who had problems did not have the recommended prerequisite courses in bachelor studies, especially the courses that exercise C programming skills.

VI. Conclusion

Mastering operating system topics, including theory, implementation details, tools and common practices, can be faster and more interesting for students if a simple system such as Benu is used in teaching and assignments. The uniqueness of Benu among other instructional operating systems is in its incremental build structure, allowing gradual introduction of operating system components. Each increment logically extends the previous one with only a few new elements, which are, consequently, easier to adopt.

An operating system is a complex system, and, even with simplifications, as in Benu, many students may find it difficult to master. However, the majority of students do not need to master all of the components of an operating system. For some, it will be enough to master the kernel layer or even just a specific components, like the interrupt subsystem, timer subsystem and threads. For others, Benu may be just a starting point, one step toward understanding more complete and complex systems.

A quantitative comparison of Benu with other (instructional) operating systems is not performed. To perform such a comparison, we cannot just use the other systems, but we would also need to prepare teaching materials and assignments closely coupled with them, as the current OSFEC materials are coupled to Benu. Nevertheless, based on our experience with Benu, we can conclude that Benu offers a great deal for independent study and exercise and provides a base for faster learning, not only in the operating system domain but also in embedded system software development.

Acknowledgment

This work was supported by FP7 project Embedded Computer Engineering Learning Platform (E2LP).

References

[1] J. J. Labrosse, *MicroC OS II: The Real Time Kernel*, 2nd ed. CMP-Books, 2002. ISBN 1578201039
[2] QNX operating systems. [Online]. Available: http://www.qnx.com/products/neutrino-rtos/
[3] Wind River VxWorks. [Online]. Available: http://www.windriver.com/products/vxworks/
[4] The Linux kernel archives. [Online]. Available: http://www.kernel.org
[5] FreeRTOS. [Online]. Available: http://http://www.freertos.org/
[6] L. Jelenković. (2012) Benu source code. [Online]. Available: https://github.com/l30nard0/Benu
[7] ——. (2010) Operating system for embedded computers, course homepage (in croatian). [Online]. Available: http://www.fer.unizg.hr/en/course/osfec
[8] C. L. Anderson and M. Nguyen, “A survey of contemporary instructional operating systems for use in undergraduate courses,” *J. Comput. Sci. Coll.*, vol. 21, no. 1, pp. 183–190, Oct. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1088791.1088822
[9] Y.-P. Cheng and J.-C. Lin, “Awk-Linux: A lightweight operating aystems courseware,” *IEEE Trans. Educ.*, vol. 51, no. 4, pp. 461–467, nov. 2008. doi: 10.1109/TE.2007.912571. [Online]. Available: http://dx.doi.org/10.1109/TE.2007.912571
[10] MINIX 3. [Online]. Available: http://www.minix3.org
[11] A. S. Tanenbaum and A. S. Woodhull, *Operating systems design and implementation*, 3rd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 0131429388
[12] W. A. Christopher, S. J. Procter, and T. E. Anderson, “The Nachos instructional operating system,” in *Proceedings of the 1993 Winter USENIX Conference*, 1993, pp. 479–488.
[13] B. Atkin and E. G. Sirer, “PortOS: an educational operating system for the post-PC environment,” in *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, 2002. doi: 10.1145/563517.563384 pp. 116–120. [Online]. Available: http://dx.doi.org/10.1145/563517.563384
[14] C. Yang, “Computer operating systems in electrical engineering curriculum,” *IEEE Trans. Educ.*, vol. 36, no. 1, pp. 177–180, 1993. doi: 10.1109/13.204841. [Online]. Available: http://dx.doi.org/10.1109/13.204841
[15] A. Silberschatz, G. Gagne, and P. B. Galvin, *Operating system concepts*, 8th ed. Wiley, 2011. ISBN 1118112733
[16] L. Budin, M. Golub, D. Jakobović, and L. Jelenković, *Operating systems (in Croatian)*, 3rd ed. Zagreb: Element, 2013. ISBN 9789851976107
[17] Meld: Diff and merge tool. [Online]. Available: http://meld.sourceforge.net
[18] GNU operating system. [Online]. Available: http://www.gnu.org
[19] Git (distributed version control system). [Online]. Available: http://git-scm.com/
[20] Apache Subversion (version control system). [Online]. Available: http://subversion.apache.org/
[21] IEEE and O. Group. The open group base specifications issue 7. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/
[22] QEMU: open source processor emulator. [Online]. Available: http://wiki.qemu.org
[23] P. A. Buhr, M. Fortier, and M. H. Coffin, “Monitor classification,” *ACM Comput. Surv.*, vol. 27, pp. 63–107, March 1995. doi: 10.1145/214037.214100. [Online]. Available: http://dx.doi.org/10.1145/214037.214100
[24] S. Kekckler, A. Chang, W. Chatterjee, and W. Dally, “Concurrent event handling through multithreading,” *IEEE Trans. on Computers*, vol. 48, no. 9, pp. 903–916, 1999. doi: 10.1109/12.795220. [Online]. Available: http://dx.doi.org/10.1109/12.795220