Operating Systems
for Embedded Computers

Summary of textbook:
Operacijski sustavi za ugrađena računala
(in Croatian)

Author: Leonardo Jelenković

Zagreb, 2012.
Contents

Preface ii
1. Introduction 1
2. Development environment and tools 2
3. Building stand-alone system 3
4. Source tree 4
5. Interrupts 5
6. Dynamic memory management 6
7. Time management 7
8. Devices 8
9. Command shell 9
10. Multithreading 10
11. Processes 11
12. Choosing operating system for embedded computers 12

Appendix A. Development tools: installing, virtual machine settings, Git usage example 13
Appendix B. Special C keywords and macros 14
Appendix C. Linker script examples 15
Appendix D. Advanced topics on synchronization 16
Appendix E. Adding schedulers: RR and EDF 17
Preface

Preface describes importance of embedded system and operating system details knowledge for embedded system engineer, and in which areas this handbook with Benu could help.

*Benu* is a collection of increments that uses step by step presentation of core operating system operations, data structures and algorithms, where each new increment brings only a few new subjects. Other educational operating systems, while presenting single topic still use complete system, highlighting related elements from it. In *Benu*, using increments student can focus better on subjects introduced in that increment, and thus simplify learning process of otherwise very complex system. With increments we can also present the evolution of operating system components, starting with basic functionality in one increment, and then adding extended functionalities, as they become needed. Beside operating system topics, using *Benu* in education might improve other skills required for embedded system development, including advanced usage of C programming language, experience with development and debugging tools and methods, and familiarization with POSIX for real-time and embedded systems.
1. Introduction

In introduction an overview of embedded systems and operating systems is given. Furthermore, possible software designs used in implementation are discussed. Finally, properties of an controlling task are detailed.
2. Development environment and tools

Second section presents layered operating system architecture with \textit{Benu} as example. Development tools used to create and modify \textit{Benu} are introduced (\textit{GNU tools} mostly).
3. Building stand-alone system

Creating operating system is very different than creating programs for existing operating system. Operating system must “take control” of system, initialize its components and provide interface for programs and users. Difference does not come only on operational level, but also on development environment.

In this section, an basic system (“hello world”) is created and explained: from source files, development tools, running and debugging in simulated environment.
4. Source tree

Bigger projects (like operating system source) have many source files. How to arrange them into directories depends on the project. For operating system source, commonly used directory structure reflects layered architecture (hardware abstraction layer, kernel layer, interfaces to programs) and subsystems. Benu uses layers for directory tree, which is presented in this chapter.

Compiling and linking ("building system") is usually defined with *Makefile* scripts. There are several conventions for placing compiled and linked files – "object files" (define through *Makefiles*). They are discussed in this chapter and one used in *Benu* is detailed.
5. Interrupts

Interrupts are mechanisms for handling asynchronous events from input-output (IO) devices. Why are they needed, how are they handled in most processors is shortly explained in this chapter. Details how are interrupts handled in x86 architecture is also presented. Finally, what an IO subsystem must provide is detailed and shown on subsystem implemented in Benu.
6. Dynamic memory management

Dynamic memory management is required for kernel operations (for descriptors, buffers, stacks and heaps for threads) and for programs (using heap). Some data structures may be statically allocated, but more flexibility is achieved with dynamic management. For real time and embedded systems algorithms for dynamic memory must have certain properties: speed, efficiency (minimal fragmentation) and predictability ($O(1)$ complexity preferred). Several algorithms for dynamic memory management are presented, while those implemented in Benu are detailed.
7. Time management

Time management subsystem must provide operations for activities (in text called “alarms”) that must be started on exact time. Those activities may be asynchronous (“programmed”), performing other jobs in the meantime, or synchronous when application is delayed until given time.

Benu implements time management with POSIX interface. Operations include clock manipulation (“get” and “set”), delay and alarm (one time or periodic). Implementation principles for mentioned operations are described, while those implemented in Benu are detailed (including POSIX interface).
8. Devices

Handling interrupts from IO devices requires interrupt handling but also device drivers. In this chapter a simple device driver interface is presented and its usage demonstrated on implementations (in Benu) for display, keyboard and serial port drivers.
9. Command shell

Simple command shell is implemented in Benu. Possibilities for creating and starting custom programs using shell or directly (with definitions through Makefile) is presented in this chapter.
10. Multithreading

This chapter discusses multithreading topics: “when is multithreading required”, “what are its benefits and drawbacks”. Implementing multithreading requires a lot of effort, changes, and new operations in the system (e.g., scheduling, synchronization, and communication). All these topics are discussed, and multithreading implementation in Benu is shown and analyzed.
11. Processes

Protecting kernel and other programs from faulty (or malicious) ones is required for some systems. Protection mechanisms are discussed in this chapter with emphasis on separating memory for each program into separate “process”. Processes are in Benu implemented using segmentation ability of x86 processors. Although not implemented, “paging” is also discussed.
12. Choosing operating system for embedded computers

Previous chapters describe operating system components, their internals and *Benu* implementation details. However, many embedded systems may be based on COTS or free operating systems as base. In this chapter some of such systems are presented and analyzed. Most basic principle in many such projects is compromise on required features and price. Somewhere will cheap systems be satisfactory, somewhere an hard RTOS must be used while elsewhere an combination of operating systems (for each node) will be adequate.
Appendix A. Development tools: installing, virtual machine settings, Git usage example

First appendix shortly presents steps for creating development environment (including tools for compiling, linking, running in emulator). Since most larger projects are created by a team (more than one person), a source control (revision control) tool is presented (which is used for students projects on Benu).
Appendix B. Special C keywords and macros

Creating software for embedded system requires many optimizations (speed, size, placement) for which deeper knowledge of programming language is required. In this appendix some useful C keywords and macros are explained (mostly on examples from Benu or in simplified forms).
Appendix C. Linker script examples

Embedded system memory usually consists of ROM and RAM. How to compile systems that should have some of its parts present in ROM while other should be loaded in RAM is usually defined through “linker scripts”. In this appendix two more system examples are presented which with appropriate scripts.
Appendix D. Advanced topics on synchronization

While most used synchronization mechanisms (semaphore and monitors) are presented with threads (Chapter 10), this chapter presents some extensions (“try wait”, “timed wait”, “multiple wait”, recursive locking), problems (“priority inversion”) and solutions (“priority inheritance” and “priority ceiling”), and other synchronization mechanisms (“spinlock”, “barrier” and “read-write locking”).
Appendix E. Adding schedulers: RR and EDF

Priority scheduler is most basic scheduler widely used in embedded and real-time systems. Benu in its core implements priority scheduler with FIFO queues for ready threads with same priority. However, this scheduler can be extend with support for other schedulers. This appendix present an idea and its implementation in Benu with round-robin and earliest-deadline-first schedulers as examples.