Simulation of Embedded Micro-Kernels over Pthreads

Thomas Röblitz, Oliver Bühn and Frank Mueller
Humboldt University Berlin, Institute f. Informatik, 10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de phone: (+49) (30) 2093-3011 fax:-3010

Abstract

This work describes the design and implementation of a simulation environment for an open-source embedded micro-kernel and an intuitive user interface to complement it. The study stresses the suitability of POSIX Threads (Pthreads) to resemble micro-kernel operations in the simulation environment. It specifies the prerequisites for using Pthreads as a means to resemble embedded task execution and suggests an I/O-based representation of device information. The experience gained with a sample implementation stresses the importance of a proper match between a Pthreads implementation and an embedded micro-kernel. It also shows the adequacy of both the simulation environment and a graphical user interface to aid program development and debugging. Furthermore, the separation of the simulation component from the user interface provides opportunities to utilize each component separately or even combine them with other components.

1. Introduction

Recently, embedded systems have increased rapidly in terms of the number of currently deployed systems as well as projections for future development [3]. These systems are characterized by low power processors to reduce operating costs with bus width between 8 and 32 bits today. The customized environment surrounding embedded processors and their operating systems generally requires a cross-platform development. However, it can be very tedious to perform cross-platform testing and debugging, which often contributes 50% of the effort in the software development cycle [2]. In such an environment, simulation tools for the embedded system allow rapid prototyping and software pre-validation. In particular, the debugging and testing phase during the software development cycle is facilitated by simulation environments. Logical program errors and runtime errors can be detected more conveniently by simulation.

This work describes the design and implementation of a simulation environment that closely resembles the operations of an embedded micro-kernel operating system. The design methodology is demonstrated by replacing the tasking structure of a micro-kernel by POSIX Threads (Pthreads) [10] on one hand, and by redirecting I/O devices to standard I/O operations, on the other hand. This approach is realized for the open-source LegOS micro-kernel [8] by implementing a simulation environment through changes in the micro-kernel sources in a systematic manner.

The resulting simulation environment provides a testbed for embedded applications that can thereby readily be executed on a standard workstation with POSIX support within or above its operating system. Input and output from the embedded simulation is then coupled with a graphical user interface that visually resembles the embedded environment, depicts output information of actuators and accepts input values for sensory devices. Alternatively, the simulated application could be coupled with a hardware simulator that resembles the physical characteristics and feed-back actions of the embedded device in a physical environment. Such an approach allows additional testing of the embedded software, *e.g.*, prior to the availability of actual physical devices but also during a later phase where a set of simulated units can be connected with each other or even with a set of real units.

Simulation environments for embedded micro-kernel exist for a number of systems, *e.g.*, WindRiver supplies pRISM+ for host-based development of embedded systems. However, the design of embedded simulation environments has not been studied much. Simulation for operating systems [9] or even processors [4, 1, 6] has received much more attention but also require considerable overhead in terms of simulation time. This paper focuses on fast simulations of small micro-kernels that can be achieved by modifying the source code of the kernel to replace tasking and device handling with facilities available under common operating systems of workstations.

The paper is structured as follows. Section 2 identifies design issues and requirements for the approach. Section 3 describes an implementation along these guidelines and gives insight to the experience gained during the implementation. Section 4 describes a user interface and presents the concepts used in this work for combining different simulation components. Section 5 summarizes the work.

2. Micro-Kernel Requirements

A simulation of micro-kernel operations requires that tasking and scheduling within the embedded operating system resemble that of the simulation system. In this paper, we assume a micro-kernel with the characteristics listed in the following.

Multi-Tasking allows concurrent execution of different threads of control.

Multiple Priority Levels allow the selection a either a fixed or a dynamically changed discrete priority value to a task.

Strict Priority Scheduling ensures that a highest priority task executes at any time.

Event-based Activations resume task execution after suspension under a certain condition.

Preemption interrupts the execution of a lower priority task upon reception of an event of a higher priority task.

For each characteristic we identify a corresponding feature of Pthreads. The scheduling model of Pthreads also supports concurrent task execution with preemptive and strict priority scheduling. The number of priorities of the micro-kernel should not exceed the number of priority levels of a Pthreads implementation. Suspension and activations can be modeled though conditional wait and wakeup.

Discrepancies between the different models can also be bridged in many cases. For example, a non-preemptive kernel can be resembled in Pthreads by a monitor lock that is released and then acquired again at each synchronization point within the micro-kernel.

When reusing the original kernel source for the simulator, additional changes to the ported micro-kernel sources are required to provide the proper interfaces for sensory devices and actuators while retaining the original logic for controlling these devices. The algorithms to control I/O ports are replaced by providing corresponding discrete values as standard input or output in conjunction with a characterization of the triggered/probed functionality of a device. These values can then be used by another software component, *e.g.*, the GUI for the embedded device or a simulator of the embedded unit. Our work includes a grammatical specification of the I/O format for values and their functional properties, which provides a documented interface utilized by other simulation components that connect to the application.

Figure 1 depicts a schematic view of a micro-kernel with its different layers. The user is only exposed to the API. Micro-kernels often use a shallow binding layer

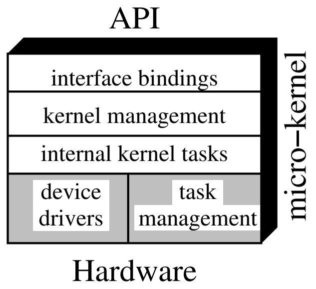

Figure 1: Schematic Structure of a Micro-Kernel

for mapping interface names onto kernel functionality but this layer can be regarded as optional. The kernel functionality is depicted by kernel management routines, for example, to enter and exit the kernel, handle internal kernel data and possibly provide intra-kernel message passing. Micro-kernels often employ kernel tasks dedicated to certain services, which can be low-level devices, abstract file systems, etc. These kernel tasks are scheduled and managed together with user tasks by a tasking layer. Finally, device drivers provide access to hardware components. In the approach described above, only the shaded layers are subject to changes within the micro-kernel source, namely the device drivers and the task management. The following section discusses details of required modifications for a sample implementation.

3. Sample Implementation

We chose the open-source micro-kernel LegOS to perform a case study. LegOS is a micro-kernel for the Lego Mindstorm RCX brick containing a Hitachi 8 bit processor (H8/3292). It replaces the standard firmware of the RCX to obtain full control over the hardware and lifts constraints in programming present in the standard firmware. The LegOS micro-kernel provides the features listed in the previous section, namely preemptive, strict priority scheduling of tasks in an event-triggered system.

The target platform was resticted to arbitrary operating systems adhering to the Pthreads standard. Specifically, FSU Pthreads [7] was used under Linux and SunOS, as well as LinuxThreads [5] under Linux. Different mixes of operating systems and Pthreads implementations were intentionally chosen to evaluate the impact of each component on the simulation environment.

The functionality of the micro-kernel was mapped onto the proper Pthreads and I/O functionality according to the following grouping:

- Initialization of the micro kernel was added as a prerequisite for executing the simulated application. This includes initialization of internals of the micro-kernel as well as initializing the middleware, as required by FSU Pthreads.

- Task execution was mapped onto threads in a straight-forward manner, including task cancellation (see Figure 2). Internal micro-kernel tasks, such as a task manager for LegOS, were resembled as high priority threads. The task manager is responsible for scheduling decisions and event handling.
- Voluntary suspension was implemented either through the corresponding sleep functionality of the operating system or, for event-based waits, by suspension on a condition variable.
- Event handling, realized through the execution of an event handler and a subsequent test of its return value to resume tasks upon a match of values, required a conditional signal upon the given match of values after handler execution.
- Device-specific directives retained their interface and simply send information to standard output (actuator) or receive values from standard input (sensors), also depicted in Figure 2. An internal observer thread was added that receives input according to the grammatical specification and triggers (signals) corresponding events, as described above.

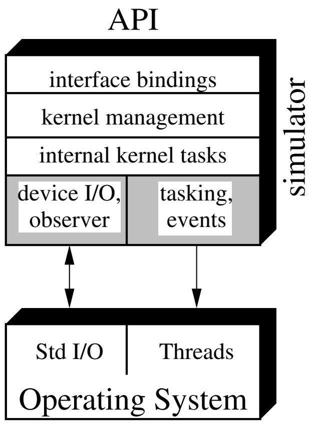

Figure 2: Schematic Structure of the Simulator

The details of event handling and the observer task are depicted by the pseudo code in Figure 3. When a task awaits an event, it supplies a handler and a parameter for the handler. If an initial call to the handler indicates that no data is pending (represented by the discrete value 0 in this example), then the calling task is suspended in a conditional wait. In addition, cleanup handlers have to be used around the wait to ensure that lock are released if a waiting task were canceled (during the conditional wait, which is a cancellation point). This ensures that resources cannot be monopolized when a task is killed, which would cause deadlocks otherwise. The observer task selects a handler depending on the input (not depicted) and calls it with the same parameter as before. If data is pending on the sensory device, the value is supplied to the waiting task and the task is signalled to resume execution, which includes that the sensory value is returned. Notice that we assume that a sensory device only be accessed by one user task. If multiple tasks were competing for a resource, then the await function would have to be augmented by a check for a race condition between tasks to ensure strict FIFO servicing of the requesters.

```
FUNC await_event(handler, data) RETURN int IS
  result := handler(data);
  IF result = 0 THEN
    pthread_cleanup_push(&pthread_mutex_unlock, L);
    pthread_mutex_lock(L);
    global_data := data;
    global_wakeup := FALSE;
    WHILE ¬ global_wakeup DO
      pthread_cond_wait(CondVar, L);
    ENDWHILE;
    result := global_data;
    pthread_mutex_unlock(L);
    pthread_cleanup_pop(0);
  ENDIF;
  RETURN result;
ENDPROC;

TASK observer IS
  FOREVER DO
    input := read();
    result := handler(global_data);
    IF result ≠ 0 THEN
      pthread_mutex_lock(L);
      global_wakeup := TRUE;
      global_data := result;
      pthread_mutex_unlock(L);
      pthread_cond_signal(CondVar);
    ENDIF;
  ENDFOR;
ENDTASK;
```

Figure 3: Simulated Event Handling

These modifications of the micro-kernel allowed the execution of arbitrary applications in the simulation environment without significant changes of the application source code. In fact, the only changes consisted of an initialization call at program start, the specification of an interface file for the simulation software and the resolution of one name conflict (since embedded API names have to be distinct from the API of the host's operating system).

It was then observed that the implementation over FSU Pthreads achieved a complete resemblance of the tasking properties with respect to scheduling actions as compared to the micro-kernel, regardless of the underlying operating system. The implementation over LinuxThreads, on the other hand, provided only partial resemblance of the tasking behavior. LinuxThreads implements kernel threads that are mapped onto processes (a thread *is* a process) with a common address space and schedules these threads in the same manner as processes, *i.e.*, it provides round-robin time-slicing rather than strict priority scheduling. This resulted in different scheduling actions, possibly even violating valid assumptions made throughout the design of an embedded application. This may cause an application to fail to properly executed in the simulation environment, which was otherwise completely valid due to the task interleaving imposed by the given priorities. On one hand, this problem could be fixed by either using real-time priorities or kernel modifications within Linux. However, the former requires additional system privileges (to execute a process within the real-time priority range) and may potentially be harmful if the program is faulty. The latter may only be possible with a different kernel, such as RT-Linux, which imposes unnecessary restrictions on the development platform. In summary, a Pthreads implementation with the proper features, such as FSU Pthreads, proved to be superior for simulation purposes of embedded kernels.

4. User Interface and Working Experience

The user interface developed for the simulation environment is based on a pictorial representation of the embedded unit, the RCX, enhanced by an interface for user interactions. A picture of the embedded unit provides an intuitive representation of the actual system. It also allows a straight-forward association between I/O information and the state of the unit. A Java Applet was chosen as a portable and easily configurable means to display a picture of the RCX. It provides a graphical user interface through the AWT classes. Figure 4 depicts a snapshot of the GUI. The LCD of the RCX interactively outputs messages ("fwwd") generated by the application. Buttons on the RCX adjacent to the LCD (*e.g.,* on/off, run) can be pressed to control the base state of the application. The black squares above the LCD (labeled 1 through 3) can be pressed to simulate a touch sensor. Other sensory input is provided through the empty panels right above that allow the user to enter discrete values. On very top,

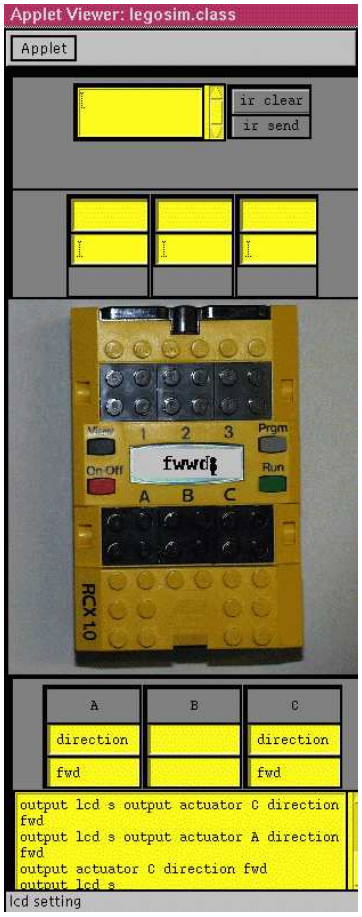

Figure 4: Sample GUI for an Embedded Application

infrared messages are displayed or can be entered and then sent. Below the RCX, the panels (labeled A, B, C) depict the state of the actuators (motors). Finally, a debugging window on the bottom summarizes all device activity in the temporal order, *i.e.*, in the order of message arrival or message generation, respectively.

A number of test programs have been used to validate the functionality of the user interface as well as the simulation environment. These tests have shown the adequacy of the interface to analyze program states, find logical errors in the program and simplify application development. In addition, the simulation environment facilitated the detection of simple errors, such as null pointer references, which would not have been easily detected on the actual hardware since the embedded processor lacks an MMU. In this environment, regular source-level debuggers could be used to probe the state of the simulated application.

The user interface adheres to the same grammatical specification as the simulation component. It accepts actuator output and sends sensory input in an exchange of messages over standard I/O with the simulated application. A simple protocol was used to connect the standard I/O components to each other. Figure 5 depicts how actuator output (on the left) from the application passes through a network interface before being accepted as input at the GUI. Sensor information travels the opposite direction through the same layers.

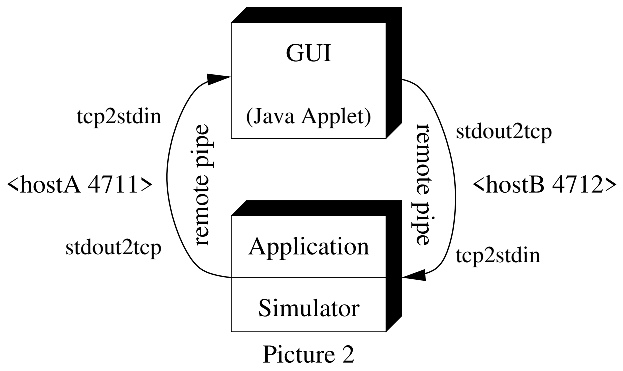

Figure 5: Combining Simulation Components

Combining components through (possibly remote) pipes in such a fashion enhances the flexibility of the simulation approach. This user interface is just one example of how the simulation environment can be used. Another example would be a set of simulation units combined by linking their I/O to each other. This could be used to resemble a broadcast medium, such as the infrared communication built into the RCX. An alternative would be to link simulation units to actual units for testing purposes and observe their interactions. Finally, components can be chained to obtain any of the above functionalities while monitoring I/O streams, for example, to visually depicts the states of units, to probe and log messages, or to filter them for debugging purposes.

5. Conclusion

This paper discusses the design issues and implementation details of a simulation environment for an open-source embedded micro-kernel. In addition, an intuitive user interface is provided. This work shows the suitability of Pthreads to resemble micro-kernel operations in the simulation environment. In particular, the prerequisites for using Pthreads as a means to resemble embedded task execution are identified. Furthermore, an I/O-based representation of device information allows other components, such as the user interface, to interact with the simulation environment. The experience gained with a sample implementation stresses the importance of a proper match between a Pthreads implementation and an embedded micro-kernel. It also shows the adequacy of both the simulation environment and a graphical user interface to aid program development and debugging. Furthermore, the separation of the simulation component from the user interface provides opportunities to utilize each component separately or even combine them with other components.

References

[1] D. Barach, J. Kohli, J. Slice, M. Spaulding, R. Bharadhwaj, D. Hudson, C. Neighbors, N. Saxena, and R. Crunk. Halsim – a very fast sparc v9 behavioral model. In *Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 52–58, January 1995.

[2] C. G. Davis. Testing large, real-time software systems. In *Software Testing, Infotech State of the Art Report*, volume 2, pages 85–105, 1979.

[3] F. Gasperoni. Embedded opportunities. In *Ada Europe*, pages 1–13, June 1998.

[4] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[5] Xavier Leroy. The linuxthreads library. *http://pauillac.inria.fr/~xleroy/linuxthreads*, 1996.

[6] Peter S. Magnusson, Fredrik Larsson, Andreas Moestedt, Bengt Werner, Fredrik Dahlgren, Magnus Karlsson, Fredrik Lundholm, Jim Nilsson, Per Stenström, and Håkan Grahn. SimICS/sun4m: A virtual workstation. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 119–130, Berkeley, USA, June 15–19 1998. USENIX Association.

[7] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[8] Markus L. Noga. Legos. *http://www.noga.de/legOS/*, 1999.

[9] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[10] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].