Chapter 13
Embedded Operating Systems

Operating Systems: Internals and Design Principles

Eighth Edition
By William Stallings
Embedded System

- Refers to the use of electronics and software within a product that is designed to perform a dedicated function
  - in many cases, embedded systems are part of a larger system or product
  - antilock braking system in a car would be an example
Figure 13.1  Possible Organization of an Embedded System
Characteristics of Embedded OS

- Real-time operation
- Reactive operation
- Configurability
- I/O device flexibility
- Streamlined protection mechanisms
- Direct use of interrupts
Developing an Embedded OS

Two general approaches:
• take an existing OS and adapt it for the embedded application
• design and implement an OS intended solely for embedded use
An existing commercial OS can be used for an embedded system by adding:
- real time capability
- streamlining operation
- adding necessary functionality

**Advantage:**
- familiar interface

**Disadvantage:**
- not optimized for real-time and embedded applications
Purpose-Built Embedded OS

- Typical characteristics include:
  - fast and lightweight process or thread switch
  - scheduling policy is real time and dispatcher module is part of scheduler
  - small size
  - responds to external interrupts quickly
  - minimizes intervals during which interrupts are disabled
  - provides fixed or variable-sized partitions for memory management
  - provides special sequential files that can accumulate data at a fast rate

Two examples are:
- eCos
- TinyOS
Timing Constraints

To deal with timing constraints, the kernel:

- provides bounded execution time for primitives
- maintains a real-time clock
- provides for special alarms and timeouts
- supports real-time queuing disciplines
- provides primitives to delay processing by a fixed amount of time and to suspend/resume execution
Embedded Linux

- A version of Linux running in an embedded system
- Embedded devices typically require support for a specific set of devices, peripherals, and protocols, depending on the hardware that is present in a given device and the intended purpose of that device
- An embedded Linux distribution is a version of Linux to be customized for the size and hardware constraints of embedded devices
  - includes software packages that support a variety of services and applications on those devices
  - an embedded Linux kernel will be far smaller than an ordinary Linux kernel
A key differentiator between desktop/server and embedded Linux distributions is that desktop and server software is typically compiled on the platform where it will execute.

Embedded Linux distributions are usually compiled on one platform but are intended to be executed on another.

- The software used for this purpose is referred to as a cross-compiler.
Embedded Linux File Systems

- File system must be as small as possible
- Commonly used examples:
  - cramfs
    - a simple read-only file system that is designed to minimize size by maximizing the efficient use of underlying storage
    - files are compressed in units that match the Linux page size
  - squashfs
    - a compressed, read-only file system that was designed for use on low memory or limited storage size environments
  - jffs2
    - a log-based file system that is designed for use on NOR and NAND flash devices with special attention to flash-oriented issues such as wear-leveling
  - ubifs
    - provides better performance on larger flash devices and also supports write caching to provide additional performance improvements
  - yaffs2
    - provides a fast and robust file system for large flash devices
Advantages of using Linux as the basis for an embedded OS include the following:

- **vendor independence**
  - the platform provider is not dependent on a particular vendor to provide needed features and meet deadlines for deployment

- **varied hardware support**
  - Linux support for a wide range of processor architectures and peripheral devices makes it suitable for virtually any embedded system

- **low cost**
  - the use of Linux minimizes cost for development and training

- **open source**
  - the use of Linux provides all of the advantages of open source software
Focus of Android lies in the vertical integration of the Linux kernel and the Android user-space components

Many embedded Linux developers do not consider Android to be an instance of embedded Linux

from the point of view of these developers, a classic embedded device has a fixed function, frozen at the factory
TinyOS

- Streamlines to a very minimal OS for embedded systems
- Core OS requires 400 bytes of code and data memory combined
- Not a real-time OS
- There is no kernel
- There are no processes
- OS doesn’t have a memory allocation system
- Interrupt and exception handling is dependent on the peripheral
- It is completely nonblocking, so there are few explicit synchronization primitives
- Has become a popular approach to implementing wireless sensor network software
Figure 13.2  Typical Wireless Sensor Network Topology
With the tiny distributed sensor application in mind, the following goals were set for TinyOS:

- allow high concurrency
- operate with limited resources
- adapt to hardware evolution
- support a wide range of applications
- support a diverse set of platforms
- be robust
Embedded software systems built with TinyOS consist of a set of modules (called components), each of which performs a simple task and which interface with each other and with hardware in limited and well-defined ways.

The only other software module is the scheduler.

Because there is no kernel there is no actual OS.

The application area of interest is the wireless sensor network (WSN).

Examples of standardized components include:

- single-hop networking
- ad-hoc routing
- power management
- timers
- nonvolatile storage control
module TimerM {
  provides {
    interface StdControl;
    interface Timer;
  }
  uses interface Clock as Clk;
}
...

(a) TimerM component

(b) TimerC configuration

Figure 13.3  Example Component and Configuration
A software component implements one or more tasks

Each *task* in a component is similar to a thread in an ordinary OS

Within a component tasks are atomic
- once a task has started it runs to completion

<table>
<thead>
<tr>
<th>A task cannot:</th>
<th>A task can:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• be preempted by another task in the same component and there is no time slicing</td>
<td>• perform computations</td>
</tr>
<tr>
<td>• block or spin wait</td>
<td>• call lower-level components (commands)</td>
</tr>
<tr>
<td></td>
<td>• signal higher-level events</td>
</tr>
<tr>
<td></td>
<td>• schedule other tasks</td>
</tr>
</tbody>
</table>
A command is a nonblocking request
- a task that issues a command does not block or spin wait for a reply from the lower-level component

Is typically a request for the lower-level component to perform some service

The effect on the component that receives the command is specific to the command given and the task required to satisfy the command

A command cannot preempt the currently running task

A command does not cause a preemption in the called component and does not cause blocking in the calling component
Components -- Events

- *Events* in TinyOS may be tied either directly or indirectly to hardware events.

- Lowest-level software components interface directly to hardware interrupts:
  - may be external interrupts, timer events, or counter events.

- An event handler in a lowest-level component may handle the interrupt itself or may propagate event messages up through the component hierarchy.

- A command can post a task that will signal an event in the future:
  - in this case there is no tie of any kind to a hardware event.
configuration TimerC {
    provides {
        interface StdControl;
        interface Timer;
    }
}

implementation {
    components TimerM, HWClock;
    StdControl = TimerM.StdControl;
    Timer = TimerM.Timer;
    TimerM.Clk -> HWClock.Clock;
}
TinyOS Scheduler

- Operates across all components
- Only one task executes at a time
- The scheduler is a separate component
  - it is the one portion of TinyOS that must be present in any system
- Default scheduler is a simple FIFO queue
- Scheduler is power aware
  - puts processor to sleep when there is no task in the queue
Figure 13.4 Example TinyOS Application

LED = light-emitting diode
ADC = analog-to-digital converter
TinyOS Resource Interface

TinyOS provides a simple but powerful set of conventions for dealing with resources

**Dedicated**
- a resource that a subsystem needs exclusive access to at all times
- no sharing policy is needed
- examples include interrupts and counters

**Virtualized**
- every client of a virtualized resource interacts with it as if it were a dedicated resource
- an example is a clock or timer

**Shared**
- abstraction that provides access to a dedicated resource through an arbiter component
- arbiter determines which client has access to the resource at which time
Figure 13.5  Shared Resource Configuration
Summary

- Embedded systems
- Characteristics of embedded operating systems
  - Adapting an existing commercial operating system
  - Purpose-built embedded operating system
- Embedded Linux
  - Kernel size
  - Compilation
  - Embedded Linux file systems
  - Advantages of embedded Linux
  - Android

- TinyOS
  - Wireless sensor networks
  - TinyOS goals
  - TinyOS components
  - TinyOS scheduler
  - TinyOS resource interface