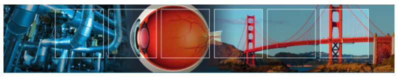
Chapter 4: Operating Systems



Outline

- Functional Aspects
  - Data Types
  - Scheduling
  - Stacks
  - System Calls
  - Handling Interrupts
  - Multithreading
  - Thread-based vs. Event-based Programming
  - Memory Allocation
- Non-Functional Aspects
  - Separation of Concern
  - System Overhead
  - Portability
  - Dynamic Reprogramming
- Prototypes
  - TinyOS
  - SOS
  - Contiki
  - LiteOS
- Evaluation

Operating Systems

- An operating System is
  - a thin software layer
  - resides between the hardware and the application layer
  - provides basic programming abstractions to application developers
- Its *main task* is to enable applications to interact with hardware resources

Operating Systems

- Operating systems are classified as: single-task/ multitasking and single-user/multiuser operating systems
  - multi-tasking OS - the overhead of concurrent processing because of the limited resources
  - single task OS - tasks should have a short duration
- The choice of a particular OS depends on several factors; typically *functional* and *non-functional* aspects

Outline

- ***Functional Aspects***
  - ***Data Types***
  - ***Scheduling***
  - ***Stacks***
  - ***System Calls***
  - ***Handling Interrupts***
  - ***Multithreading***
  - ***Thread-based vs. Event-based Programming***
  - ***Memory Allocation***
- Non-Functional Aspects
  - Separation of Concern
  - System Overhead
  - Portability
  - Dynamic Reprogramming
- Prototypes
  - TinyOS
  - SOS
  - Contiki
  - LiteOS
- Evaluation

Data Types

- Interactions between the different subsystems take place through:
  - well-formulated protocols
  - data types
- Complex data types have strong expression power but consume resources - struct and enum
- Simple data types are resource efficient but have limited expression capability - C programming language

Scheduling

- Two scheduling mechanisms:
  - queuing-based scheduling
    - FIFO - the simplest and has minimum system overhead, but treats tasks unfairly
    - sorted queue - e.g., shortest job first (SJF) - incurs system overhead (to estimate execution duration)
  - round-robin scheduling
    - a time sharing scheduling technique
    - several tasks can be processed concurrently

Scheduling

- Regardless of how tasks are executed, a scheduler can be either
  - a non-preemptive scheduler - a task is executed to the end, may not be interrupted by another task
  - or preemptive scheduler - a task of higher priority may interrupt a task of low priority

Stacks & System Calls

- *Stacks*
  - a data structure that temporarily stores data objects in memory by piling one upon another
  - objects are accessed using last-in-first-out (LIFO)
- *System Calls*
  - decouple the concern of accessing hardware resources from implementation details
  - whenever users wish to access a hardware resource, they invoke these operations without the need to concern themselves how the hardware is accessed

Handling Interrupts

- An interrupt is an asynchronous signal generated by
  - a hardware device
  - several system events
  - OS itself
- An interrupt causes:
  - the processor to interrupt executing the present instruction
  - to call for an appropriate interrupt handler
- Interrupt signals can have different priority levels, a high priority interrupt can interrupt a low level interrupt
- Interrupt mask: let programs choose whether or not they wish to be interrupted

Multi-threading

- A *thread* is the path taken by a processor or a program during its execution
- *Multi-threading* - a task is divided into several logical pieces
  - scheduled independent from each other
  - executed concurrently
- Two advantages of a multi-threaded OS:
  1. tasks do not block other tasks
  2. short-duration tasks can be executed along with long-duration tasks

Multi-threading

- Threads cannot be created endlessly
  - the creation of threads *slows down* the processor
  - no sufficient resources to divide
- The OS can keep the number of threads to a *manageable size* using a thread pool

Thread-based vs. Event-based Programming

- Decision whether to use threads or events programming:
  - need for separate stacks
  - need to estimate maximum size for saving context information
- *Thread-based programs* use multiple threads of control within:
  - a single program
  - a single address space

Thread-based vs. Event-based Programming

- *Advantage:*
  - a thread blocked can be suspended while other tasks are executed in different threads
- *Disadvantages:*
  - must carefully protect shared data structures with locks
  - use condition variables to coordinate the execution of threads

Thread-based vs. Event-based Programming

- In *event-based programming*: use events and event handlers
  - event-handlers register with the OS scheduler to be notified when a named event occurs
  - a loop function:
    - polls for events
    - calls the appropriate event-handlers when events occur
- An event is processed to completion
  - unless its handler reaches at a blocking operation (callback and returns control to the scheduler)

Memory Allocation

- The memory unit is a precious resource
- Reading and writing to memory is costly
- How and for how long a memory is allocated for a piece of program determines the speed of task execution

Memory Allocation

- Memory can be allocated to a program:
  - *statically* - a frugal approach, but the requirement of memory *must be known* in advance
    - memory is used efficiently
    - runtime adaptation is not allowed
  - *dynamically* - the requirement of memory is *not known* in advance (on a transient basis)
    - enables flexibility in programming
    - but produces a considerable management overhead

Outline

Separation of Concern

- In general, separation between the operating system and the applications layer
- The operation systems can provide:
  - a number of lightweight modules - “wired” together, or
  - an indivisible system kernel + a set of library components for building an application, or
  - a kernel + a set of reconfigurable low-level services
- Separation of concern enables:
  - flexible and efficient reprogramming and reconfiguration

Portability

- Ideally, operating systems should be able to co-exist and collaborate with each other
- However, existing operating systems do *not* provide this type of support
- In order to accommodate unforeseen requirements, operating systems should be portable and extensible

System Overhead

- An operating system executes program code - requires its own share of resources
- The resources consumed by the OS are the system’s overhead, it depends on
  - the size of the operating system
  - the type of services that the OS provides to the higher-level services and applications

System Overhead

- The resources of wireless sensor nodes have to be shared by programs that carry out:
  - sensing
  - data aggregation
  - self-organization
  - network management
  - network communication

Dynamic Reprogramming

- Once a wireless sensor network is deployed, it may be necessary to reprogram some part of the application or the operating system for the following reasons:
  1. the network may not perform optimally
  2. both the application requirements and the network's operating environment can change over time
  3. may be necessary to detect and fix bugs

Dynamic Reprogramming

- Manual replacement may not be feasible - develop an operating system to provide dynamic reprogramming support, which depends on
  - clear separation between the application and the OS
  - the OS can receive software updates and assemble and store it in memory
  - OS should make sure that this is indeed an updated version
  - OS can remove the piece of software that should be updated and install and configure the new version
  - all these consume resources and may cause their own bugs

Dynamic Reprogramming

- Software reprogramming (update) requires robust *code dissemination protocols*:
  - splitting and compressing the code
  - ensuring code consistency and version controlling
  - providing a robust dissemination strategy to deliver the code over a wireless link

Outline

- Functional Aspects
  - Data Types
  - Scheduling
  - Stacks
  - System Calls
  - Handling Interrupts
  - Multithreading
  - Thread-based vs. Event-based Programming
  - Memory Allocation
- Non-Functional Aspects
  - Separation of Concern
  - System Overhead
  - Portability
  - Dynamic Reprogramming
- ***Prototypes***
  - ***TinyOS***
  - SOS
  - Contiki
  - LiteOS
- Evaluation

TinyOS (Gay et al. 2007)

- TinyOS is *the most widely used*, *richly documented*, and *tool-assisted* runtime environment in WSN
  - static memory allocation
  - event-based system
- TinyOS's architecture consists of
  - a scheduler
  - a set of components, which are classified into
    - configuration components - "wiring" (how models are connected with each other)
    - modules - the basic building blocks of a TinyOS program

TinyOS (Gay et al. 2007)

- A component is made up of
  - a frame
  - command handlers
  - event handlers
  - a set of non-preemptive tasks
- A component is similar to an object in object-based programming languages:
  - it encapsulates state and interacts through well-defined interfaces
  - an interface that can define commands, event handlers, and tasks

TinyOS (Gay et al. 2007)

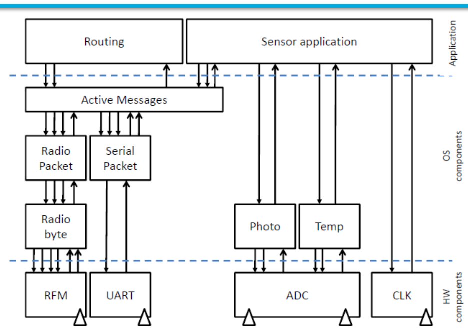


Figure 4.1 Logical distinction between low-level and high-level components (Hill et al. 2000)

TinyOS (Gay et al. 2007)

- Components are structured hierarchically and communicate with each other through commands and events:
  - higher-level components issue commands to lower-level components
  - lower-level components signal events to higher-level components
- In Figure 4.1, two components at the highest level communicate asynchronously through active messages
  - routing component - establishing and maintaining the network
  - sensor application - responsible for sensing and processing

TinyOS (Gay et al. 2007)

- The logical structure of components and component configurations

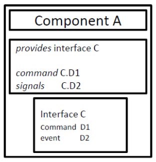


Figure 4.2
A TinyOS component providing an interface

In Figure 4.2, *Component A* declares its service by providing *interface C*, which in turn provides *command D1* and signals *event D2*.

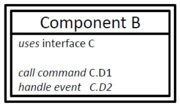


Figure 4.3
A TinyOS components that uses an interface

In Figure 4.3, *Component B* expresses interest in *interface C* by declaring a call to *command D1* and by providing an event handler to process *event D2*.

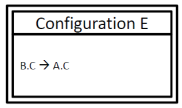


Figure 4.4
A TinyOS configuration that wires an interface provider and an interface user

In Figure 4.4, a binding between *Component A* and *Component B* is established through the *Configuration E*.

Tasks, Commands and Events

- The fundamental building blocks of a TinyOS runtime environment: *tasks, commands,* and *events*
  - enabling effective communication between the components of a single frame
- *Tasks :*
  - *monolithic processes* - should execute to completion - they cannot be preempted by other tasks, though they can be interrupted by events
  - possible to allocate a single stack to store context information
  - call lower level commands; signal higher level events; and post (schedule) other tasks
  - scheduled based on FIFO principle (in TinyOS)

Tasks, Commands and Events

- *Commands:*
  - non-blocking requests made by higher-level components to lower-level components
  - split-phase operation:
    - a function call returns immediately
    - the called function notifies the caller when the task is completed
- *Events:*
  - events are processed by the event handler
  - event handlers are called when hardware events occur
  - an event handler may react to the occurrence of an event in different ways
    - deposit information into its frame, post tasks, signal higher level events, or call lower level commands

Outline

- Functional Aspects
  - Data Types
  - Scheduling
  - Stacks
  - System Calls
  - Handling Interrupts
  - Multithreading
  - Thread-based vs. Event-based Programming
  - Memory Allocation
- Non-Functional Aspects
  - Separation of Concern
  - System Overhead
  - Portability
  - Dynamic Reprogramming
- ***Prototypes***
  - TinyOS
  - ***SOS***
  - Contiki
  - LiteOS
- Evaluation

SOS (Han et al. 2005)

- The SOS operating system (Han et al. 2005)
  - establishes a balance between flexibility and resource efficiency
  - supports runtime reconfiguration and reprogramming
- The SOS operating system consists of:
  - a kernel :
    - provides interfaces to the underlying hardware
    - provides priority-based scheduling mechanism
    - supports dynamic memory allocation
  - a set of modules – can be loaded and unloaded - a position independent binary
    - enables SOS to dynamically link modules with each other

Interaction

- Interaction with a module through:
  1. *messages* (asynchronous communication)
     - a message that originates from module A to module B
       - the message goes through the scheduler
       - the kernel calls the appropriate message handler in module B and passes the message to it
  2. *direct calls to registered functions* (synchronous communication)
     - requires modules to register their public functions at the kernel - all modules can subscribe to these functions
       - the kernel creates a *function control block (FCB)* to store key information about the function
       - this information is used to:
         - handle function subscription
         - support dynamic memory management
         - support runtime module update (replacement)

Interaction

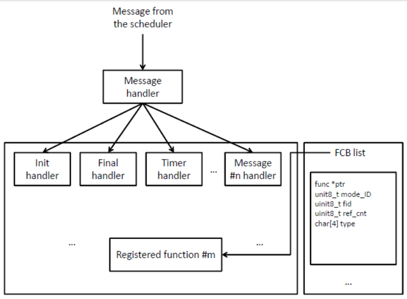


Figure 4.5 illustrates the two basic types of interactions between modules

- Interaction through a function call is faster than message-based communication

Dynamic Reprogramming

- *Five basic features* enable SOS to support dynamic reprogramming
  - modules are *position independent binaries*
    - they use relative addresses rather than absolute addresses ---- they are re-locatable
  - every SOS module implements two types of handlers – the *init* and *final* message handlers
    - the *init* message handler - to set the module's initial state
    - the *final* message handler - to release all resources the module owns and to enable the module to exit the system gracefully
    - after the *final* message, the kernel performs garbage collection

Dynamic Reprogramming

- SOS *uses a linker script* to place the *init* handler of a module at a known offset in the binary
  - enables easy linking during module insertion
- SOS keeps the state of a module *outside* of it
  - enables the newly inserted module to inherit the state information of the module it replaces
- Whenever a module is inserted, SOS generates and *keeps* metadata that *contains information*:
  - the ID of the module
  - the absolute address of the *init* handler
  - a pointer to the dynamic memory holding the module state

Dynamic Reprogramming

- In SOS, dynamic module replacement (update) takes place in *three steps*:
  1. a code distribution protocol *advertises* the new module in the network
  2. the protocol *proceeds* with downloading the module and examines the metadata
     - the metadata contains the size of the memory required to *store the local state of the module*
     - if a node does not have sufficient *RAM,* module insertion is immediately *aborted*
  3. if everything is correct, module insertion takes place and the kernel invokes the handler by *scheduling* an *init* message for the module

Outline

- Functional Aspects
  - Data Types
  - Scheduling
  - Stacks
  - System Calls
  - Handling Interrupts
  - Multithreading
  - Thread-based vs. Event-based Programming
  - Memory Allocation
- Non-Functional Aspects
  - Separation of Concern
  - System Overhead
  - Portability
  - Dynamic Reprogramming
- ***Prototypes***
  - TinyOS
  - SOS
  - ***Contiki***
  - LiteOS
- Evaluation

Contiki (Dunkels et al. 2004)

- Contiki is a hybrid operating system
  - an event-driven kernel but multi-threading with a dynamic linking strategy
  - separate the kernel from processes
  - communication of services through the kernel by posting events
  - the kernel does not provide hardware abstraction
  - device drivers and applications communicate directly with the hardware
  - the kernel is easy to reprogram and it is easy to replace services

Contiki (Dunkels et al. 2004)

- For each SOS service:
  - it manages its own state in a private memory
  - the kernel keeps a pointer to the process state
  - it shares with other services the same address space
  - it implements an event handler and an optional poll handler

Contiki (Dunkels et al. 2004)

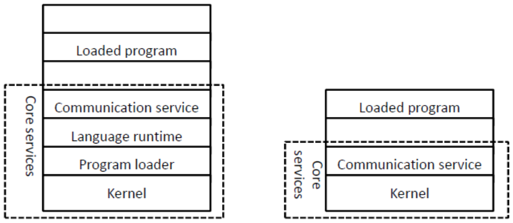


Figure 4.6 The Contiki operating system: the system programs are partitioned into core services and loaded programs

Contiki (Dunkels et al. 2004)

- Figure 4.6 illustrates Contiki's memory assignment in ROM and RAM
- Basic assignment:
  - dispatch events
    - synchronous events
    - asynchronous events
  - periodically call polling handlers
    - the status of hardware components is sampled periodically

Service Structure

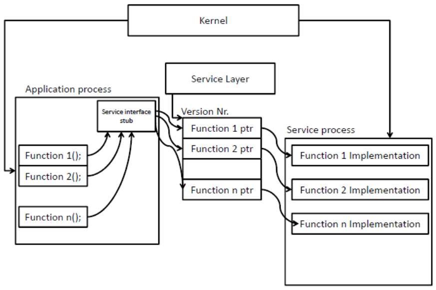


Figure 4.7 A Contiki service interaction architecture (Dunkels et al. 2004)

Service Structure

- Figure 4.7 illustrates how application programs interact with Contiki services
- Contiki OS supports
  - dynamic loading
  - reconfiguration of services
- This is achieved by defining
  - services
  - service interfaces
  - service stubs
  - service layers

Outline

- Functional Aspects
  - Data Types
  - Scheduling
  - Stacks
  - System Calls
  - Handling Interrupts
  - Multithreading
  - Thread-based vs. Event-based Programming
  - Memory Allocation
- Non-Functional Aspects
  - Separation of Concern
  - System Overhead
  - Portability
  - Dynamic Reprogramming
- ***Prototypes***
  - TinyOS
  - SOS
  - Contiki
  - ***LiteOS***
- Evaluation

LiteOS (Cao et al. 2008)

- LiteOS is a *thread-based* operating system and supports *multiple applications*
  - based on the principle of *clean separation* between the OS and the applications
  - *does not* provide components or modules that should be “wired” together
  - provides several *system calls*
  - provides *a shell* - isolates the system calls from a user
  - provides a hierarchical *file management system*
  - provides a dynamic *reprogramming* technique

LiteOS (Cao et al. 2008)

- LiteOS is modeled as a distributed file system

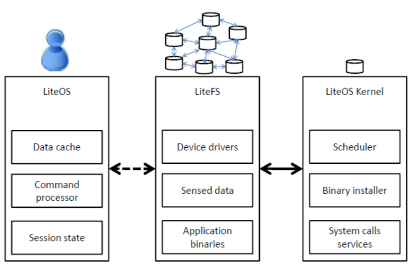


Figure 4.8 The LiteOS operating system architecture (Cao et al. 2008)

Shell and System Calls

- The shell provides:
  - a *mounting mechanism* to a wireless node which is one-hop away from it
    - a distributed and hierarchical file system
    - a user can access the resources of a named node
  - a large number of *Linux commands*
    - file commands - move, copy and, delete files and directories
    - process commands - manage threads
    - debugging commands - set up a debugging environment and debug code
    - environment commands
      - user - managing the environment of OS
      - manual - displaying interaction history and providing command reference
    - device commands - provide direct access to hardware devices

LiteFS

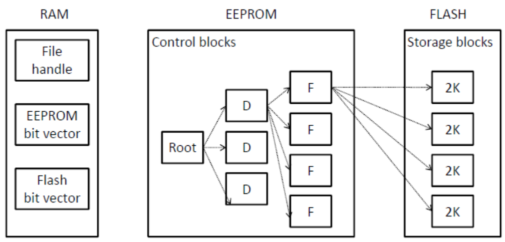


Figure 4.9 The file system structure of LiteFS (Cao et al. 2008)

Dynamic Reprogramming

- The LiteFS is a *distributed file system*
- A user can
  - access the entire sensor network
  - program and manage individual nodes
- LiteOS supports the *dynamic replacement* and *reprogramming* of user applications
  - if the original source code is available to the OS
    - recompiled with a new memory setting
    - the old version will be redirected

Dynamic Reprogramming

- If the original source code is not available to the OS
  - use a *differential patching* mechanism to replace an older version binary
    - the start address (S) of the binary executable in the flash memory
    - the start address of allocated memory in RAM (M)
    - the stack top (T)
    - T - M = the memory space allocated for the program code
  - but the parameters are obtained empirically and require knowledge of the node architecture - limits the usefulness of the patching scheme

Outline

- Functional Aspects
  - Data Types
  - Scheduling
  - Stacks
  - System Calls
  - Handling Interrupts
  - Multithreading
  - Thread-based vs. Event-based Programming
  - Memory Allocation
- Non-Functional Aspects
  - Separation of Concern
  - System Overhead
  - Portability
  - Dynamic Reprogramming
- Prototypes
  - TinyOS
  - SOS
  - Contiki
  - LiteOS
- ***Evaluation***

Evaluation

OS	Programming Paradigm	Building Blocks	Scheduling	Memory Allocation	System Calls
TinyOS	Event-based (split-phase operation, active messages)	Components, interfaces and tasks	FIFO	Static	Not available
SOS	Event-based (Active messages)	Modules and messages	FIFO	Dynamic	Not available
Contiki	Predominantly event-based, but it provides an optional multi-threading support	Services, service interface stubs and service layer	FIFO, poll handlers with priority scheduling	Dynamic	Runtime libraries
LiteOS	Thread-based (based on thread pool)	Applications are independent entities	Priority-based scheduling with an optional Round-robin support	Dynamic	A host of system calls available to the user (file, process, environment, debugging and device commands)

Table 4.1 Comparison of functional aspects of existing operating systems

Evaluation

OS	Minimum System overhead	Separation of Concern	Dynamic reprogramming	Portability
TinyOS	332 Bytes	There is no clean distinction between the OS and the application. At compilation time a particular configuration produces a monolithic, executable code.	Requires external software support	High
SOS	ca. 1163 Byte	Replaceable modules are compiled to produce an executable code. There is no clean distinction between the OS and the application.	Supported	Midium to low
Contiki	ca. 810 Byte	Modules are compiled to produce a reprogrammable and executable code, but there is no separation of concern between the application and the OS.	Supported	Medium
LiteOS	Not available	Application are separate entities; they are developed independent of the OS	Supported	Low

Table 4.2 Comparison of nonfunctional aspects of existing operating systems