# State of the Art in Wireless Sensor Networks Operating Systems: A Survey

Muhammad Omer Farooq[1], Sadia Aziz[2], and Abdul Basit Dogar[3]

[1] Transmission Systems Research Group, Jacobs University Bremen, Germany
[2] Computer Engineering Department, CASE, Islamabad, Pakistan
[3] Department of Computer Science, Virtual University of Pakistan, Lahore, Pakistan
m.farooq@jacobs-university.de, sadia.aziz@case.edu.pk,
abasit126@yahoo.com

**Abstract.** This paper, presents a survey on current state of the art in Wireless Sensor Networks (WSNs) Operating Systems (OSs). WSN is composed of miniature senor and resource constraint devices. WSN is highly dynamic network because nodes die out due to severe environmental conditions and battery power depletion. Stated characteristics of WSN impose additional challenges on OS design for WSN. Therefore; OS design for WSN deviates from traditional OS design. The purpose of this survey is to point out strengths and weaknesses of contemporary OS for WSNs, keeping in view the requirements of emerging WSNs applications. State of the art, in operating systems for WSNs has been examined in terms of Architecture, Scheduling, Threading Model, Synchronization, Memory Management, and Communication Protocol support. The examination of these features is performed for both real time and non real time operating systems for WSNs. We believe that this survey will help the network designers and programmers to choose the right OS for their network and applications. Moreover, pros and cons of different operating systems will help the researchers to design more robust OSs for WSNs.

**Keywords:** Wireless Sensor Networks (WSN), Operating System (OS), Embedded Operating Systems.

## 1 Introduction

Advances in Micro-Electro Mechanical System (MEMS) based sensor technology has led to the development of miniaturized and cheap sensor nodes, capable of wireless communication, sensing and performing computations. Wireless Sensor node is composed of micro-controller, transceiver, timer, memory and analog to digital converter. Figure 1 shows the block diagram of a typical sensor node. Sensor nodes are deployed to monitor multitude of natural and unnatural phenomenon i.e., habitant monitoring, wild life monitoring, patient monitoring, industrial process monitoring, battle field monitoring, traffic control, home automation to name a few. Sensor nodes have constraint resources i.e., small amount of battery, few kilobytes of memory and a microcontroller that operates at very low frequency compared to traditional contemporary processing units. These resource constraint tiny sensors are an example

T.-h. Kim et al. (Eds.): FGIT 2010, LNCS 6485, pp. 616–631, 2010.
© Springer-Verlag Berlin Heidelberg 2010

of System on Chip (SoC). Dense deployment of sensor nodes in sensing field and distributed processing through multi-hop communication among sensor nodes is required to achieve high quality and fault tolerance in WSN. Application areas for sensors are growing and new applications for sensor networks are emerging rapidly.

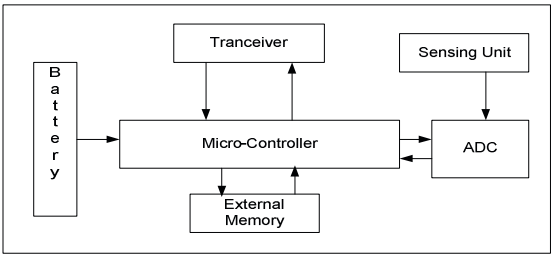


**Fig. 1.** Sensor Node Architecture

OS acts as a resource manager for complex systems. In a typical system these resources include processors, memories, timers, disks, mice, keyboard, network interfaces etc. The job of OS is to manage allocation of these resources to users in an orderly and controlled manner. An OS multiplex system resources in two ways i.e., in time and in space. Time multiplexing involves different programs taking turn in using the resources. Space multiplexing involves different programs getting part of the resource possibly at the same time.

Literature exist that survey the application, transport, network and Medium Access Control (MAC) protocols for WSN, one such effort was made in [11]. A survey on Operating Systems for WSN also exists and published in [10]. Since [10] was published, new features like Architecture, Execution Model, Reprogramming, Scheduling and Power Management have been introduced in contemporary operating systems for WSN, hence the need for this survey remains important.

In this survey, we have examined the core OS features like Architecture, Scheduling, Threading Model, Synchronization, Memory Management, and Communication Protocol Support in both real time and non-real time WSN operating systems. We have discussed different design approaches taken by different WSN operating systems with their relative pros and cons.

Section 2 presents major design concern for WSNs OS. TinyOS has been investigated in Section 3. In Section 4, we investigate the Contiki operating system and Section 5 presents MANTIS operating System. We have identified future research directions in Section 6. Finally, this paper is concluded in Section 7.

## 2 Major Concerns in WSN OS Design

This section, gives details of major issues related to an OS design for WSN.

### 2.1 Architecture

Organization of an OS constitutes its structure. In an OS perspective, architecture of the OS kernel makes up its structure. Architecture of an OS has an influence on the size of the OS kernel as well as on the way it provides services to the application programs. Some of the well known OS architectures are Monolithic Architecture, Micro-Kernel Architecture, Virtual Machine Architecture and Layered Architecture.

A monolithic architecture in-fact does not have any structure. Services provided by an OS are implemented separately and each service provides an interface for other services. Such architecture, allows bundling of all the required service together into a single system image thus, results in larger OS footprint. Advantage of monolithic architecture is that cost of module interaction is low. Disadvantages associated with this architecture are: system is hard to understand, modify, and to maintain. Disadvantages associated with monolithic kernels make them as a bad OS design choice for sensor nodes.

Alternate choice for an OS design is Microkernel architecture. In microkernel minimum functionality is provided inside the kernel. Thus, kernel size is significantly reduced. Most of the OS functionality is provided in user level servers like file server, memory server, time server etc. If one server crashes down whole system does not crash. Microkernel kernel architecture provides better reliability, ease of extension and customization. The disadvantage associated with microkernel is poor performance because of user kernel boundary crossing. Microkernel is the design choice for many embedded OS due to the small kernel size secondly; context switches are far less in embedded systems. Thus, boundary crossing is less compared to traditional systems.

Virtual machine architecture is another design alternative for an OS. Main idea is to export virtual machines to user programs, which resemble hardware. A virtual machine has all hardware features. Advantage is portability. Disadvantage is poor system performance.

Layered OS architecture implements services in the form of layers. Advantages associated with layered architecture are: more manageability, understandability, and reliability. The disadvantage is that it is not flexible.

An OS for Wireless Sensor Network should have an architecture that results in a small kernel footprint. Architecture must allow extensions to the kernel if required. Architecture must be flexible i.e., only application required services get loaded onto the system.

### 2.2 Resource Sharing

Responsibility of an OS includes resources allocation and resource sharing is of immense importance when multiple programs are concurrently executing. Majority of sensor network OS now provide some sort of multithreading therefore, there must be a resource sharing mechanism available. This can be performed in time e.g., scheduling of a process on the CPU and in space e.g., writing data to system memory. In some cases we need a serialized access to resources and this is done through the use of synchronization primitives.

### 2.3 Protection

In traditional operating system, protection refers to protecting of one process from another. In early sensor network operating systems like TinyOS [1] there was no memory management available. Early, operating systems for sensor networks assumed that only a single thread executes on a process therefore; there is no need for memory protection. Latest WSN involves multiple threads of execution therefore; memory management becomes an issue for WSN OS.

### 2.4 Performance

In OS, how we make the system all go fast is called performance. Performance of a system can be measured by throughput, access time, and response time. The ultimate responsibility of an OS designer is to enhance the overall performance of the system, keeping in view the kind of application that runs on the system.

### 2.5 Communication

In OS context, communication refers to inter-process communication within the system as well as with other nodes in the network. Sensor networks operate in a distributive environment therefore; senor nodes communicate with other nodes in the network. The job of sensor network OS is to provide Application Programming Interface (API) that provides easy and energy efficient way of communication. It is possible that sensor network is composed of heterogeneous sensor nodes therefore; communication protocol provided by the OS must also consider heterogeneity. In network based communication, OS should provide transport layer, network layer and MAC layer protocol implementation.

### 2.6 Scheduling

Central Processing Unit (CPU) scheduling determines the order in which tasks are executed on CPU. In traditional computer systems, the goal of a perfect scheduler is to minimize latency, maximize throughput, maximize resource utilization, and fairness.

The type of scheduling algorithm for sensor network typically depends on the nature of the application. For applications having real time requirements we need to use real time scheduling algorithm and for other applications we can use non-real time scheduling algorithms.

Sensor networks are being used in both real time and non-real time phenomenon therefore, a sensor network OS must provide scheduling algorithm that can accommodate the application requirements. Moreover, scheduling algorithm should be memory and energy efficient.

### 2.7 Multithreading

Multithreading provides a convenient application development environment. In threaded systems, context switching and scheduling are the source of major overhead [8]. We know that sensor nodes are battery operated, memory limited and have low

computational power. Therefore, sensor network operating system should support high concurrency with minimal memory usage and low energy consumption.

## 3 TinyOS

TinyOS [1] is an open source flexible component based, and application specific operating system designed for sensor networks. TinyOS can support concurrent programs with very low memory requirements. The OS has footprint that fits in 400 bytes. TinyOS component library includes network protocols, distributed services, sensor drivers, and data acquisition tools. Following subsections survey TinyOS design in more detail.

### 3.1 Architecture

TinyOS fall under the monolithic architecture. TinyOS uses the component model and according to the requirements of an application different components are glued together with the scheduler to compose a static image that runs on the mote. A component is an independent computational entity that exposes one or more interfaces. Components have three computational abstractions: commands, events, and tasks. Mechanisms for inter-component communication are commands and events. Tasks are used to express intra-component concurrency. A command is a request to perform some service, while the event signals the completion of the service.

TinyOS provides single shared stack and there is no separation between kernel space and the user space. For program execution TinyOS uses an event driven model. Following figure shows the TinyOS architecture.

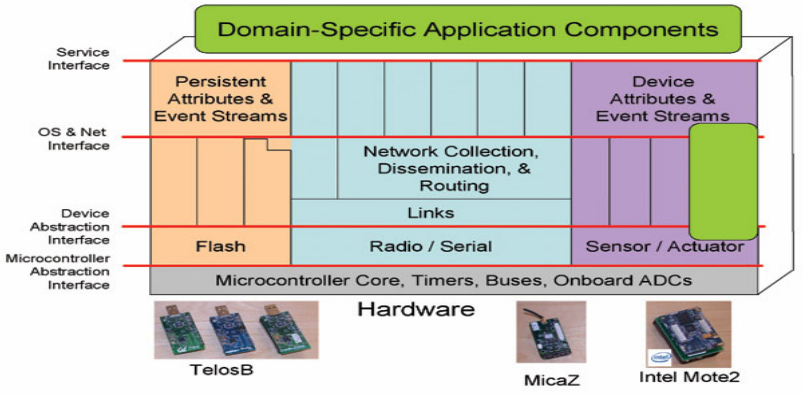


**Fig. 2.** TinyOS Architecture

### 3.2 Scheduling

Earlier versions of TinyOS support non preemptive First In First Out (FIFO) scheduling algorithm. Therefore; those versions of TinyOS does not support real time application. This prevents TinyOS usage in real time systems. The core of the

execution model in TinyOS is task that runs to completion in FIFO manner. Since, TinyOS supports only non preemptive scheduling therefore, task must obey run to completion semantics. Tasks run to completion with respect to other task but they are not atomic with respect to interrupt handlers and commands and events they invoke. Since TinyOS uses FIFO scheduling therefore, disadvantages associated with FIFO scheduling are also associated with the TinyOS scheduler. The wait time for a task depends on the tasks arrival time. FIFO scheduling can be unfair to latter tasks especially when short tasks are waiting behind the longer ones.

In [1], authors have claimed that they have added support for Earliest Deadline First (EDF) scheduling algorithm in TinyOS, to facilitate real time application. EDF scheduling algorithm does not produce a feasible schedule when tasks content for the resources. Thus, TinyOS does not provide a good real time scheduling algorithm if different threads content for resources.

### 3.3 Threading Model and Synchronization

Earlier versions of TinyOS do not provide any multithreading support. TinyOS version 2.1 provides support for multithreading and these TinyOS threads are called TOS Threads. In [3], authors pointed out the problem that given the motes resource constraints, an event based OS permits greater concurrency. However, preemptive threads offer an intuitive programming paradigm. TOS threading package provides ease of a threading programming model with the efficiency of an event driven kernel. TOS threads are backward compatible with existing TinyOS code. TOS threads use cooperative threading approach, i.e., TOS threads rely on applications to explicitly yield the processor. This adds on an additional burden on the programmer to manage the concurrency explicitly. Application level threads in TinyOS can preempt other application level threads but they cannot preempt tasks and interrupt handlers. High priority kernel thread is dedicated to run the TinyOS scheduler. For communication between the application threads and kernel, TinyOS 2.1 provides the mechanism of message passing. When an application program makes a system call, it does not directly execute the code rather it posts a message to the kernel thread by posting a task. Afterwards, kernel thread preempts the running thread and executes the system call. This mechanism ensures that only kernel directly executes TinyOS code. System calls like *Create, Destroy, Pause, Resume and Join* are provided by the TOS threading library.

TOS threads dynamically allocate Thread Control Blocks (TCB) with space for fixed size stack that does not grow over time. TOS Threads context switches and system calls introduce an overhead of less than 0.92% [3].

Earlier versions of TinyOS impose atomicity by disabling the interrupts i.e., telling the hardware to delay handing the external events until aftersystem is done with the atomic operation. This scheme works well on uni-processor systems. Secondly, critical section can occur in the user level threads and the designer of OS does not want user to disable the interrupts due to system performance and usability issues. This problem is circumvented in TinyOS version 2.1. It provides synchronization support with the help of condition variables and mutexes. These synchronization primitives are implemented with the help of special hardware instructions e.g., test & set instruction.

### 3.4 Memory Management and Safety

In [2], efficient memory safety for TinyOS is presented. In sensor nodes, hardware based memory protection is not available and the resources are scarce. Resource constraints necessitate the use of unsafe, low level languages like nesC [17]. In TinyOS version 2.1 memory safety is incorporated. The goals for memory safety as given in [2] are: trap all pointer and array errors, provide useful diagnostics, and provide recovery strategies. Implementations of memory safe TinyOS exploits the concept of Deputy. Deputy is a resource to resource compiler that ensures type and memory safety for C code. Code compiled by Deputy relies on a mix of compile and run time checks to ensure memory safety. Safe TinyOS is backward compatible with earlier version of TinyOS. Safe TinyOS tool chain inserts checks into the application code to ensure safety at run time. When a check detects that safety is about to be violated, code inserted by Safe TinyOS take remedial action.

### 3.5 Communication Protocols Support

Earlier versions of TinyOS use two multi-hop protocols: dissemination [14] and TYMO [15]. Dissemination protocol, reliably delivers data to every node in the network. This protocol enables administrators to reconfigure query and reprogram a network. Dissemination Protocol provides two interfaces: DisseminationValue and DisseminationUpdate. A producer should call the DisseminationUpdate. The command DisseminationUpdate.chage() should be called each time the producers wants to disseminate a new value. On the other hand DisseminationValue is for the consumer. The event DisseminationValue.changed() is signaled each time the dissemination value s changed. TYMO is the implementation of the DYMO protocol, a routing protocol for mobile ad hoc networks. In TYMO, packet formats have changed and it has been implemented on top of the active messaging stack.

K. Lin et al [16], presents DIP a new dissemination protocol for sensor networks. DIP is a data discovery and dissemination protocol that scales to hundreds of values. At MAC layer TinyOS provide implementation of the following protocols: a single hop TDMA protocol, a TDMA/CSMA hybrid protocol which implements Z-MAC's slot stealing optimization, and an optional implementation of 802.15.4 complaint MAC is available.

## 4 Contiki

Contiki [5], is a lightweight open source OS written in C language for WSN sensor nodes. Contiki is highly portable OS and it is build around an event driven kernel. Contiki provides preemptive multitasking that can be used at the individual process level. A typical Contiki configuration consumes 2 kilobytes of RAM and 40 kilobytes of ROM. A full Contiki installation includes features like: multitasking kernel, preemptive multithreading, proto-threads, TCP/IP networking, IPv6, Graphical User Interface, web browser, personal web server, simple telnet client, screensaver, and virtual network computing. In the following subsections, we shall explore Contiki OS in more detail.

### 4.1 Architecture

Contiki OS, follows the hybrid architecture i.e., it combines advantages of events and threads. At the kernel level it follows the event driven model but it provides optional threading facility to individual processes. Contiki kernel comprises of a lightweight event scheduler that dispatches events to the running processes. Process execution is triggered by the events dispatched by the kernel to the processes or by the polling mechanism. Polling mechanism is used to avoid race conditions. Any scheduled event will run to completion however, event handlers can use internal mechanism for preemption.

There are two kinds of events supported by Contiki OS: asynchronous events and synchronous events. The difference between two is that synchronous events are dispatched immediately to the target process that causes it to be scheduled, on the other hand asynchronous events are more like deferred procedure calls that are enqueued and dispatched later to the target process.

Polling mechanism used in Contiki can be seen as high priority events that are scheduled in between each asynchronous event. When a poll is scheduled all processes that implement a poll handler are called in order of their priority.

All OS facilities e.g., senor data handling, communication, and device drivers are provided in the form of services. Each service has its interface and implementation. Application using a particular service needs to know its interface. Application is not concerned about the implementation of a service. Following is the block diagram of Contiki OS architecture, as given in [18].

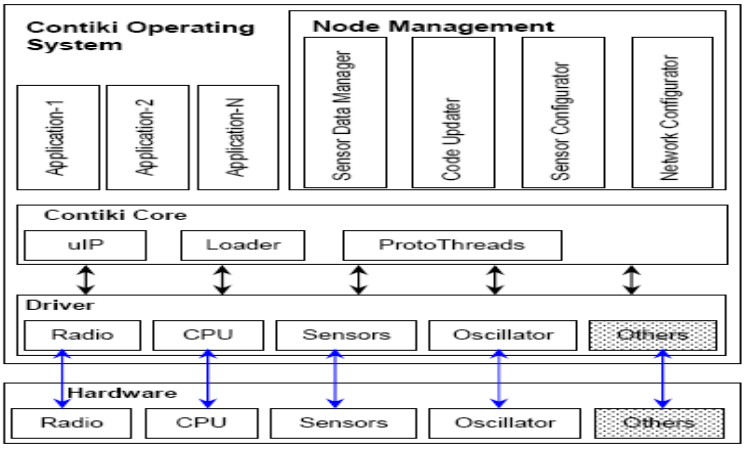


**Fig. 3.** Contiki Architecture [18]

### 4.2 Scheduling

Contiki is an event driven OS therefore, it does not employ any sophisticated scheduling algorithm. Events are fired to the target application as they arrive. In case of interrupts, interrupt handlers of an application runs w.r.t. its priority.

### 4.3 Threading Model and Synchronization

Contiki supports preemptive multithreading. Multi-threading is implemented as a library on top of the event driven kernel. The library can be linked with the applications that require multithreading. Contiki multithreading library is divided in two parts: a platform independent part and a platform specific part. The platform independent part interfaces to the event kernel and the platform specific part of the library implements stack switching and preemption primitives. Since, preemption is supported therefore; preemption is implemented using the timer interrupt and the thread state is stored on a stack. Available threading system calls are: *mt_yield(), mt_post(id,event,dataptr), mt_wait(event,dataptr), mt_exit(), mt_start(thread,funptr,dataptr), mt_exec(thread)* .

For multithreading Contiki uses protothreads [19]. Protothreads are designed for severely memory constraint devices because they are stack less and lightweight. Main features of protothreads are: very small memory overhead only two bytes per protothreads, no extra stack for a thread, highly portable, can be used with or without OS, provides blocking wait without full multithreading and stack switching, and freely available under BSD like open source license.

Since, events run to completion and Contiki does not allow interrupt handlers to post new events therefore; there is no process synchronization provided in Contiki.

### 4.4 Memory Management

Contiki supports dynamic memory management apart from this it also supports dynamic linking of the programs. In-order to guard against memory fragmentation Contiki uses Managed Memory Allocator [22]. Contiki's managed memory allocator makes sure that memory fragmentation does not occur. The primary responsibility of managed memory allocator is to keep the allocated memory free from fragmentation by compacting the memory when blocks are free. Therefore, a program using the memory allocator module cannot be sure that allocated memory stays in place.

For dynamic memory management Contiki also provide memory block management functions [22]. This library provides simple but powerful memory management functions for blocks of fixed length. A memory block is statically declared using the MEMB() macro. Memory blocks are allocated from the declared memory by the memb_alloc() function, and are deallocated using memb_free() function.

### 4.5 Communication Protocol Support

Contiki supports rich set of communication protocols. In Contiki, we can use both versions of IP i.e., IPv4 and IPv6. Contiki provides the implementation of *u*IP TCP/IP protocol stack which makes it possible to communicate with TCP/IP protocol suite even on small 8 bit micro-controllers. *u*IP does not require its peers to have full size stacks, but it can communicate with peers running a similar lightweight stack.

*u*IP implementation have the minimum set of features needed for a full TCP/IP stack. *u*IP is written in C language, it can only support one network interface, and it supports TCP, UDP, ICMP, and IP protocols.

Since, memory is a scare resource in embedded devises therefore; *u*IP uses memory efficiently by using memory management mechanisms. *u*IP stack does not use explicit dynamic memory allocation. It uses a global buffer to hold the incoming data packets. Whenever, a packet is received Contiki places it in the global buffer and notifies the TCP/IP. If it's a data packet, TCP/IP notifies the appropriate application. Application needs to copy the data in the secondary buffer or it can immediately process the data. Once the application is done with the received data, Contiki overwrites the global buffer with new incoming data. If application delays data processing, then data can be overwritten by new incoming data packets.

Contiki provides implementation of RPL (IPv6 routing protocol for low power and lossy networks) [21] by the name ContikiRPL [20]. ContikiRPL operates on low power wireless links and lossy power line links.

# 5 MANTIS

MANTIS, MultimodAl system for NeTworks of In-situ wireless sensors provides a new multithreaded operating system for wireless sensor networks. MANTIS is a lightweight and energy efficient operating system and it has a footprint of 500 bytes, which includes kernel, scheduler, and network stack. MANTIS Operating System (MOS), key feature is that it is portable across multiple platforms i.e., we can test MOS applications on PDA, and on x86 personal computers afterwards, application can be ported to the sensor node. MOS also supports remote management of sensor nodes through dynamic programming. MOS is written in C and it supports application development in C. Following subsection discusses the design features of MOS in more detail.

## 5.1 Architecture

MOS follows the layered multithreading design as shown in Figure 4. In layered architecture, services provided by an OS are implemented in layers. Each layer acts as an enhanced virtual machine to the layers above. Following are the different services implemented at each layer of MOS.

**Layer 3:** Network Stack, Command Server, and User Level Threads
**Layer 2:** MANTIS system API
**Layer 1:** Kernel/Scheduler, Communication Layer (MAC and PHY), and Device Drivers
**Layer 0:** Hardware

Layering structure imposes a hierarchical structure and fixed layering is not flexible. Crossing a layering boundary has associated overheads. Due to the layered approach an OS gets more reliable, manageable, understandable, and easily modifiable. Since, senor networks OS are not that complex as compared with traditional OS therefore; it's not a bad idea to use layering architecture.

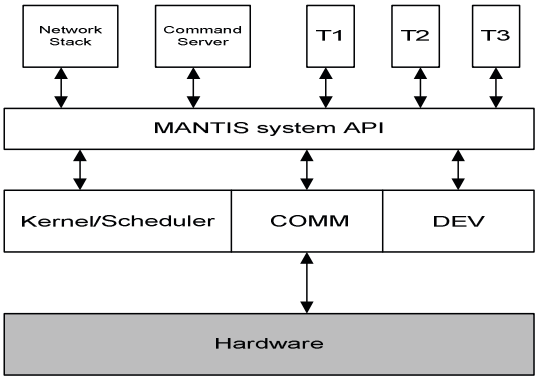


**Fig. 4.** MANTIS OS Architecture. Kernel Scheduler, COMM, DEV, MANTIS System API, Network Stack, and Command Server comprises MOS

MOS supports rich set of Application Programming Interface (API), written in C language. The choice of C language API simplifies cross platform support [4]. The C code developed for MANTIS sensor can be compiled to X86 PCs with little or no modification.

MOS kernel only handles the timer interrupt all other interrupts are directly sent to associated device drivers. When a device driver receives an interrupt, it posts a semaphore in order to activate a waiting thread, and this thread handles the event that has caused the interrupt.

### 5.2 Scheduling

MOS uses preemptive priority based scheduling. MOS uses a UNIX like scheduler with multiple priority classes and it uses round robin within each priority class. The length of time slice is configurable, by default it is set to 10 milliseconds (ms). The scheduler uses a timer interrupt for context switches. Context switches are also triggered by system calls and semaphore operations. Energy efficiency is achieved by the MOS scheduler by switching microcontroller to sleep mode when application threads are idle.

The ready queue of the MOS scheduler comprises of five priorities ranging from high to low: Kernel, Sleep, High, Normal, and Idle. The scheduler schedules the highest priority task in the ready queue. The task either runs to completion or gets preempted if its time slice expires. For time slicing MOs scheduler uses 16 bit timer. When there is no thread in the ready queue, system gets to the sleep mode. If the system is suspended on I/O, then the system enters the moderate idle sleep mode. If the application threads have called sleep system call, then system gets to deep power save sleep mode. A separate queue maintains the ordered list of thread that have called the sleep(), and is ordered by sleep time from low to high. The sleep priority in the ready queue enables newly woken threads to have the highest priority so that they can be serviced first after wake up.

The MOS kernel maintains ready list head and tail pointers for each priority level. There are 5 priority levels and these pointers consume 20 bytes in total. These two pointers helps in fast addition and deletion of threads from a ready queue hence, improved performance in manipulating thread lists. It also uses a current thread pointer of 2 bytes, an interrupt status byte, and one byte of flags. The total static overhead for scheduling is 144 bytes.

MOS scheduler uses round robin scheduling within the each priority class. This means threads of highest priority class can make lower priority class threads to strave. MOS use priority scheduling that may support real time task better than TinyOS scheduler. But it still needs real time schedulers like Rate Monotonic and Earliest Deadline First in-order to accommodate real time tasks.

### 5.3 Threading Model and Synchronization

MOS supports preemptive multitasking. MOS team designed a multithreaded OS because of the facts presented in [23], i.e., "A thread driven system can achieve the high performance of event based systems for concurrency intensive applications, with appropriate modification to the threading package." Memory of the sensor node is a scare resource therefore, MOS maintains two logically distinct sections of RAM: the space for global variables that is allocated at the compile time, and the rest of the RAM is managed as a heap. Whenever a thread is created, stack space is allocated by the kernel out of heap. The stack space is returned to heap once the thread exit. Thread table is the main data structure that is being handled by the MOS kernel. In thread table, there is one entry per thread. MOS statically allocates memory for the thread table therefore, there can be fixed maximum number of threads hence fixed overhead. The maximum number of threads can be adjusted at the compile time. By default it is 12. Thread table entry comprises of 10 bytes and it contains: current stack pointer, stack boundary information (base pointer and size), pointer to thread starting function, thread priority level, and pointer to next thread. Once a thread is suspended its context is saved on the stack. Since, each thread table entry comprises of 10 bytes and by default 12 threads can be created therefore, associated overhead in terms of memory is 120 bytes. By default each thread gets a time slice of 10 ms and context switch happens with the help of timer interrupt. System calls and posting of a semaphore operation can also trigger context switch.

Multithreading support in MOS comes at the cost of context switching and stack memory overhead. In [4], the argument presented in favor of context switching overhead is that it is only a moderate issue in WSNs. It has been observed that each context switch incurs 60 microseconds overhead in comparison to this default time slice is much larger i.e., 10 ms which is less than 1% of the microcontroller cycles. Second cost is of stack memory allocation. The default thread stack in MOS is 128 bytes and MICA2 motes have a 4 KB RAM. Since, MOS kernel occupies 500 bytes therefore considerable space is available to support threading.

MOS avoids race conditions using binary mutex and counting semaphores. Semaphore in MOS is a 5 byte structure and it is declared by an application as needed. Semaphore structure contains a lock or count byte along with head and tail pointers.

### 5.4 Memory Management and Security

MANTIS allows dynamic memory management but it discourages to do so because dynamic memory management incurs lot of overhead. Since, memory is a scarce resource in senor nodes therefore; MANTIS OS discourages the dynamic memory management mechanisms. MANTIS manages different threads memory using the thread table that has already been discussed. MANTIS does not provide any mechanism for memory security.

### 5.5 Communication Protocol Support

MOS implements network stack in two parts. The first part of the network protocol stack is implemented in user space as shown in Figure 4. First part contains the implementation of layer 3, layer 2 and layer 1 protocols. While second part contains the implementation of the MAC and PHY layer operations. The rational behind implementing the layer 3 and above layers functionality in user space is to provide flexibility. If an application wants to use its own data driven routing protocol, then it can implement its routing protocol in the user space and can check its functionality. The downside of the approach is performance i.e., network protocol stack has to use API's provided by MANTIS instead of communication directly with the device driver and hardware. This results in many context switches that involves computational and memory overheads.

The second part of the networking protocol stack is implemented in a COMM layer. COMM layer primarily implements synchronization and MAC layer functionalities. COMM layer provides a unified interface for communication device drivers, for interfaces such as serial, USB, and radio devices. The COMM layer also performs packet buffering functionality. It is possible that packets arrive from the network for a thread that is not currently scheduled. In such scenarios COMM layer will buffer packets. Once the thread gets scheduled COMM layer passes a pointer to the data to the concerned thread.

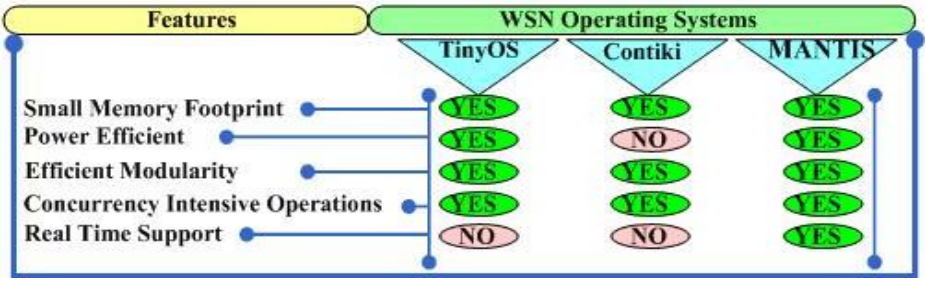


**Fig. 5.** WSN OS Grading

## 6 Future Research Directions

Plenty of research has been done on WSN OS but still it's not an out dated research domain. It's relatively new research domain therefore; there are some issues that need

to be resolved. Following are the some issues that must be taken up for future research.

### 6.1 Support for Real Time Applications

There are many real time application areas for WSN e.g., in industry automation, chemical processes, and multimedia data processing and transmission. Schedulers have been designed to support soft real time operations in some operating systems but the effort is far from complete. In future, we need scheduling algorithms that can accommodate both soft and hard real time requirements of applications.

### 6.2 Secondary Storage Management

With the passage of time new application areas for WSN are emerging and applications are requiring more and more memory. Typical databases application requires a secondary storage with sensor nodes. According to the best of authors knowledge, there exist no work on secondary storage and file management in sensor nodes. Secondary storage management can be an active area of research for WSN OS in future.

### 6.3 Virtual Memory

Since, sensor node has very limited RAM and applications are requiring more and more RAM. Therefore, in future we need to introduce a virtual memory concept in sensor networks OSs. We need to device virtual memory management techniques that are power as well as memory efficient.

### 6.4 Memory Management and Security

Little work has been done on memory management in WSN OS. The primary reason behind this is that, it has been assumed that only single application runs on a WSN OS. In future, we can have sensor nodes that can sense different phenomenon's therefore, it is possible that multiple application runs on sensor node. In such a scenario we need to manage node's memory and we need to protect one process memory from another. Research needs to be done in memory management and security keeping in view the limitations of the sensor nodes.

## 7 Conclusions and Future Work

In this paper, we have investigated the most widely used operating systems for WSNs. This paper helps to understand the characteristics of an OS for WSNs in particular and embedded devices in general. Design strategies for various components of an OS for WSN has been explained, investigated along with their relative pros and cons. Target application areas of different WSN OS has been pointed out. We believe that presented pros and cons of different design strategies presented here will motivate the researcher to design more robust OSs for WSNs. Moreover, this survey will help the application and network designer to select an appropriate OS for their WSN applications.

In future, we plan to investigate other OSs for WSN i.e, SensorOS [6], A Dynamic Operating System for sensor nodes [7], and Nano-RK [9].

## References


1. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D.: Tinyos: An operating system for sensor networks, pp. 115–148 (2005), http://dx.doi.org/10.1007/3-540-27139-2_7
2. Cooprider, N., Archer, W., Eide, E., Gay, D., Regehr, J.: Efficient memory safety for tinyos. In: 5th international conference on Embedded networked sensor systems. SenSys 2007, pp. 205–218. ACM, New York (2007)
3. Klues, K., Liang, C.J.M., Paek, J., Musaloiu, R., Levis, P., Terzis, A., Govindan, R.: TOSThread: Thread-safe and Non-Invasive Preemption in TinyOS. In: 7th ACM conference on Embedded Networked Sensor Systems, pp. 127–140 (2009)
4. Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, H.R.: Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. Mob. Netw. Appl. 10(4), 563–579 (2005)
5. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: 29th Annual IEEE International Conference on Local Computer Networks, pp. 455–462. IEEE Computer Society, Washington (2004)
6. Kuorilehto, M., Alho, T., Hannikainen, M., Hamalainen, T.D.: SensorOS: A New Operating System for Time Critical WSN Applications. In: Vassiliadis, S., Bereković, M., Hämäläinen, T.D. (eds.) SAMOS 2007. LNCS, vol. 4599, pp. 431–442. Springer, Heidelberg (2007)
7. Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A Dynamic Operating System for Sensor Nodes. In: 3rd International Conference on Mobile systems, applications and services, pp. 163–176 (June 2005)
8. Kim, H., Cha, H.: Multithreading Optimization Techniques for Sensor Network Operating Systems. In: 4th European conference on Wireless Sensor Networks, pp. 293–308 (January 2007)
9. Eswaran, A., Rowe, A., Rajkumar, R.: Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks. In: 26th IEEE International Real Time Systems Symposium, pp. 256–265 (December 2005)
10. Reddy, V., Kumar, P., Janakiram, D., Kumar, G.A.: Operating Systems for Wireless Sensor Networks: A Survey. International Journal of Sensor Networks 5(4), 236–255 (2009)
11. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless Sensor Networks: a survey. Computer Networks 38(4), 393–422 (2002)
12. Romer, K., Mattern, F.: The design space of wireless sensor networks. IEEE Wireless Communication 11(6), 54–61 (2004)
13. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.S.J.: System architecture directions for networked sensors. In: Architectural Support for Programming Languages and Operating Systems, pp. 93–104 (2000)
14. TinyOS Network Working Group, Web page, http://docs.tinyos.net/index.php/TinyOS_Tutorials#Network_Protocols
15. Network Protocols- TinyOS documentation Wiki. Web page, http://docs.tinyos.net/index.php/Network_Protocols

16. Lin, K., Levis, P.: Data Discovery and Dissemination with DIP. In: 7th International Conference on Information Processing in Sensor Networks, pp. 433–444 (2008)
17. Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nes C language: A holistic approach to networked embedded systems. In: SIGLAN 2003 (2003)
18. Dwivedi, A.K., Tiwari, M.K., Vyas, O.P.: Operating Systems for Tiny Networked Sensors: A Survey. International Journal of Recent Trends in Engineering 1(2) (May 2009)
19. Protothreads- Lightweight, Stackless Threads in C, http://www.sics.se/~adam/pt/
20. Tsiftes, N., Eriksson, J., Dunkels, A.: Low-Power Wireless Ipv6 Routing with ContikiRPL. In: ACM/IEEE IPSN (2010)
21. Winter, T., Thubert, P.: RPL: Ipv6 Routing Protocol for Low Power and Lossy Networks, July 28 (2010) draft-ietf-roll-rpl-11
22. Contiki Documentation, http://www.sics.se/~adam/contiki/docs/
23. Von Behren, R., Condit, J., Brewer, E.: Why Events are a Bad Idea (for High Concurrency Servers). In: 9th Workshop on Hot Topic in Operating Systems, HOTOS IX (2003)