The Use of POSIX in Real-time Systems, Assessing its Effectiveness and Performance

Kevin M. Obenland
The MITRE Corporation, 1820 Dolley Madison Blvd. McLean, VA 22102
obenland@mitre.org

Abstract

The POSIX standard promotes portability of applications across different operating system platforms. This is especially important for applications designed for longevity, where the hardware and software infrastructure may change during the application's life cycle. However in real-time systems, where predictability and low overhead are important, portability is often sacrificed. In this paper we will discuss the use of POSIX® in real-time systems, including the POSIX real-time and thread extensions. We will first discuss what POSIX covers, and the differences that still exist between operating system implementations. We will then look at the performance of various POSIX mechanisms, using as case studies a general purpose OS (Solaris™ 8) and a real-time operating system (LynxOS®).

1 Introduction

In the design of today's computing systems it is becoming increasingly important to design software with an open system architecture utilizing industry adopted standards. The need to develop open systems is driven by three major factors. *First*, gone are the days where a single developer can implement the entire system from scratch. Software development programs are continuously growing in scale, requiring teams of increasing size. *Secondly*, software does not operate in isolation; it must co-exist with the vast amount of commercially available software. *Lastly*, the lifecycle of a software application is typically long requiring numerous modifications and updates as new features are added.

An open software architecture addresses the challenges of today's software development process by defining standard software interfaces, which promotes interoperability and portability. Openly published standard interfaces also reduce the cost of adding functionality in the future.

Standards are pervasive in today's computer systems. New standards are constantly being defined to address the ever-changing state of software technology. A standard will not be effective if it is not used, or if it is gone tomorrow. To be effective it is important for a standard to be based on well-established technology and accepted by a wide portion of the industry.

The original Portable Operating System Interface for Computing Environments (POSIX®) standard was first published in 1990 [1]. POSIX is based on UNIX, a well-established technology dating back to the early 1970s. POSIX defines a standard way for an application to interface to the operating system. The original POSIX standard defines interfaces to core functions such as: file operations, process management, signals, and devices. Subsequent releases of POSIX have also been defined to cover real-time extensions and multi-threading [1].

In a perfect world, because of the advantages cited above, one would always choose a standard. However, in the real world, there are a number of questions that must be asked before deciding to use a standard. These include:

- Does the standard provide the **functionality** needed by my application?
- Is the **performance** of the standard, or implementation of the standard, suitable for my application?
- Do commercially **available** implementations of the standard exist?

In this paper we assess the usefulness of POSIX in real-time time systems by looking at these three factors: (functionality, performance, and availability.) Because real-time systems typically have stringent performance constraints emphasis is placed on the performance of POSIX implementations.

This paper is organized into six sections. The following section reviews the features important in a POSIX operating system. Section 3 discusses implementation details of two POSIX operating systems:

Solaris and LynxOS. Section 4 introduces a set of benchmarks used to measure the performance of real-time operating systems, and in Section 5 we use these benchmarks to measure the real-time performance of LynxOS and Solaris. Finally Section 6 presents the conclusions of this paper.

2 POSIX Real-time Operating Systems

The POSIX family of standards includes over 30 individual standards, ranging from specifications for basic operating system services to specifications for testing the conformance of an operating system to the standard [2]. This paper focuses on those standards important in the development of real-time embedded systems. In this section we discuss real-time systems as well as giving a brief review of the relevant POSIX standards.

2.1 Real-time Systems

A real-time system is one where the timeliness of the result of a calculation is important [3][4]. Examples include military weapons systems, factory control systems, and Internet video and audio streaming. Real-time systems are typically categorized into two classes: hard and soft. In a *hard real-time system* the time deadlines must be met or the result of a calculation is invalid. For example in a missile tracking system, if the missile is delayed it may miss its intended target. The timing constraints in a *soft real-time system* are not as stringent. There is still some utility to the result of a calculation if it does not meet its timing deadline. Internet audio/video streaming is an example of a soft real-time system. If a packet of data is late or lost the quality of the audio/video is degraded, but the stream may still be audible.

To guarantee that the timing requirements of a real-time system are met the behavior, and timing, of the underlying computing system must be predictable [5]. The time required by all operations must be bounded for the timing of the system to be called *predictable*. This implies that the worst case timing of all operations is known. Typically though a system is called predictable only if its worst case timing is very close to its average case timing.

Table 1: POSIX Standards

Standard	Name	Description
1003.1a	OS Definition	Basic OS interfaces; includes support for: (single process, multi process, job control, signals, user groups, file system, file attributes, file device management, file locking, device I/O, device specific, system database, pipes, FIFO, and C language
1003.1b	Real-time Extensions	Functions needed for real-time systems; includes support for: real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, file sync, mapped files, memory locking, memory protection, message passing, semaphores, and shared memory
1003.1c	Threads	Functions to support multiple threads within a process; includes support for: thread control, thread attributes, priority scheduling, mutexes, mutex priority inheritance, mutex priority ceiling, and condition variables
1003.1d	Additional Real-time Extensions	Additional interfaces; includes support for: new process create semantics (spawn), sporadic server scheduling, execution time monitoring of processes and threads, I/O advisory information, timeouts on blocking functions, device control, and interrupt control.
1003.1j	Advanced Real-time Extensions	More real-time functions including support for: typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues
1003.21	Distributed Real-time	Functions to support real-time distributed communication; includes support for: buffer management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols
1003.1h	High Availability	Services for Reliable, Available, and Serviceable Systems (SRASS); includes support for: logging, core dump control, shutdown/reboot, and reconfiguration

2.2 POSIX Real-time Related Standards

Of the more than 30 POSIX standards the seven standards listed in Table 1 are especially relevant to the development of real-time and embedded systems. With the first three standards (1003.1a,1b,1c) being the most widely supported. POSIX 1003.1a defines the interface to basic operating system functions, and was the first to be adopted in 1990 [1][6]. Real-time extensions are defined in the standards 1003.1b, 1003.1d, 1003.1j and 1003.21 [7][8][9][10]. However, the original real-time extensions, defined by 1003.1b, are the only standard commonly implemented. Support for multiple threads in a process is provided in a separate standard, POSIX 1003.1c. POSIX also includes support for high availability in the 1003.1h standard [11].

Commercial support for POSIX varies widely. Because POSIX 1003.1a is based on UNIX, any UNIX based operating system will naturally be very close to the standard. To be POSIX *conformant* to the standard, the operating system, and hardware platform, has to be certified using a suite of tests [12]. Currently test suites exist only for POSIX 1003.1a. Because POSIX is structured as a set of optional features, operating system vendors can choose to implement portions of POSIX and still be *compliant* to POSIX. Compliance only requires the vendor to state which features of POSIX are and are not implemented. This is a source of confusion because, for marketing reasons, almost all vendors report that they are POSIX compliant.

2.2.1 POSIX profiles

Embedded systems typically have space and resource limitations, and an operating system that includes all the features of POSIX may not be appropriate. The POSIX 1003.13 profile standard was defined to address these types of systems [13]. POSIX 1003.13 does not contain any additional features; instead it groups the functions from existing POSIX standards into units of functionality. The profiles are based on whether or not an operating system supports more than one process and a file system. The four current profiles are summarized in Table 2.

Table 2: POSIX 1003.13 Profiles

Profile	Number of Processes	Threads	File System
54	Multiple	Yes	Yes
53	Multiple	Yes	No
52	Single	Yes	Yes
51	Single	Yes	No

2.2.2 POSIX real-time extensions

POSIX 1003.1b, as well as 1003.1d and 1003.1j define extensions useful for development of real-time systems. Functions defined in the original real-time extension standard 1003.1b are supported across a wider number of operating systems than the other two specifications. For this reason this paper focuses on POSIX 1003.1b. The following features constitute the bulk of the features defined in POSIX 1003.1b:

- **Timers**: Periodic timers, delivery is accomplished using POSIX signals
- **Priority scheduling**: Fixed priority preemptive scheduling with a minimum of 32 priority levels
- **Real-time signals**: Additional signals with multiple levels of priority
- **Semaphores**: Named and memory counting semaphores
- **Memory queues**: Message passing using named queues
- **Shared memory**: Named memory regions shared between multiple processes
- **Memory locking**: Functions to prevent virtual memory swapping of physical memory pages

Figure 1 shows C code for creating and using a POSIX timer. Creating a timer consists of two steps: specifying a signal to be used to deliver the timer, and creating/setting the timer itself. In this example we use the highest priority real-time signal (SIGRTMIN) to asynchronously call the timer handler routine. Two values must be specified for the timer: the initial expiration time (*it_value*) and the frequency (*tv_sec*). The structure (*itimerspec*) allows nanosecond time specification; however, actual resolution is dependent on the system. The POSIX call *clock_getres*() can be used to determine the actual resolution, typically 10 or 1 ms.

```
#include <signal.h>
#include <time.h>

void timer_create(int num_secs, int num_nsecs)
{
    struct sigaction sa;
    struct sigevent sig_spec;
    sigset_t allsigs;
    struct itimerspec tmr_setting;
    timer_t timer_h;

    /* setup signal to respond to timer */
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = timer_intr;

    if ( sigaction(SIGRTMIN, &sa, NULL) < 0 )
        perror("sigaction");

    sig_spec.sigev_notify = SIGEV_SIGNAL;
    sig_spec.sigev_signo = SIGRTMIN;

    /* create timer, which uses the REALTIME clock */
    if (timer_create(CLOCK_REALTIME,&sig_spec,&timer_h) < 0 )
        perror("timer create");

    /* set the initial expiration and frequency of timer */
    tmr_setting.it_value.tv_sec = 1;
    tmr_setting.it_value.tv_nsec = 0;
    tmr_setting.it_interval.tv_sec = num_secs;
    tmr_setting.it_interval.tv_sec = num_nsecs;
    if ( timer_settime(timer_h,0,&tmr_setting,NULL) < 0 ) {
        perror("settimer");
    }
    /* wait for signals */
    sigemptyset(&allsigs);
    while ( 1 ) {
        sigsuspend(&allsigs);
    }
}
/* routine that is called when timer expires */
void timer_intr(int sig, siginfo_t *extra, void *cruft)
{
    /* perform periodic processing and then exit */
}
```

Figure 1: Creating and using a POSIX timer

POSIX 1003.1b provides support for fixed priority preemptive scheduling. To be compliant with POSIX an operating system must contain at least 32 priorities. POSIX defines three scheduling policies to handle processes running at the same priority. For SCHED_FIFO processes are scheduled first in first out, and run until completion. For SCHED_RR the scheduler uses a time quanta to schedule processes in a round robin fashion. The SCHED_OTHER policy is also included to handle an implementation-defined scheduling policy. Because SCHED_OTHER is implementation dependent, it is not portable across different platforms, and its use should be limited.

POSIX uses named objects for several different mechanisms including: semaphores, shared memory, and message queues. These names are analogous, but independent, to names in the file system. For semaphores one process creates the semaphore, and other processes can attach to the semaphore using its name. Both processes can perform signal (*sem_post*) or wait (*sem_wait*) operations.

2.2.3 POSIX threads

In POSIX, threads are implemented in an independent specification, which means that their specification is independent of the other real-time features [1][14]. Because of this there are a number of features from the real-time specification that are carried over to the thread specification. For example priority scheduling is done on a per-thread basis, but is handled in a similar manner as scheduling in POSIX 1003.1b. A thread's priority and scheduling policy is typically specified when it is created.

The POSIX thread specification defines functionality and/or makes modifications to POSIX in the following areas:

- **Thread control**: Creation, deletion and management of individual threads
- **Priority scheduling**: POSIX real-time scheduling extended to include scheduling on a per thread basis; the scheduling scope is either done globally across all threads in all processes, or performed locally within each process
- **Mutexes**: Used to guard critical sections of code; mutexes also include support for priority inheritance and priority ceiling protocols to help prevent priority inversions
- **Condition variables**: Used in conjunction with mutexes, condition variables can be used to create a monitor synchronization structure
- **Signals**: Ability to deliver signals to individual threads

2.3 POSIX coverage in operating system implementations

Table 3 shows the level of compliance to POSIX 1003.1(a/b/c) for several commercially available operating systems. General-purpose operating systems like Solaris and IRIX are the most compliant. LynxOS is conformant to POSIX 1003.1a and with the 3.1 release will be very close to full compliance to the three standards. VxWorks® only supports a subset of the POSIX standards because VxWorks is based on a single process model. VxWorks uses its own threading library, but a POSIX thread implementation is provided through a third party vendor. Linux® provides good support for the base POSIX APIs and threads, but is missing real-time features such as timers and message queues.

Table 3: POSIX in commercial operating systems

OS	POSIX 1003.1a (Base POSIX)	POSIX 1003.1b (Real-time extensions)	POSIX 1003.1c (threads)
Solaris	Full support	Full support	Full support
LynxOS	Conformant	Full support	3.0.1 based on draft and missing thread attributes; 3.1 based on final standard
VxWorks	Partial support; support for functions that do not require a process model	Partial support; support for functions that do not require a process model	Support through a third party product
IRIX	Conformant	Full support	Full support
Linux	Full support	Partial support; no support for timers or message queues	Full support

3 Operating System Design

The design of an operating system can have a significant impact on its ability to be used in a real-time system. This includes the internal design of the operating system as well as the features it provides to the application programmer. This section focuses on the design of two operating systems (Solaris, and LynxOS), and their suitability for use in a real-time system.

3.1 Desired features of a real-time operating system

Real-time systems are typically implemented with multiple asynchronous threads of execution. This is dictated by the need to react to external events, and control asynchronous devices. Because of this characteristic a real-time operating system (RTOS) must support multithreading. Also because the criticality and rates of events are different, the RTOS must support a notion of priority so that a time critical task is not delayed because of a non-critical task. Furthermore tasks need to communicate, therefore the OS must provide synchronization and communication facilities.

A real-time OS also needs to support timing features like high-resolution timers and clocks. Timers are used to support periodic processing and to detect system timeout errors. Clocks are needed to keep track of time. Typical real-time applications may need to be aware of time at a granularity of micro or milliseconds.

With respect to performance the operating system must be predictable and add minimal overhead. As discussed in Section 2.1, a real-time system must be predictable or deterministic. This implies that the time required by all operations, including operating system functions, must be deterministic as well. To be deterministic an operating system must be preemptable. Meaning that if the OS is processing a request on behalf of a low priority task, it must be able to stop what it is doing and turn its attention to a higher priority task. This prevents a situation where a high priority task is forever delayed by the operating system.

3.2 Solaris

Solaris is a general-purpose UNIX operating system available from Sun Microsystems™ developed to run on SPARC™ and Pentium™ class CPUs. Solaris has many of the features required for a real-time system [15]. These features are detailed below:

- A multithreaded preemptable kernel
- Global priority model: Threads are mapped to lightweight processes, which are allocated to priority classes and then scheduled globally. See Section 3.2.1 for a discussion of Solaris priority classes.
- Configurable clock tick: The frequency of the clock tick can be changed, thereby increasing or decreasing the frequency that the scheduler runs.
- High resolution POSIX timers: Solaris defines an additional POSIX timer (CLOCK_HIGHRES) that, based on the capability of the hardware, can provide timers with nanosecond and µs resolution.
- Priority I/O streams
- Additional support for POSIX real-time APIs: Solaris 8 now supports all of POSIX 1003.1b.
- Symmetrical multiprocessing support: Solaris supports multiprocessing that is transparent to the user. This also allows processors to be reserved for real-time processing, increasing the determinism.

3.2.1 Solaris thread implementation

Solaris implements both user-level and kernel-level threads. User-level threads are implemented as a library at the user application level [16]; whereas kernel-level threads are the unit of execution seen by the kernel. Solaris uses the Lightweight Processes (LWP) mechanism to run kernel-level threads on processors. The mapping of user-level threads to LWPs can be done in a number of different ways. If multiple user-level threads are mapped to a single kernel-level thread then at most one thread can be active at a time. To take advantage of multiple processors user-level threads can be mapped one-to-one to LWPs.

Figure 2 illustrates how Solaris processor sets and processor binding can be used to dedicate processors for real-time tasks [15][17]. The *psrset* command is first used to create a pool of one or more processors. Note that all but one processor is eligible for inclusion in the processor set; one processor is needed to process lightweight processes outside the set. The *psradm* command can then be used to disable unbound interrupts on the processors in the processor set. The *psrset* command is then used to run real-time processes on the processors in the bound processor set. All other non-real-time processes, and interrupts, run on processors outside the real-time processor set. As shown in Section 5, this mechanism has a dramatic effect on the timeliness of real-time processing.

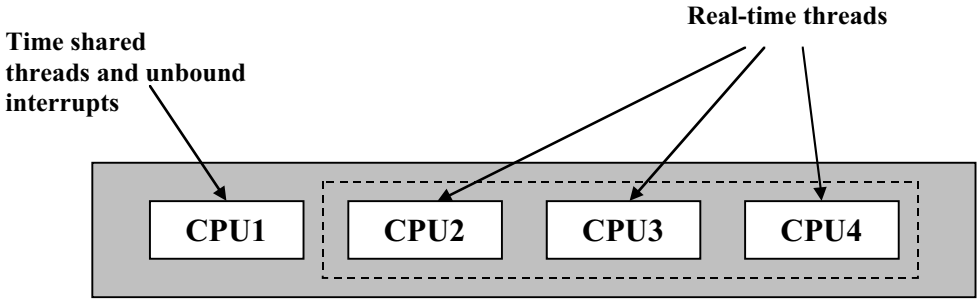

Figure 2: Solaris Processor Binding and Control

3.2.2 The Solaris scheduler

To support different types of scheduling policies, Solaris runs each lightweight process in one of four priority classes. These classes are shown in Table 4 [15]. Interrupt service routines are not part of the scheduling process, but they are included in Table 4 because they run at a higher priority than all tasks, and thus can interfere with normal LWP processing. Application LWPs run in one of three classes: Real-time, System, or Timesharing. Interrupt threads are reserved for interrupt processing not done in the interrupt service routine.

Scheduling consists of two processes: deciding which LWP to run, and performing tick processing [18]. When the scheduler is invoked it dispatches the LWP with the highest global priority. If there are multiple CPUs in the machine, the scheduler can dispatch multiple LWPs. The second aspect of scheduling is tick processing; the processing that takes place at every clock tick. The scheduler will scan all the active LWPs and update their state. For timesharing threads the scheduler may increase the priority of a LWP if it determines that thread is not receiving a fair share of the CPU. Solaris may also promote a LWP to the system class if the LWP is holding a system resource. Because real-time threads run with a fixed priority scheduling policy, very little tick processing is done for them.

Table 4: Solaris Priority Classes

Class	Priority Range	Description
ISRs	N/A	Asynchronous interrupt service routines; not scheduled
Interrupt threads	160 - 169	Interrupt processing not done in the ISR; scheduled based on priority of ISR
Real-time	100 -159	Time critical tasks; fixed priority preemptive scheduling
System/Kernel	60 - 99	System level functions
Timesharing/Interactive	0 - 59	General purpose applications; OS may dynamically adjust priorities to achieve fairness

3.3 Lynx OS

LynxOS is a UNIX style operating system developed for real-time embedded systems. The Lynx kernel is preemptable, reentrant, and can be scaled down to a footprint as low as 97 Kilobytes [19].

3.3.1 Lynx Scheduling

LynxOS 3.0.1 supports a single scheduling policy, fixed priority preemptive with 256 priority levels. The clock tick frequency is fixed at 100 Hertz, which limits the resolution of timers to 10 milliseconds. The scheduler is also invoked in response to asynchronous events and change in the system state.

3.3.2 Lynx priority tracking

LynxOS uses a mechanism called priority tracking to handle interrupt processing not done in the interrupt service routine [20]. This is in contrast to the interrupt thread class used by Solaris. The problem with using an interrupt thread class is that interrupt processing on behalf of low priority tasks will run at higher priority than application processing of a high priority task. This creates a priority inversion. The way LynxOS solves this problem is to tie the priority of the interrupt processing to the priority of the application thread. The 256 task priorities are subdivided into 512 priorities and application threads use the 256 even

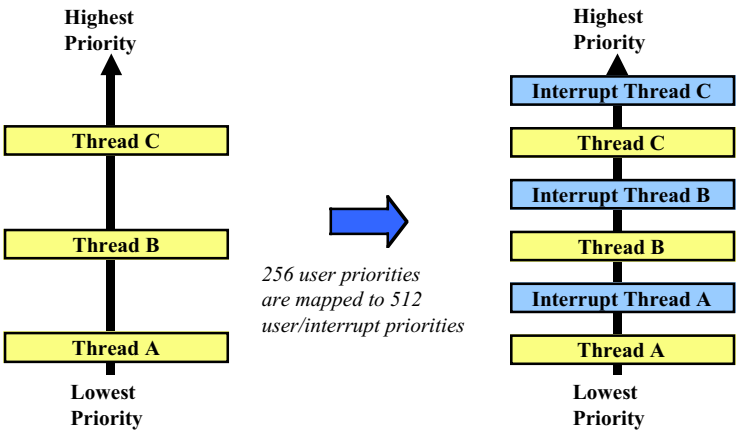

Figure 3: Lynx priority tracking

priorities and interrupt threads use the 256 odd priorities. This idea is illustrated in Figure 3, where interrupt threads run a half step above their corresponding application thread.

Interrupt threads are written as part of the device driver for a particular device, and therefore are not associated with a particular application thread. Because of this LynxOS provides a mechanism by which the device driver can determine the priority of the thread that it is currently running on behalf of. Using this feature, the interrupt thread can adjust its priority to the appropriate level. If in the future a different application thread needs the same device, the interrupt thread is notified and can change its priority.

4 Testing the Real-time Performance of Operating Systems

The benchmarks used in this study are divided into two categories: those that measure the determinism of the OS and those that measure the latency of particular important operations. These benchmarks are motivated by the real-time performance requirements discussed in Section 3.1. The benchmarks test core operating system capabilities and are independent of any actual application. Also because we are interested in determining the best possible real-time performance, all real-time threads are run at the maximum possible real-time priority, and the virtual memory used by the benchmarks is locked into physical memory. Table 5 summarizes the six benchmarks used in this study.

Table 5: Real-time benchmarks

Benchmark	Description	Aspect tested	Parameters
Timer Jitter	Create a periodic thread and measure the deviation between desired and actual expiration	Measures the response time of the operating system	Timer period: (1,10,100 ms)
Response	Execute a fixed processing load and measure its execution time over a number of runs	Determine if a thread can respond in a deterministic fashion	Type of processing: (add,copy,whetstone)
Bintime	Call a time of day clock and measure interval between calls	Measures the maximum kernel blocking time	None
Sync	Measure the latency of thread to thread or process to process synchronization	Measures the context switching time between threads and processes	Type of semaphore: (POSIX named/unnamed semaphore, pthread mutex, lynx semaphore); process to process or thread to thread
Message passing	Measure the latency of sending data form thread to thread or from process to process	Measures the possible throughput of data between processes and threads	Data buffer size; process to process or thread to thread
RT Signals	Measure the latency of real-time signals between two processes	Measures the latency of POSIX real-time signals	None

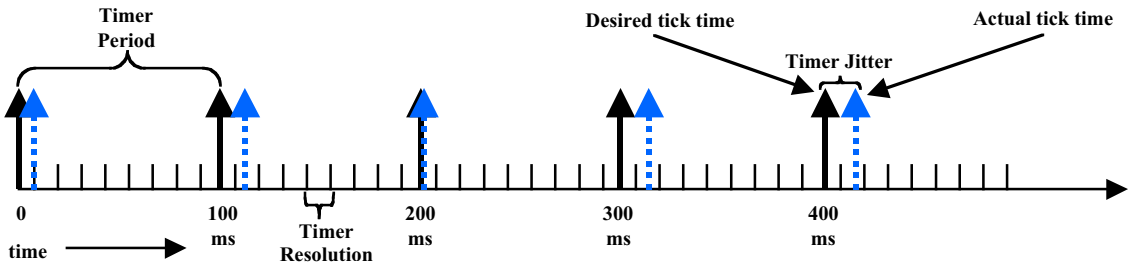

Figure 4: Timer Jitter benchmark

4.1 Deterministic benchmarks

The first three benchmarks shown in Table 5, (Timer Jitter, Response, and Bintime) are designed to measure the determinism of an operating system [21]. Because determinism implies that the time it takes to perform an operation is known under all circumstances, we typically report the worst case time for these benchmarks.

The structure of the *Timer Jitter* test is shown in Figure 3 below. The test creates a timer, sets it to expire at a given period, and then when it expires determines the actual expiration time. The jitter is then defined as the deviation between the actual and desired expiration times. Most current CPUs include a stamp counter that is updated on every CPU cycle. The POSIX *clock_gettime* function in most operating systems uses this stamp counter, giving a high precision time of day clock.

The second deterministic benchmark (Response) measures the actual execution time of a 10 millisecond fixed block of processing. The actual execution time over a number of separate runs is calculated to determine whether or not application response time is deterministic. The fixed processing is generated with a loop consisting of one of three different types of operations: additions (**add**), memory copies (**copy**), or the synthetic Whetstone benchmark (**whet**) [22].

The last deterministic benchmark (Bintime) determines the maximum kernel blocking time [23]. The benchmark uses a high priority real-time thread to repeatedly call a time of day clock and calculate the time required by each call. The time required by each call consists of the time to perform the system call and any time spent blocked in the kernel. Since the time to perform the system call should be constant, the deviation between the maximum time reported by the benchmark and the average time gives a good indication of the maximum time spent blocked in the kernel.

4.2 Latency benchmarks

The final three benchmarks test the synchronization, message passing, and RT signaling capabilities of an operating system. For a real-time system it is important to minimize synchronization and communication latency. Therefore it is important that the average latency of operations is small to minimize the total overhead. Bounding the maximum latency is important as well to achieve determinism.

Three different synchronization tests are shown in Figure 5. In the first test a single thread signals (**S**) and then waits (**W**) on a semaphore. This test measures the latency of semaphore system calls. The second test uses semaphores to signal between two threads. The threads are either in a single, or two different processes. Measurements from the first two tests can be used to determine the context switching time by subtracting off the system call overhead, obtained in test one, from half of the roundtrip signaling time, obtained in test two.

The last test assesses an operating systems ability to deal with priority inversion. The test sets up a classic priority inversion using semaphores. Note for clarity the semaphores are not shown in the picture. The priority inversion occurs when a low priority task acquires (**A**) a resource needed later by a high priority task. The high priority task blocks waiting on the resource and is delayed indefinitely because a independent medium priority task is monopolizing the CPU. This is a priority inversion because now the medium priority task is favored over the high priority task. One typical way of solving this problem is to allow the low priority task to inherit the priority of the high priority task so that it can run and release the resource **R.** In the test a fixed-duration processing loop is used for the medium priority task. If a priority inversion occurs then the time between when the low priority task acquires the resource and when the high priority task receives it will be at least the time in this fixed-duration of processing. If the OS synchronization mechanism prevents a priority inversion then this time will be negligible.

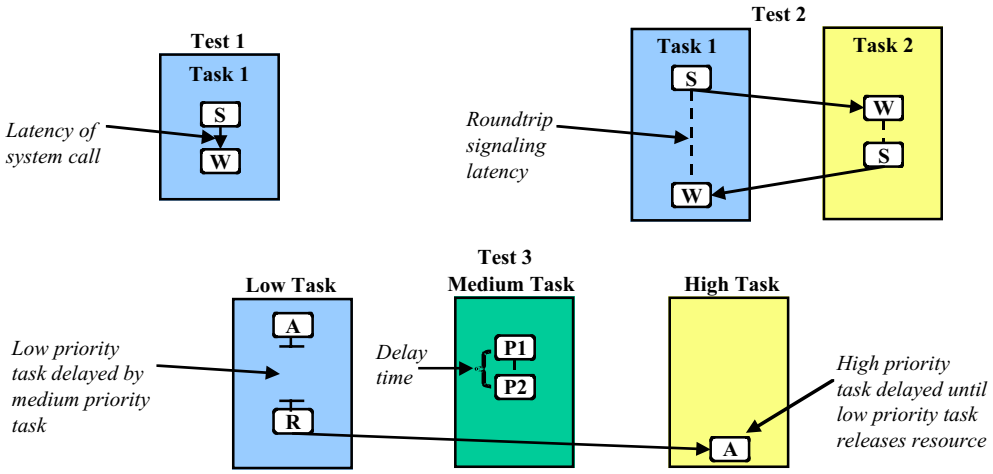

Figure 5: Synchronization tests

The message passing benchmark uses POSIX message queues to measure the latency and throughput of data transfers between two threads in the same process or in different processes. The last benchmark measures the latency of POSIX real-time signals.

5 Benchmark Results

The benchmarks defined in the previous section were run on two different operating systems: LynxOS and Solaris 8. The details of the two systems are shown in Table 6. Note that the CPU, amongst other hardware characteristics, differs between the two platforms. Because our benchmarks were written to test the determinism of the operating systems, and we observe the worst case time, this difference has little impact on the results. However, the speed difference should be considered when comparing the results of average timings.

Table 6 identifies three different Solaris configurations. These different configurations allow us to investigate the impact of using multiple CPUs. The first configuration uses the two processor Ultra 60 as is. For the second configuration one of the CPUs is disabled. In the last configuration one of the CPUs is reserved and the real-time benchmarks are run on it, as described in Section 3.2.1. Also for this configuration the reserved processor is sheltered from all unbound interrupts.

Table 6: Experimental platforms

Platform	Hardware	CPU (Speed)	Operating System	CPU Configuration
Lynx	Dell	Pentium 2 (266 MHz)	Lynx OS 3.0.1	1 CPU
Solaris (2 proc)	Sun Ultra 60	SPARC (360 MHz)	Solaris 8	2 CPUs
Solaris (1 proc)	Sun Ultra 60	SPARC (360 MHz)	Solaris 8	1 CPU
Solaris (1 rt)	Sun Ultra 60	SPARC (360 MHz)	Solaris 8	2 CPUs, 1 CPU reserved to run RT benchmarks

5.1 Non real-time external load

The benchmarks were run stand-alone, i.e. without any other user processes running, and in combination with a non-real-time load. Typically a real-time system will run a mixture of applications, some with real-time requirements and some without. A graphical user interface is an example of a non-real-time application. Table 7 shows the types of processing used to generate the non-real-time load. The load contains CPU intensive applications as well as applications that use interrupting I/O devices such as the file and network subsystems.

Table 7: Non real-time (Heavy) load

Name	Description	Load Degree
CPU	Processing load generated with the Whetstone synthetic benchmark	10 ms every 100 ms
Disk	File write operations	10 ms every 100 ms
Interrupt	External serial interrupt	1000 interrupts/sec
Network	TCP/IP socket transfers	4000 packets/sec
System call	Sequence of utility system calls	10 ms every 100 ms
Memory	Dynamic memory allocation	10 ms every 100 ms
File search	Search files in a directory and all sub-directories	Continuous

5.2 Timer Jitter

Figure 6 shows the results of the timer jitter tests run on all four platforms. Without a load, shown in Figure 6(a), all platforms have acceptable jitter under 200 µsec. Solaris (1 rt) configuration has the least amount of jitter. The jitter for the Lynx configuration is also quite low. Under a heavy load, shown in Figure 6(b), the jitter for the Solaris configurations that do not reserve a real-processor is out of bounds. The worst case jitter, for these configurations, is as great as ten seconds.

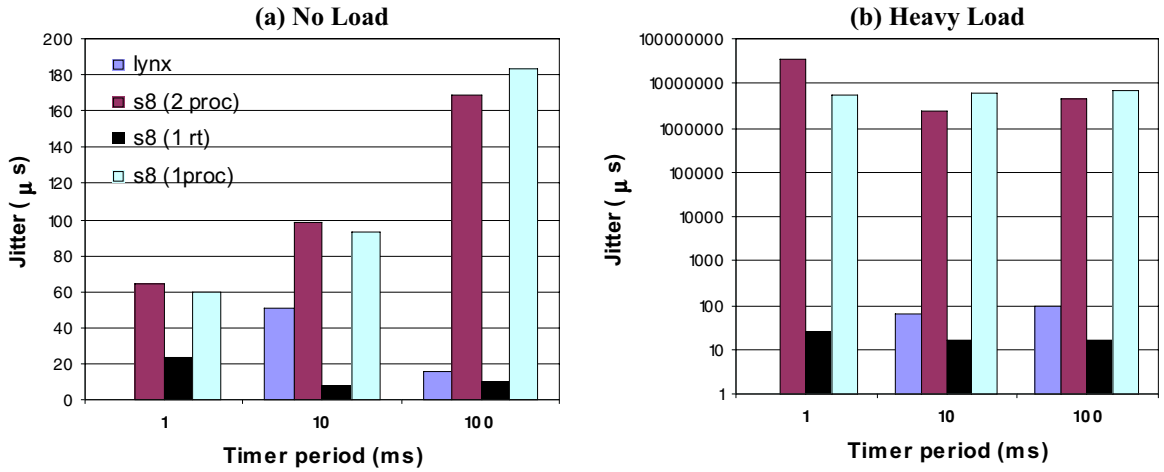

Figure 6: Timer jitter results

5.3 Application Response

Table 7 shows the worst case response results for all configurations. Without a load all configurations have a response result very close to the calibrated value of 10 milliseconds. With a load only the Lynx and Solaris (1 rt) configuration come close to the 10 millisecond value. The worst case results for the standard Solaris platform (Solaris 2 proc) is three orders of magnitude worse than the calibrated value.

Table 8: Worst case response results (in milliseconds)

	add		copy		Whet	
Configuration	**No Load**	**Heavy Load**	**No Load**	**Heavy Load**	**No Load**	**Heavy Load**
Lynx	9.9	9.9	10.0	10.1	10.1	10.2
Solaris (2 procs)	10.1	11236.5	10.7	12061.7	10.6	12162.8
Solaris (1 proc)	10.2	7310.7	10.2	4599.3	10.7	6328.2
Solaris (1 rt)	10.0	10.0	10.0	10.0	10.5	10.5

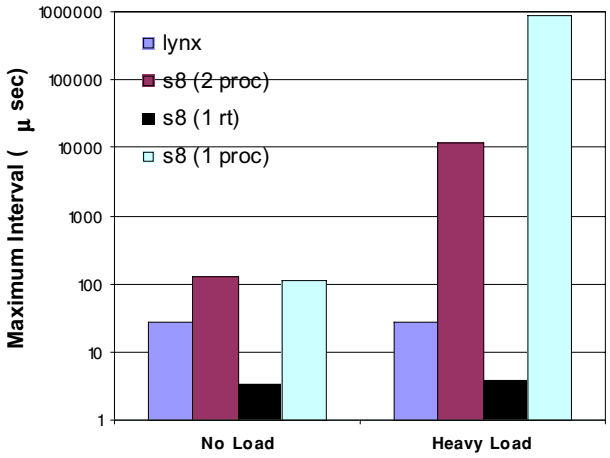

Figure 7: Bintime results

5.4 Bintime

Figure 7 shows the results for the deterministic Bintime benchmark for all configurations. Without a load the kernel imposes very little delay. For the Solaris (1 rt) configuration the delay is below 10 µsec, and for all other configurations the delay is at or less than 100 µsecs. Under a heavy load, the Solaris configurations without a reserved real-time processor again are very non-deterministic. The maximum delay for the single CPU Solaris configuration is close to 1 second.

5.5 Synchronization

In this section we present the results of the synchronization tests described in Section 4.2.

5.5.1 Test 1 (Signaling within a thread)

Figure 8 shows the results of the simple synchronization test for the Lynx and Solaris (1 rt) configurations. Four different types of synchronization mechanisms were tested for Lynx, and three for Solaris. As Figure 8(a) shows, the worst case latency for the Solaris platform is much better than the latency for Lynx platform. Also for both platforms the additional of a load has little affect on the worst case timings.

Figure 8(b) shows the average latencies for the same synchronization mechanisms. For Lynx the lynx semaphores exhibit the highest latency, most likely due to the fact that priority inheritance is implemented for this semaphore. For Solaris the latency of the POSIX named semaphore is much higher than the latency of the other mechanisms. An explanation for this is that the semaphore name is kept in the file system.

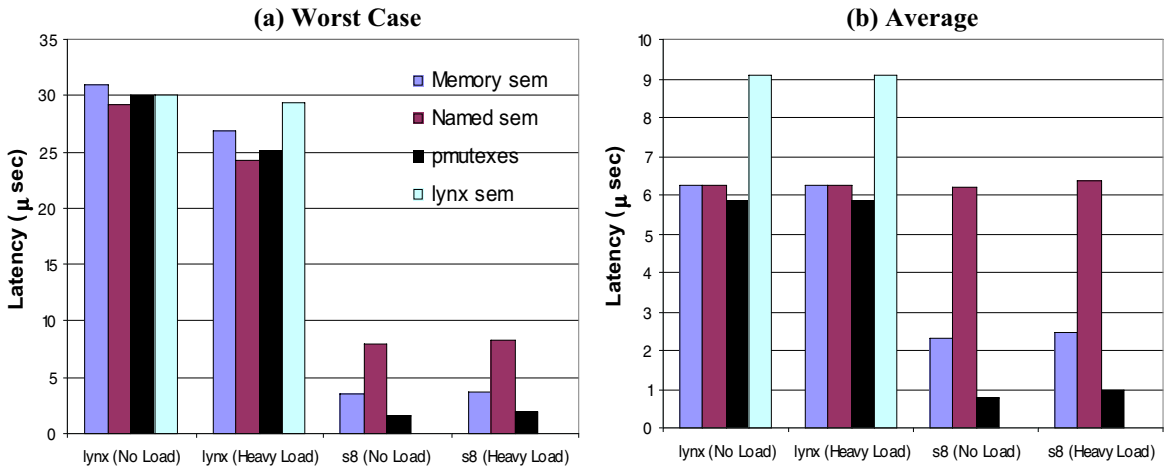

Figure 8: Sync Test 1: Lynx and Solaris (1 rt)

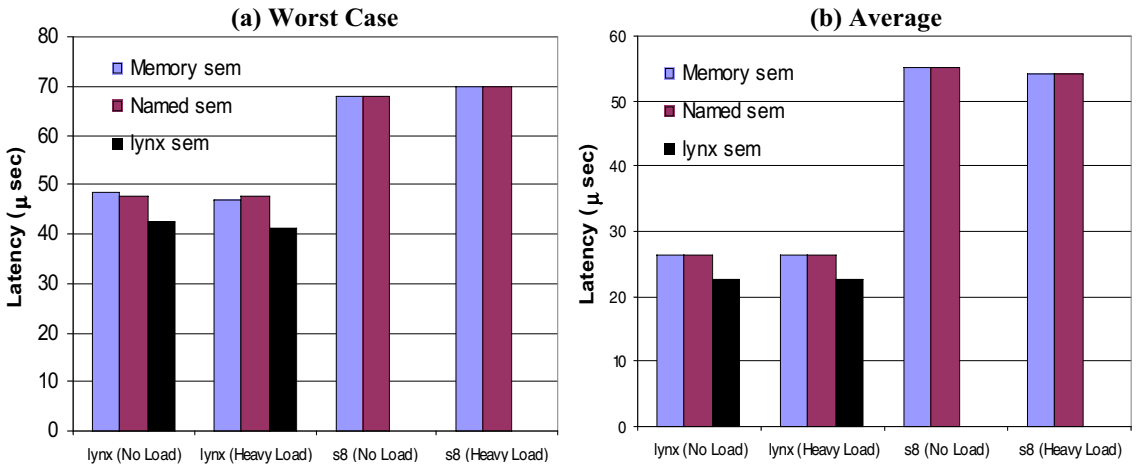

Figure 9: Sync Test 2: Lynx and Solaris (1 rt)

5.5.2 Test 2 (Inter-thread signaling)

Figure 9 shows the results of the inter-thread signaling test for the Lynx and the Solaris (1 rt) configurations. In all cases the average and worst case round-trip time is better for Lynx than Solaris. This result is especially significant because the Solaris test was run on a faster processor than the Lynx test. Figure 9 also shows that the latency of all types of synchronization mechanisms if roughly equal.

5.5.3 Test 3 (Priority inversion)

The results for the priority inversion test are shown in Figure 10 for all configurations. For all cases, except the lynx (lsem) case, a pthread mutex is used to guard the resource shared by the low and high priority tasks. Without a load the first Lynx configuration exhibits a latency corresponding to the delay time of the medium priority task of 10 milliseconds. This is due to the fact that in LynxOS 3.0.1 priority inheritance is not implemented for pthread mutexes. This problem is not seen with Lynx semaphores. Priority inheritance is implemented in Solaris, and the latency for all Solaris configurations, without a load, is low.

Under a heavy load only the lynx (lsem) and Solaris (1 rt) configurations exhibit an acceptable latency. The Solaris 1rt and 2 proc configurations are affected by the heavy load, and the lynx still has a high latency because of the lack of a priority inheritance protocol.

5.5.4 Context switching time

Table 9 shows context switching time for all platforms computed from the results for memory semaphores in the first two synchronization tests. The context switching time for Lynx is less than half the value of the best Solaris configuration. Also for Lynx the process-to-process context switching time is only slight worse than the thread-to-thread context switching time.

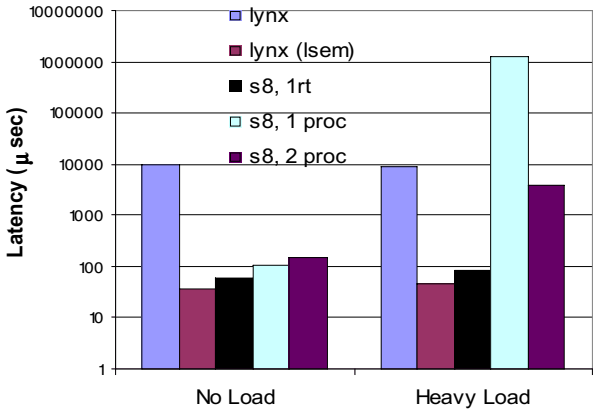

Figure 10: Sync Test 3: Lynx and Solaris (1 rt)

The context switching time for Solaris threads is more deterministic than the context switching time for processes. For the Solaris (1 rt) configuration the maximum thread-to-thread context switching time is close to average. However, for the same configuration, the process-to-process context switching time is an order of magnitude worse than the average value. Another interesting observation is that for Solaris the context switching time between processes is slightly better than between threads. In both cases there is a context switch between LWPs, this seems to imply that the bulk of the overhead is in the scheduler.

Table 9: Context switching times

Configuration	No Load				Heavy Load			
	Thread		Process		Thread		Process	
	Max	Avg	Max	Avg	Max	Avg	Max	Avg
Lynx	42.2	20.1	47.2	24.2	40.5	20.1	53.2	24.0
Solaris (1 rt)	65.4	52.9	446.8	49.9	67.2	51.8	461.0	50.6
Solaris (1 proc)	198.9	53.0	459.1	50.3	160.8	53.2	23240	51.4
Solaris (2 proc)	247.5	48.1	119.6	41.4	7149.0	68.7	639191	82.2

5.6 Communication

5.6.1 Real-time signals

Figure 11 shows the results of the real-time signal benchmark for all configurations. The Lynx configuration has a lower signal latency than any of the Solaris configurations. Also the Solaris 1 proc and 2 proc configurations are severely affected by the addition of a non-real-time load.

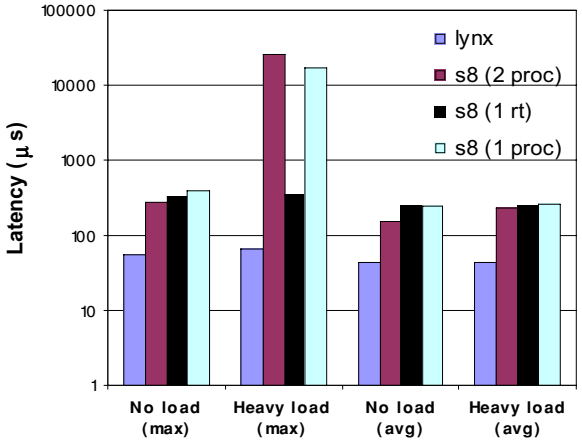

Figure 11: Real-time signal latency

Message queues
The latency and throughput of POSIX message queues for all configurations is shown in

Table 10 The latency for the Lynx platform is better than the Solaris platform, but the Solaris platform has better throughput. This better throughput is most likely due to faster hardware on the Solaris platform.

Table 10: POSIX message queues (No Load)

Configuration	Latency (μsec)				Throughput (Mbytes/sec)			
	Thread		Process		Thread		Process	
	Worst	Avg	Worst	Avg	Worst	Avg	Worst	Avg
Lynx	50.1	30.5	57.7	35.9	46.2	51.6	45.9	50.0
Solaris (1 rt)	98.7	90.5	118.9	102.7	62.4	77.8	61.5	76.5
Solaris (1 proc)	152.8	89.6	159.0	102.4	77.7	77.3	72.9	76.3
Solaris (2 proc)	148.7	82.8	146.8	77.5	41.3	66.6	58.2	65.5

6 Conclusion

In this paper we have assessed the use of POSIX in the development of software for real-time and embedded systems. We discussed the features of POSIX and how well these features match the features required in real-time software development. We also empirically evaluated the real-time performance characteristics of two implementations of POSIX: LynxOS 3.0.1 and Solaris 8.

The empirical evaluation showed that both LynxOS and Solaris 8 are suitable for real-time systems. LynxOS exhibited a low overhead for all operations and was deterministic even under heavy loading conditions. Solaris 8 contains a number of features that are important in real-time development, including: high resolution timers, processor partitioning, and SMP support. These last two features are key in Solaris's use as a real-time operating system. There is a dramatic difference in the determinism of the standard Solaris configuration versus a configuration were all real-time tasks are run on a dedicated processor. The standard configuration is unsuitable for real-time, whereas the second configuration is very deterministic.

Although this study did not perform an exhaustive comparison of the POSIX APIs between Solaris and LynxOS, our conclusion is that there is a great deal in common between the two implementations of POSIX. The biggest differences are in the areas of clock resolution and number of real-time priorities. Clock resolution could pose a portability problem if a resolution of greater than 10 milliseconds is needed. Other differences that we encountered, like discrepancies in the LynxOS threads implementation, are being rectified in version 3.1 of the operating system.

References

[1] IEEE/ANSI Std 1003.1: Information Technology-- (POSIX®)--Part 1: System Application: Program Interface (API) [C Language], includes (1003.1a, 1003.1b, and 1003.1c). 1996.

[2] IEEE Portable Applications. Available at: http://standards.ieee.org/catalog/posix.html.

[3] J.A. Stankovic. "Misconceptions About Real-time Computing." IEEE Computer. October 1988.

[4] E. Douglas Jensen. "Real-time for the Real World." Available at: http://www.real-time.org/.

[5] J.A. Stankovic and K. Ramamritham. "What is Predictability for Real-time Systems?" *Journal of Real-time Systems*, 2, 1990.

[6] D. Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates. 1991.

[7] 1003.1d Information Technology-- (POSIX®)--Part 1: System Application Program Interface (API)--Amendment x: Additional Real-time Extensions. 1999.

[8] 1003.1j-2000: Information Technology-- (POSIX®)--Advanced Real-time Extensions.

[9] 1003.21, LIS D3.0: Information Technology-- (POSIX®) RT Distributed Composite Insulators. 1999.

[10] B.O. Gallmeister. *Programming for the Real World, POSIX.4*. O'Reilly & Associates. 1995.

[11] 1003.1h D5, Draft POSIX® Part 1: System API Extension--RASS. 1999.

[12] National Institute of Standards and Technology, PCTS: 151-2, POSIX Test Suite.

[13] 1003.13-1998 IEEE Standard for Information Technology--Standardized Application Environment Profile (AEP)--POSIX® Real-time Application Support. 1998.

[14] B. Nichols, D. Buttlar, and J.P. Farrell. *Pthreads Programming*. O'Reilly & Associates. 1996.

[15] Scalable Real-time Computing in the Solaris™ Operating Environment. SUN White paper.

[16] W. Stallings. *Operating Systems*. Prentice-Hall, Inc. 1998.

[17] A. Cockcroft. "Processor Partitioning." Performance Q&A. SunWorld. 1998.

[18] J. Mauro. "The Solaris process model." Inside Solaris. SunWorld, available at: http://www.sunworld.com/common/f_swol-backissues-columns.html. 1998-99.

[19] The Lynx Real-time Operating System. Information available at http://www.lynuxworks.com/.

[20] William Weinberg. "Meeting Real-time Performance Goals with Kernel Threads."

[21] K. Obenland, T. Frazier, J.S. Kim, J. Kowalik. "Comparing the Real-time Performance of Windows NT to an NT Real-time Extension." *Proceedings RTAS*. 1999.

[22] H.J. Curnow, B.A. Wichmann. A Synthetic Benchmark. Computer Journal 19(1): 43-49. 1976.

[23] L. Monk, et al. "Real-time Communications Scheduling: Final Report." MITRE MTR 97B69. 1997.