An Overview of POSIX.4

Ozan S. Yigit

Department of Computer Science
York University

oz@cs.yorku.ca

ABSTRACT

POSIX.4 is a set of real-time extensions to POSIX.1. This document summarizes these extensions, with a brief summary of each function in various functional areas of these extensions.

Ancient Principle of **WYGIWYGAINGW:**
What You Get Is What You´re Given, And It´s No Good Whining.
--Terry Pratchett et al. (The Science of Discworld)

Introduction

POSIX, the Portable Operating System Interface, is a family of ANSI and ISO standards. The first of these standards, POSIX.1, (IEEE std. 1003.1) attempts to define a standard operating system interface based on the UNIX operating system documentation, to support application portability at the source level.

POSIX 4 (officially called IEEE Std 1003.1b-1993), approved in september 1993, is a set of real-time extensions to POSIX.1. This document summarizes these extensions, with a brief summary of each function in various functional areas of these extensions.

POSIX 4 uses the following definition of *realtime* for its scope:

> *Realtime in operating systems: the ability of the operating system to provide a required level of service in a bounded response time.* [1003.1b]

The key elements [P1003.1b] defining the scope are:

> *defining a sufficient set of functionality to cover a significant part of the realtime application program domain*
>
> *defining sufficient performance constraints and performance related functions to allow a realtime application to achieve deterministic response from the system*

and in addition:

> *defining interfaces that do not preclude high-performance implementations on on traditional uniprocessor realtime systems.*

POSIX.4 real-time extensions are grouped into several functional areas. The following are the real-time functional areas, and and their scope as defined in [P1003.1b]:

semaphores
a minimum synchronization primitive to serve as a basis for more complex synch mechanisms to be defined by the application program

process memory locking
a performance improvement facility to bind processes into the high-performance random access memory of the system. this avoids latencies introduced by the OS in storing unreferenced parts of the program in the secondary memory devices (eg. paging).

memory mapped files and shared memory
a performance improvement facility to allow for programs to access files as a part of the program images and for seperate app programs to have portions of their program image generally accessible

priority scheduling
a performance and determinism improvent facility to allow apps to determine the order in which processes that are ready to run are granted access to processor resources

realtime signal extension
a determinism improvement facility that augments the signal mechanism of historical implementations to enable asynchronous signal notifications to an application to be queued without impacting compatibility with the existing signal interface

timers
a functionality and determinism improvement facility to increase the resolution and capabilities of the time base interface

interprocess communication
a functionality enhancement to add a high-performance, deterministic interprocess communication facility for local communication.

synchronized input and output
a determinism and robustness improvement mechanism to enhance the data input and output mechanisms so that an application can insure that the data being manipulated is physically present on secondary mass storage devices

asynchronized input and output
a functionality enhancement to allow an application process to queue data input and output commands with asynchronousnotification of completion. [This facility includes in its scope the requirements of supercomputer applications]

We summarize each one of thes functional areas in turn, and show the programming interfaces for each area.

Semaphores

Posix semaphores provide a minimum synchronization primitive between multiple processes that share memory or mapped files. A semaphore allows guarded access to a resource or allows processes to wait for some change to happen. Posix semaphores are *counted* semaphores; Processes *wait* on or *post* a semaphore (Edsger Dijkstra's P (proberen) and V (verhogen) operations [UNPV22E]), decrement and increment a counter associated with the semaphore. POSIX.4 defines two types of semaphores: *named* and *unnamed* or *memory-based* semaphores. Named semaphores require names constructed like a normal file pathname. To run on all systems, a name must start with a "/" character (but may not contain other "/" characters in the name)[1]. A memory-based semaphore only requires a specific memory address.

1. These naming restrictions also apply to message queue and shared memory object names.

P1003.1b Semaphore Functions (`_POSIX_SEMAPHORES`)

`sem_init`
: Initializes an *unnamed* semaphore at a given location. [semaphore can be used for `sem_wait`, `em_trywait`, `sem_post` and `sem_destroy`]

`sem_open`
: Opens/creates a *named* semaphore. [semaphore can be used for `sem_wait`, `sem_try-wait`, `sem_post` and `sem_close`]

`sem_getvalue`
: Gets the value of a specified semaphore without affecting the state of the semaphore.

`sem_post`
: Unlocks a locked (named or unnamed) semaphore. If the resulting value is positive, then no processes were blocked waiting for the semaphore to be unlocked; the semaphore value is incremented.

`sem_wait`
: Performs a semaphore lock on a (named or unnamed) semaphore. If the semaphore value is zero, this function waits until it either locks the semaphore, or the call is interrupted by a signal.

`sem_trywait`
: Performs a semaphore lock on a (named or unnamed) semaphore only if it is currently not locked, that is the semaphore value is currently positive. Otherwise, it returns without waiting to lock the semaphore.

`sem_unlink`
: Removes a specified named semaphore. Processes that have the semaphore open can continue to use it. The semaphore will be actually removed only after all process close it.

`sem_close`
: Terminates access to a named semaphore. This does NOT remove a semaphore.

`sem_destroy`
: Destroys an unnamed semaphore at a given location. [Note that if the system detects other processes still using the semaphore, it may return EBUSY error for this call]

Process Memory Locking

Memory locking guarantees the memory residence of portions of the address space. Under POSIX.4, a process can lock all or a portion (range) of the pages mapped to its address space. These per-process memory locks are not inherited across a fork() and all memory locks owned by a process are unlocked upon exec() or process termination.

P1003.1b Memory Locking Functions (`_POSIX_MEMLOCK` and `_POSIX_MEMLOCK_RANGE`)

`mlockall`
: locks all of the pages mapped by a process's address space in physical memory. These pages cannot be paged or swapped to disk. [allows FUTURE locking]

`munlockall`
: Unlocks all currently mapped pages of a process's address space.

`mlock`
: Locks a specified section (a range between two addresses) of a process's address space in memory. These pages cannot be paged or swapped to disk.

`munlock`

Unlocks a specified (previously locked) section of a process's address space.

Memory Mapped Files and Shared Memory

Memory mapping establishes a mapping between the address space of the the process and a memory object represented by a file descriptor. The descriptor may point to an actual file, or it may be a shared memory (see `shm_open`) segment. [as result, a shared memory is first opened, and than mapped; what should take one step takes two. This is due to a previously existing `mmap` call] [UNPV22E]

P1003.1b Memory Mapping Functions (`_POSIX_MAPPED_FILES`)

`mmap`

establishes a virtual mapping between the address space of a process and a specified memory object. This allows the contents of the object to appear as a part of the process's memory.

`munmap`

removes any mappings for specified section of the address space.

`mprotect`

changes the access protections (RWEN) for a mapped section of the address space.

`msync`

synchronizes all modified data in the specified section of the address space with the underlying object. If the underlying object is a file, mapped section is written to permanent storage.

P1003.1b Shared Memory Functions (`_POSIX_SHARED_MEMORY_OBJECTS`)

`shm_open`

Opens or creates a shared-memory object and returns a file descriptor. [This descriptor is used with mmap to map the object into a process's address space]

`shm_unlink`

Destroys the named shared memory object, and removes the name of the shared-memory object. Processes which have opened or mmaped the object can still use it, until all processes close the object and munmap it.

Priority Scheduling

In a real-time operating environment, we would like to do one or more of the following: [POSIX.4]

- make sure something happens at or before a specific time
- make sure something happens before something else
- make sure something is not delayed if it is not designed to be delayed
- make sure scheduling guarantees are met

POSIX.4 provides priority-based scheduling policies. These are (conceptually) lists of processes, one list per priority. A POSIX.4 scheduling policy defines allowable operations (eg. moving processes between and within lists) on this set of lists. Associated with each policy is a priority range. POSIX.4 defines three schedule policies:

`SCHED_FIFO`

preemptive, priority-based scheduling. This is a common scheduling policy found in many real-time systems. this policy is usually implemented as an array of FIFO queues, one queue per priority level. Under this policy:

[1] when a running process is preempted, it becomes the head of the process list for its priority.

[2] when a blocked process becomes runnable, it becomes the tail of the process list for its priority.

[3] a running process can call `sched_setscheduler` to modify a specified process's policy and priority. if that process is running or runnable, it becomes the tail of the process list for its new priority

[4] a running process can call `sched_setparam` to modify the priority of a specified process's priority.

[5] when a running process calls `sched_yield`, it becomes the tail of the process list for its priority.

[6] at no other time does the position of a process in a process list with this policy is effected.

`SCHED_RR`
: preemptive, priority based round-robin scheduling with quanta. This is identical to SCHED_FIFO with the addition that each process has a time quantum. a running process gets preempted [and inserted at the end of the process list for the same priority level] if it runs longer than its quantum, and other processes of the same priority level are waiting in the queue. A process under this policy that is preempted and subsequently resumes execution completes the unexpired portion of its quantum.

`SCHED_OTHER`
: this is an implementation-defined scheduler.

P1003.1b Priority Scheduling Functions (`_POSIX_PRIORITY_SCHEDULING`)

`sched_getscheduler`
: Returns the scheduling policy (identifier `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER`) of a specified process

`sched_getparam`
: Returns the scheduling priority of a specified process

`sched_get_priority_max`
: Returns the maximum priority value allowed for a scheduling policy

`sched_get_priority_min`
: Returns the minimum priority value allowed for a scheduling policy

`sched_rr_get_interval`
: Returns the current quantum (timespec) for the round-robin scheduling policy

`sched_setscheduler`
: Sets the scheduling policy and priority of a specified process

`sched_setparam`
: Sets the scheduling priority of a specified process

`sched_yield`
: Yields execution to another process

Realtime Signal Extension

Signals are an integral part of the POSIX world for exception handling, process notifications, interprocess communication, etc, as defined in POSIX.1. In POSIX.4, a new range of signals are defined (`SIGRTMIN` to `SIGRTMAX`) for application use. POSIX.4 Signals can be queued and carry extra data, such as an integer or pointer data value. Real time signals are delivered in order, lowest numbered signal first. [so one would use the low numbers for higher-priority signals] and are received faster.

P1003.1b Realtime Signal Functions (`_POSIX_REALTIME_SIGNALS`)

`sigaction`
: Specifies the action a process takes when a particular signal is delivered (POSIX.4 adds `SA_SIGINFO`)

`sigqueue`
: Sends a a specified signal, plus identifying information, to a process. If the resources are available, the signal is queued for the receiving process.

`sigtimedwait`
: Waits for a signal for a specified amount of time and, if the signal is delivered within that time, returns the signal number and any identifying information provided by the signaller.

`sigwaitinfo`
: Waits indefinitely for a signal and, upon its delivery, returns the signal number and any identifying information provided by the signaller.

Timers

POSIX.4 defines a set of clock and timer functions that meet the requirements of many realtime applications. These clocks and timers are similar to those found in the Berkeley and AT&T UNIX systems, but with improvements: they support additional clocks [All POSIX.4 systems *must* support `CLOCK_REALTIME`], allow greater time resolution, implementation defined timers, and more flexible timer signal delivery.

P1003.1b Clock Functions (`_POSIX_TIMERS`)

`clock_getres`
: Returns the resolution of the specified clock. Every POSIX.4 system must support at least one clock, identified as `CLOCK_REALTIME`. This clock must support a resolution of at least 50Hz, or 20,000,000 nanoseconds.

`clock_gettime`
: Returns the current value for the specified clock

`clock_settime`
: Sets the specified clock to the specified value

P1003.1b Timer Functions (`_POSIX_TIMERS`)

`nanosleep`
: Causes the calling process to suspend execution for a specified number of nanoseconds. [This is a higher-resolution version of *sleep(3)*] The specified value may be rounded up to the resolution of the system clock.

`timer_create`
: Creates an interval timer based on a particular system clock. [Usually `CLOCK_REALTIME`] Timer_create Returns a unique timer ID used in subsequent calls to identify the

timer.

`timer_gettime`
: Returns the amount of time before the specified timer is due to expire and the repetition value (ie. the interval between succesive expirations)

`timer_settime`
: Sets the value of the specified timer to either an offset from the current clock setting or to an absolute time value.

`timer_delete`
: Removes a previously created timer, and frees up its resources.

`timer_getoverrun`
: Returns the timer expiration overrun count for the specified timer. [this is the number of timer expirations that occurred between the time timer expiration signal was queued, and the time at which the signal was delivered]

Interprocess Communication

POSIX.4 Message queues are intended as an flexible and efficient means of communication between multiple processes. It allows sending and receiving of messages, does message prioritization, and provides asynchronous process notification (for only *one* process). Processes can query the number of messages in the queue, the length of the queue or the maximum size of a message.

P1003.1b Message Functions (`_POSIX_MESSAGE_PASSING`)

`mq_open`
: Opens a named message queue. [Message queue naming rules are the same as semaphores]

`mq_getattr`
: Gets the attributes of a message queue.

`mq_notify`
: Requests that a process be notified when a message is available on an empty message queue

`mq_receive`
: Receives a message from the message queue

`mq_send`
: Sends a message on a message queue

`mq_setattr`
: Sets the attributes of a message queue. Only the flags attribute can be set, which includes a flag (MQ_NONBLOCK) that alters the behaviour of mq_receive.

`mq_close`
: Closes a message queue. Message queues are persistent; messages remain in the queue even after a queue is closed.

`mq_unlink`
: Removes a message queue. A message queue only goes away after all processes that have the queue open, close it.

Sychronized Input and Output

POSIX.4 provides for *synchronized* input and output. When I/O is synchronized, it is considered complete only when the underlying device is properly updated; For example, a *synchronized* write does not complete until the data is written to disk (or tape etc). [This is different than *synchronous* input and output, which just means I/O takes place while the caller waits.]

P1003.1b Synchronized I/O Functions (`_POSIX_SYNCHRONIZED_IO`)

`fcntl`
: Controls operations on files and memory objects. [especially `O_DSYNC` and `O_SYNC`]

`fdatasync`
: Flushes modified data only (possibly leaving the control information inconsistent) from the buffer cache, providing operation completion with data integrity.

`fsync`
: Flushes modified data and file control information from the buffer cache, providing operation completion with file integrity.

Asynchronized Input and Output

POSIX.4 Asynchronized input and output extensions provide the ability to perform I/O in parallel with other operations of an application, as needed in many real-time applications. When an asynchronous `read` or `write` call is issued, the operating system queues the request and immediately returns the control to the application. I/O is performed in parallel with the application. Optionally, the application can be notified for I/O completion with a signal. [important: if `_POSIX_PRIORITIZED_IO` and `_POSIX_PRIORITY_SCHEDULING` are defined, then asynchronous I/O is queued in priority order, using the current scheduling priority]

P1003.1b Asynchronous I/O Functions (`_POSIX_ASYNCHRONOUS_IO`)

`aio_cancel`
: Tries to cancel one or more asyncronous I/O requests pending against a file descriptor.

`aio_error`
: Returns the error status for a specified asynchronous operation.

`aio_fsync`
: Asynchronously writes system buffers containing a file's modified data to permanent storage.

`aio_read`
: Initiates an asynchronous read request on the specified file descriptor.

`aio_return`
: Retrieves the return status of a completed I/O operation.

`aio_suspend`
: Suspends the calling process until at least one of the specified asynchronous I/O requests has completed

`aio_write`
: Initiates an asynchronous write request to the specified file descriptor

`lio_listio`
: Initiates a list of I/O requests

References

[P1003.1B] IEEE Std 1003.1b-1993 IEEE Standard For Information Technology, *Portable Operating System Interface (POSIX) Part 1: System Application Program Interface, Amendment 1: Realtime Extension.*, IEEE Press, 1994, New York.

[POSIX.4] Bill O. Gallmeister, *POSIX.4: Programming For The Real World*, O'Reilly & Associates, Jan 1995.

[UNPV22E] W. Richard Stevens, *Unix Network Programming Volume 2: Interprocess Communications* (2nd ed) Prentice Hall, 1999.

Appendix

The following POSIX 1003.1b-1993 compile-time symbolic constants may be defined in `<unistd.h>`, and indicate which optional facilities are present.

```
_POSIX_ASYNCHRONOUS_IO
_POSIX_FSYNC
_POSIX_MAPPED_FILES
_POSIX_MEMLOCK
_POSIX_MEMLOCK_RANGE
_POSIX_MEMORY_PROTECTION
_POSIX_MESSAGE_PASSING
_POSIX_PRIORITIZED_IO
_POSIX_PRIORITY_SCHEDULING
_POSIX_REALTIME_SIGNALS
_POSIX_SEMAPHORES
_POSIX_SHARED_MEMORY_OBJECTS
_POSIX_SYNCHRONIZED_IO
_POSIX_TIMERS
```

As this list indicates, almost the entire real time functionality is optional. Only the real-time, queued signals are not optional.

Example feature checking code:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309
#include <unistd.h>
...

struct p4def {
        char *p4option;
        int support;
} p4sup[] = {
        { "asynchronous io",
#ifdef _POSIX_ASYNCHRONOUS_IO
        1
#else
        0
#endif
        },

        { "fsync",
#ifdef _POSIX_FSYNC
        1
```

```
#else
        0
#endif
        },
...
for (i = 0; i < TABSIZE; i++)
        printf("%s %s0, p4sup[i].p4option,
               p4sup[i].support ? "supported" : "not supported");
```

This feature checking code prints the following under Solaris 7:

```
asynchronous io: supported
fsync: supported
mapped files: supported
memlock: supported
memlock range: supported
memory protection: supported
message passing: supported
prioritized i/o: not supported
priority scheduling: supported
real-time signals: supported
semaphores: supported
shared memory: supported
synchronized i/o: supported
timers: supported
```

See Gallmeister's excellent book [POSIX.4] for more examples.