Scheduler Activations

Andy Tucker
PSARC 1996/021
Version 2.0

1 Introduction

Although the Solaris scheduler does a reasonable job of scheduling most applications, some important applications have particular requirements that require additional support. These include multithreaded (MT) applications using the Solaris threads library, and database servers.

The two-level model of Solaris threads creates a number of performance and functionality problems. The user-level threads scheduler controls the scheduling of application threads onto lwps. However, it has no real control over or knowledge of the scheduling of lwps onto CPUs. This disconnection makes it difficult to make appropriate scheduling decisions at user level.

Since there is no way to tell if an lwp has been preempted, the synchronization primitives cannot use adaptive algorithms similar to those used in the kernel (e.g., threads trying to acquire a mutex lock might spin if the thread holding the lock is running, but block if that thread is blocked). Priority inversion can easily occur when an lwp running a high priority thread is preempted in favor of another running a low priority thread. Thread scheduling based on processor cache affinity is difficult when lwps can migrate between processors at any time without the knowledge of the threads library. The "ideal" number of lwps for an application is difficult to determine and varies over time: too many, and there is excess context switching (causing cache displacement and locking problems); too few, and processors are left idle as the lwps block.

The current workarounds to these problems in Solaris are heavy-handed and inefficient. Priority inversion can be reduced by binding threads to lwps and placing the lwps in the real-time (RT) scheduling class. However, this is a waste of resources and is inappropriate for many applications (e.g., Java). If too few lwps are running an application, the SIGWAITING signal generated by the kernel assures that some forward progress will be maintained, but not that all available CPUs will be used to get maximal performance. Other problems have no workarounds at all.

A related problem occurs with non-multithreaded applications that synchronize between multiple lwps or processes. If an lwp holding a critical lock is preempted, other lwps and processes may be forced to wait for that lwp to be rescheduled. If the locks are spin-locks, it can be quite a while before the waiting lwps finish their time quanta and allow the lock owner to be rescheduled. This problem has been shown to cause major performance problems for database servers running at high loads. Preliminary studies show that preventing untimely preemptions in database servers results in a performance increase of 10-20% in peak performance and of over 200% at very high loads.

1.1 Scheduler Activations

Scheduler activations is an attempt to design a flexible and efficient mechanism for cooperation and communication between application and kernel. Data is shared between user and kernel to allow efficient communication of scheduling-related information. Kernel-to-user upcalls, based on doors [2], allow the kernel to directly "activate" lwps for use by an application.

The functionality provided by this project is hidden from the actual application by libraries. For MT applications using the Solaris threads (or POSIX threads) library, the functionality is made available without changing the library API. For non-MT applications such as database servers, the functionality is made available through a new library, libsched, created for this purpose.

1.2 Goals

The aim of scheduler activations is to support efficient and effective solutions to the following problems:

- *Adaptive mutexes*. Enable an lwp waiting for a mutex to determine whether or not the current holder of the mutex is running on a processor. If it is, the waiter should spin; if not, it should block, as it may be a long time before the mutex is released.
- *Preemption control*. For non-MT applications, allow lwps to block preemption for short periods of time. This is a great performance advantage for applications that use spinlocks (e.g., database servers), as they can prevent preemption while holding a lock.
- *CPU utilization*. When an lwp blocks, create or activate another lwp to run in its place. This keeps the CPU from going idle if there are insufficient runnable lwps available.
- *Thread priority.* Support a full thread priority model where if a threads application is running on *n* CPUs then the *n* highest priority threads will be running. Currently a high-priority thread may be running on an lwp that is preempted, while a low-priority thread continues running. This makes the user-level thread priority model somewhat ineffective.
- *Affinity scheduling*. Allow the use of scheduling based on processor or cache affinity at user level. This is impossible without knowledge of lwp migration between processors.
- *Dynamic lwp control*. Let the number of lwps used by an application vary dynamically as the need dictates. If CPUs are idle and there are runnable threads, create additional lwps. If lwps are being preempted, or if there are idle lwps, reduce the number of active lwps.

2 Internal Interfaces

Most of the interfaces introduced for scheduler activations are consolidation-private internal interfaces between the kernel and ON consolidation libraries (libthread and libsched). They are described in this section to help explain the project design and to delineate issues that might impact the system as a whole. The new public interfaces are discussed in Section 3.

2.1 _lwp_schedctl() System Call

A new system call, _lwp_schedctl(), is used to initialize the scheduler activations functionality for an lwp, and to change flags for an lwp with scheduler activations already initialized. The format of the call is:

```
int _lwp_schedctl(lwpid_t lwp, unsigned int flags,
     int upcall_did, schedctl_state_t **statep)
```

`lwp` is the id of the affected lwp. One lwp may set or change the scheduler activations properties of another, but only within the same process.

The flags argument can contain one or more of the following flags: SC_STATE, SC_PREEMPT, SC_BLOCK, and SC_PRIORITY. The flags control kernel operations as follows.

- **SC_STATE:** If this is set the kernel will maintain data indicating whether or not the lwp is currently running on a CPU, and on which CPU it is running or last ran. This data is continually copied into a location accessible from user level whenever the lwp blocks or is preempted or scheduled. This is used to implement adaptive mutexes, among other things.
- **SC_PREEMPT:** If this is set, the kernel will avoid preempting the lwp when it has indicated (via the shared data below) that it should not be preempted. If a preemption is blocked, the lwp should yield the next time it is safe to do so. Failure to yield when requested will result in a reduced priority. SC_PREEMPT is used to implement preemption control via libsched.
- **SC_BLOCK:** If this is set, the kernel may perform an *upcall* when the lwp blocks. An upcall involves switching directly to another lwp in the process. The new lwp begins running at the entry point to the door identified by upcall_did. In the case of the threads library, the entry point is the "idle loop" of the library; the new lwp looks for threads to begin executing.
- **SC_PRIORITY:** If this is set, the kernel will perform the necessary actions to maintain proper thread priority semantics for the process, based on the data supplied by the threads library. This is currently implemented by forcing an lwp running a low priority thread off a CPU if one running a high priority thread is preempted, but it may be implemented in other ways in the future. Note that the vast majority of threads applications, where all threads have the same priority, will not need this feature. It is turned on by the threads library when a thread of non-default priority is scheduled.

upcall_did is the file descriptor of the door created to support upcalls when SC_BLOCK is set.

statep is a pointer where, on successful completion of the system call, the location of data used for communicating scheduling information about this lwp (see Section 2.2) will be stored.

_lwp_schedctl() returns 0 on success and an error code on failure:

- EINVAL if the passed flags are invalid
- EBADF if the passed door descriptor is invalid
- ESRCH if the lwp id is invalid
- EFAULT if the statep pointer is invalid

_lwp_schedctl() may be called multiple times with the same lwp to change the current flag settings.

2.2 Scheduling Data

The _lwp_schedctl() call returns an pointer used by the threads library to access lwp state data. The data is stored in non-pageable kernel memory (it must be non-pageable since it is accessed by the kernel at times it cannot take a page fault). The page containing the data is automatically mapped into the process's address space on successful return from the _lwp_schedctl() call. The data is available on the basis of effective uid; that is, two processes with the same effective uid may receive mappings to the same kernel page, although the pointers within that page returned by _lwp_schedctl() will be unique to each process. This provides a way of protecting processes of different users from each other, while allowing multiple processes of a single user to have lwp data on the same set of pages (saving kernel memory).

The data for each lwp has the structure:

```
typedef struct schedctl_state {
        int sc_state;
        processorid_t sc_cpu;
        int sc_priority;
        schedctl_t sc_public;
} schedctl_state_t;
```

Where schedctl_t is the public part of the data exposed by the libsched interface (see Section 3) and has the structure:

```
typedef struct schedctl {
        short sc_nopreempt;
        short sc_yield;
} schedctl_t;
```

- The sc_state field is set by the kernel and contains the current state of the lwp: running, runnable, stopped, sleeping, and free (or zombie).
- The sc_cpu field is set by the kernel and contains the last CPU on which the lwp was scheduled (or the current one if the lwp is running).
- The sc_priority field is set by the user to indicate the priority of the thread currently scheduled on the lwp.
- The sc_nopreempt field is used to indicate that preemption of the lwp should be avoided if possible. This is used to support the SC_PREEMPT flag.
- The sc_yield field is set by the kernel to indicate that the lwp should yield at the next chance. This is set if the kernel wanted to preempt the lwp but it was blocked by the sc_nopreempt field.

2.3 Upcalls

Kernel to user upcalls are used to provide additional lwps to the threads library. This is intended to replace the current use of SIGWAITING. In the current threads library, when all lwps in a process block in the kernel, a SIGWAITING signal is sent to one of the lwps. The SIGWAITING handler in the threads library then creates an additional lwp to avoid deadlocks. With upcalls, the threads library can keep an extra lwp in a door pool associated with the process. When all lwps in the process block, that lwp can be “activated” via an upcall. The threads library can then run on that lwp (creating an additional lwp if the door pool is now empty) without the signal overhead of the current approach. Upcalls may also be generated when a processor is idle and there are threads waiting to run on the threads run queue.

3 External Interfaces

Multithreaded applications can take advantage of scheduler activations without any change to the applications. The functionality is provided “under the covers” without changes to the libthread interface. However, non-multithreaded applications must be modified to use the "preemption control" facility of scheduler activations. A new library, libsched, provides an uncommitted ("evolving") public interface for this use.

3.1 libsched

NAME

schedctl_init, schedctl_lookup, schedctl_exit, schedctl_start, schedctl_stop - preemption control

SYNOPSIS

cc [flag ...] file ... -lsched [library ...]

#include <schedctl.h>

schedctl_t *schedctl_init(void);

schedctl_t *schedctl_lookup(void);

void schedctl_exit(void);

void schedctl_start(schedctl_t *);

void schedctl_stop(schedctl_t *);

DESCRIPTION

These functions provide limited control over the scheduling of a lightweight process (LWP). They allow a running LWP to give a hint to the kernel that preemptions of that LWP should be avoided. The most likely use for these functions is to block preemption while holding a spin-lock for a short period of time. Improper use of this facility, including attempts to block preemption for sustained periods of time, will result in reduced performance.

schedctl_init initializes per-LWP scheduling control data for the calling LWP and returns a pointer used to refer to the data. If **schedctl_init** is called more than once by the same LWP, the most recently returned pointer is the only valid one.

schedctl_lookup returns the currently allocated scheduling control data pointer associated with the calling LWP.

schedctl_exit removes the scheduling control data associated with the calling LWP.

schedctl_start is a macro that gives a hint to the kernel scheduler that preemption should be avoided on the current LWP. The pointer passed to the macro must be the same as the pointer returned by the call to **schedctl_init** by the current LWP. The behavior of the program when other values are passed is undefined.

schedctl_stop is a macro that removes the hint that was set by **schedctl_start**. As with **schedctl_start**, the pointer passed to the macro must be the same as the pointer returned by the call to **schedctl_init** by the current LWP.

schedctl_start and **schedctl_stop** are intended to be used to bracket short critical sections, such as the time spent holding a spinlock. Other uses, including the failure to call **schedctl_stop** soon after calling **schedctl_start**, may result in poor performance.

RETURN VALUES

schedctl_init returns a pointer to a **schedctl_t** structure if the initialization was successful, or **NULL** otherwise. **schedctl_lookup** returns a pointer to a **schedctl_t** structure if the data for that lwp was found, or **NULL** otherwise.

ERRORS

None returned

SEE ALSO

priocntl(1M), priocntl(2)

NOTES

Scheduling control is intended for use by LWPs belonging to the time-sharing (TS) and interactive (IA) scheduling classes. If used by LWPs in other scheduling classes, such as real-time (RT), no errors will be returned but **schedctl_start** and **schedctl_stop** will not have any effect.

Use of scheduling control by unbound threads in multithreaded applications (see **thr_create**(3T)) is not supported and will result in undefined behavior.

If a process containing LWPs using scheduling control performs a **fork(2)**, and the child does not immediately call **exec(2)**, the each LWP in the child must call **schedctl_init** again prior to any future uses of **schedctl_start** and **schedctl_stop**. Failure to do so will result in undefined behavior.

3.2 schedctl.h

The schedctl.h header file contains the definition of the schedctl_t structure, macros for schedctl_start() and schedctl_stop(), and prototypes for schedctl_init(), schedctl_lookup(), and schedctl_exit().

The schedctl_t structure is described in Section 2.2.

The schedctl_start() macro simply sets the sc_nopreempt field to a nonzero value. The schedctl_stop() macro clears the sc_nopreempt field and checks the sc_yield field. If sc_yield is set, it calls the yield() system call. Macros are used instead of function calls for performance.

3.3 Interoperability with libthread

The libsched functionality will work in conjunction with libthread. However, the results may be unpredictable when used from within unbound threads, as the pointer corresponding to the current lwp must be passed to schedctl_start() and schedctl_stop(). Since unbound threads can migrate between lwps without the knowledge of the application, there is now way to ensure that the pointer passed to schedctl_start() corresponds to the current lwp on which an unbound thread is running. Hence, use of the libsched macros by unbound threads is not supported.

References

[1] Andy Tucker. Scheduler Activations Design. Online at http://trinity/sact/design.ps.

[2] Jim Voll. SpringOS Doors in Solaris. PSARC case 1993/620. Online at http://pacific/home/jjv/doors/doc/white_paper.ps.