Improving the FreeBSD SMP implementation

Greg Lehey

JaeMoon Seo
jmseo@rtcc.hanyang.ac.kr

Real-Time Computing and Communications Lab.
Hanyang University
Contents

- The rules of traditional UNIX
  - Why the UNIX kernel is non-preemptive?

- Adapting the UNIX model to SMP
  - The initial FreeBSD model
  - Synchronization primitives

- The new FreeBSD implementation
  - Interrupt threads
  - Locking constructs
  - Removing the Big Kernel Lock

- Conclusion
The rules of traditional UNIX

- **There is only one**
  - All code runs on it

- **If both an interrupt handler and a process are available to run**
  - The interrupt handler runs

- **Interrupt handlers have different priorities**
  - The higher priority interrupt immediately preempts the lower priority interrupt

- **The scheduler chooses the highest priority process which is ready to run**
  - Have a different priority

- **If the process is in kernel mode**
  - When its time slice expires or a higher priority process becomes runnable
  - The system wait until in returns to user mode or sleeps before running the scheduler
Why the UNIX kernel is non-preemptive?

- To protect the shared data in the kernel
  1. All processes share the kernel address space
  2. All processes in the kernel can run kernel code and modify kernel data structures
  3. The shared data structure could be corrupted

```c
struct process *free_process_list;

During fork()
1. struct process *child = free_process_list;
2. free_process_list = free_process_list->next;
3. return child;
```

If two processes run step ① concurrently?

Two child processes share a same process structure
Synchronizing processes and interrupts & Protection

- Non-preemption rule only applies to processes
  - Interrupts happen independently of process context
  - The process context and the interrupt context must share data

- Protection
  - UNIX solves this problem by locking out interrupts
    - During critical sections
  - Blocking interrupts has a potential danger
    - Interrupt will not be serviced in a timely fashion
Adapting the UNIX model to SMP

- More than one processor is available
  - Code can run in parallel

- Interrupt handlers and user processes can run on different processors at the same time

- The “non-preemption” rule is no longer sufficient to ensure that two processes can’t execute at the same time
  - It would theoretically be possible for two processes to allocate the same memory

- Locking out interrupts must happen in every processor
  - This can adversely affect performance
The initial FreeBSD model

- **For reliability rather than performance**
  - Only one process could run in the kernel at any one time
    - By spin-lock (called Big Kernel Lock)
      - Unix Interrupt handlers can not block
    - Busy wait for holding lock consumes CPU time

- **Problem**
  - **BKL could degrade SMP performance**
    - If user processes on each multi cores call kernel concurrently?
      - The execution of other processes would be delayed because they don’t relinquish their CPU
Synchronization primitives (1)

- Spin locks
  - Controls a single resource
  - “busy wait” when lock is not available
  - Almost used in interrupt handler

- Semaphores
  - Oldest synchronization primitives
  - Include a count variable which defines how many processes may access the resource in parallel
  - “blocking” when lock is not available

- Read-write lock
  - Allows multiple readers or alternatively one writer
Synchronization primitives (2)

- **Conditional variable**
  - Tests an external condition
  - If it is not met?
    - block
  - If the condition is met?
    - all processes sleeping on the wait queue are woken

- **Big problem**
  - Locking primitives with the exception of spin locks is that they can block
  - This requires a process context
    - UNIX interrupts handler can’t block
The new FreeBSD implementation

- SMPng
  - The name of the new FreeBSD SMP version
  - “ng” means “new generation”

- The most radical difference in SMPng
  - Interrupt code (“bottom half”) runs in a process context
    - This process context is termed an interrupt thread
  - Interrupt lockout primitives have been removed
    - i.e., spl for masking interrupt
    - The system uses mutexes instead of locking out interrupts
Interrupt threads (1)

- **Interrupt threads**
  - Are *kernel threads*
    - for giving a *process context* to interrupt handler (bottom half)
    - i.e., stack, process control block
  - Allows a uniform approach to synchronization for both halves
    - *blocking* interrupt handler by using *mutexes*
    - The *spl* primitives are no longer needed
  - Does not allow preemption if the process is running in kernel mode
    - The UNIX approach to scheduling
Interrupt threads (2)

Interrupt threads (cont.)

- Are scheduled by *lazy scheduling*
  1. On receiving an interrupt
  2. Note the PID of the interrupt thread, do not schedule the thread
  3. Continues execution in the context of the interrupted process
  4. The thread will be scheduled
      - When the thread has to block
      - When the interrupt nesting level gets too deep
- Run at a higher priority than all user processes
- Have a fixed priority at anytime
Interrupt threads (3)

Interrupt threads (cont.)

- From a scheduling viewpoint, the threads differ from normal processes
  - Never enter user mode
  - All share the address space of Process 0
  - Run at a higher priority than all user processes
  - Have a fixed priority at anytime
  - SWAIT has been introduced for interrupt processes which are currently idle
Interrupt threads (4)

The initial implementation
- Very similar to that normal processes
- The benefit
  - relatively easy debugging and of stability
- The disadvantage
  - a significant drop in performance because of context switches
Interrupt threads (5)

- The improved implementation (Not completed yet)
  - Allows limited kernel preemption
    - Allowing interrupt threads to be scheduled immediately
    - Do not wait for the current process to leave kernel
  - The potential exists for complete kernel preemption
    - Where any higher priority process can preempt a lower priority process running in the kernel
  - The final lazy scheduling implementation has been tested
    - No significant performance increase because of the current kernel lock implementation
  - No all interrupts have been changed to threaded interrupts
    - The old fast interrupts remain relatively unchanged
      - Not using any blocking mutexes
      - i.e., The serial drivers
Locking constructs (1)

- **Spin/Sleep mutex**
  - Default locking construct
  - Is similar in concept to the binary semaphore
  - Allows spinning for a certain period of time
    - If this appears to be of benefit
  - Allows the user to specify that the mutex should not spin
    - It is placed on a sleep queue and woken when the resource becomes available

- **Spin mutex**
  - Use only in exceptional cases
Locking constructs (1)

- Condition variables
  - Acquire a condition variable with cv_wait() families
  - The associated mutex must be held before acquiring this
  - Unblock one waiter with cv_signal() and all waiter with cv_broadcast()
Locking constructs (2)

- Shared/Exclusive locks
  - Another name for reader/writer locks
  - Create an sx lock with sx_init()
  - Attain a read lock with sx_slock() and release it with sx_sunlock()
  - Attain a write lock with sx_xlock() and release it with sx_xunlock()
  - Destroy an sx lock with sx_destroy

```c
struct sx {
    struct lock_object sx_object; /* Common lock properties. */
    struct mtx    sx_lock;   /* General protection lock. */
    int     sx_cnt;         /* -1: xlock, > 0: lock count. */
    struct cv sx_shrd_cv;   /* Slock waiters. */
    int     sx_shrd_wcnt;   /* Number of slock waiters. */
    struct cv sx_excl_cv;   /* Xlock waiters. */
    int     sx_excl_wcnt;   /* Number of xlock waiters. */
    struct proc *sx_xholder; /* Thread presently holding xlock. */
};
```
Removing the Big Kernel Lock

- **Big Kernel Lock (BKL)** is replaced with
  - **Giant**
    - Is used in a similar manner to the *BKL*
    - Is a **blocking mutex**
    - Protects all entry to the kernel including interrupt handlers
  - **sched_lock**
    - Is a spin lock for protecting the scheduler queues

- The combination of locks
  - supplied the bare minimum of locks necessary to build the new framework
  - Does not improve the performance of the system yet
    - Because of Giant blocking
Migrating to fine-grained locking

- Replacing thh BKL with Giant and schedlock
- Developed the following guidelines for locking
  - Use sleep mutexex
  - Do not tsleep() while holding a mutex other than Giant
  - Do not msleep() or cv_wait() while holding a mutex other than Giant
  - Do not call a function that can grab Giant and then sleep unless no mutexes
  - If calling msleep() or cv_wait() while holding Giant and another mutex, Giant must be acquired first and released last
  - Except for the Giant mutex used during the transition phase
  - Do not msleep() or cv_wait() with a recursed mutex
  - Try to hold mutexes for as little time as possible
  - Try to avoid recursing on mutexes if at all possible
Conclusion

- The process synchronization of UNIX-derived OS
  - Is unsuited to multiprocessor application
  - Allows only one process to run in kernel mode at any time
  - Cause seriously suboptimal performance of I/O bound systems

- The improving FreeBSD SMP implementation
  - Performed synchronization primarily by a locking construct
    - Between a spinlock and a binary semaphore termed *mutexes*
  - Addressed the issue of blocking interrupt handlers
    - By attaching a process context to the interrupt handlers

- Further reference