A Survey of Parallel Processing in Linux

Kojiro Akasaka
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
kojiro.akasaka@sjsu.edu

ABSTRACT

Any kernel with parallel processing support must protect shared data for data integrity, and adjust workload of multiple processors to control system performance.

To fulfill these responsibilities, Linux had to undergo many careful considerations. Different locking mechanisms are appropriately used in different places for performance, and different scheduling algorithms, and a new load balancing scheme have been introduced to control performance.

This paper discusses concepts and implementations that concern parallel processing support in Linux: atomicity, synchronization techniques, and load balancing.

1. INTRODUCTION

1.1 Concurrency

Parallel processing refers to the execution of two or more machine instructions on a given system at the same instant. The term readily concerns hardware developers because parallel processing on a computer system is directly associated with extra hardware, that is, multiple processing units.

Comparatively, concurrency is the progress of multiple tasks at the same time period. The progress doesn't need to be instantaneous. Concurrency is a concept, more abstract than parallel processing, and therefore is the term that concerns software developers better. A computer system with concurrency doesn't have to be implemented with parallel processing.

The terms parallel processing and concurrency can be confusing because they have the same synchronization issues or because many concurrent systems nowadays are parallel processing systems, but there is a distinction.

1.2 Kernel Preemption

The capability of parallel processing is not the only way to the possibilities of concurrency issues in Linux; kernel preemption is another way.

Kernel preemption refers to preemption of a sequence of instructions to be executed in kernel mode. Kernel preemption is similar to process preemption in that it refers to swapping out the currently executing control flow in a processor to swap in another control flow to the same processor.

1.3 Linux

Parallel processing is apparently the current trend in making high performance computer systems. Of course, realizing parallel processing is not as simple as adding many processors on the hardware end; software such as Linux needs to be changed as well.

Linux is a kernel - the lowest level software that can coordinate all software and hardware activities of a computer system.

In Linux, symmetrical multiprocessing and kernel preemption were introduced with version 2.0 and version 2.6, respectively. Thus, the latest version of Linux is also capable of both symmetrical multiprocessing and kernel preemption.

Linux uses preprocessor macros in its various parts of the code to toggle the inclusion and exclusion of fields or operations for SMP.

For example, the source file for Linux core scheduler includes or excludes, depending on kernel configuration, SMP related functions starting from the line 977 and to the line 1256.

All kernel source code contained or mentioned therein is based on Linux version 3.13.8, which was the latest version when this paper was written.

1.3.1 SMP

SMP stands for Symmetric Multi-Processing, and used to denote a form of a multi-processor system. The role and the peripheral configuration of each processor in a SMP system is symmetrical.

This means parallel processing is free from any bias toward one processor on the system, and can require a simpler implementation than Asymmetric SMP.

Linux is based on the SMP model; however, it doesn't mean it cannot be run on a ASMP system.

1.4 Race Condition

A race condition refers to the condition that allows incorrect result depending on the timing of events.

At the processor level, a race condition is caused by non-atomic CPU instructions because it can take more than one clock cycle to complete the operation.

In Intel 64 and IA-32 architectures, an increment instruction to main memory is not atomic. This can cause a race condition:

One CPU on a multi-processor system executes an increment instruction to a value stored in a memory location and reads the value in that location. Another CPU executes an increment instruction to the same memory location and reads the same value as read by the first CPU because the second CPU read it before the first CPU hasn't finished writing the incremented value.

2. Atomicity

To prevent race conditions in Linux, the first and foremost necessity is atomicity, at the hardware level, to ensure that no race condition can exist. No race can exists among multiple processors because atomic CPU operations cannot be interleaved.

In Linux, atomicity is the basis for implementing most of the locking mechanisms discussed next.

However, implementing atomicity costs clock cycles. So, it shouldn't be used where unnecessary.

3. Locking Synchronization

In operating systems with support for parallel processing or preemption of control flows, care must be taken to avoid unsafe access to shared data; otherwise, data corruptions that can lead to system failures result.

There are different synchronization mechanisms available in Linux because there is no single mechanism that is suitable for all situations.

Linux includes different synchronization mechanisms in different places in order to increase system responsiveness and to avoid overhead.

3.1 Spin Lock

A spin lock is a lock that can consume CPU cycles endlessly until signaled by the corresponding function, and isn't designed to be used in uni-processor systems where parallel processes can't exist. Also, it should only be used if the wait is known to be short to avoid wasting CPU clock cycles and bus traffic.

The state of a spin lock can be either unlocked or locked. When a spin lock is found in unlocked state, it can be acquired immediately; then, the control flow is mutually exclusive to any other flows, and safe to access the target shared data. If it's found in locked state, the control flow needs to be stopped at that point, and spins or wastes CPU clock cycles until it's unlocked by the control of the lock holder.

In Linux, spin locks have an additional flag that's used to let the process holding the spin lock know that there are other processes waiting to acquire the lock. This is done so that the holder can release the lock temporarily to reduce the latency in acquiring the lock.

Linux on a uni-processor system implements the spin lock's lock or unlock function to merely disable or enable kernel preemption, respectively. This is because in such a system only kernel preemption can cause race conditions.

A spin lock is one of the most used locks in Linux. For example, it's used to protect the region of the __scheduler() function from concurrent access.

Linux 3.13.8 kernel/sched/core.c __schedule

```
raw_spin_lock_irq(&rq->lock);


switch_count = &prev->nivcsw;
if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
        if (unlikely(signal_pending_state(prev->state, prev))) {
                prev->state = TASK_RUNNING;
        } else {
                deactivate_task(rq, prev, DEQUEUE_SLEEP);
                prev->on_rq = 0;

                /*
                 * If a worker went to sleep, notify and ask workqueue
                 * whether it wants to wake up a task to maintain
                 * concurrency.
                 */
                if (prev->flags & PF_WQ_WORKER) {
                        struct task_struct *to_wakeup;

                        to_wakeup = wq_worker_sleeping(prev, cpu);
                        if (to_wakeup)
        try_to_wake_up_local(to_wakeup);
                        }
                }
                switch_count = &prev->nvcsw;
}

pre_schedule(rq, prev);

if (unlikely(!rq->nr_running))
        idle_balance(cpu, rq);

put_prev_task(rq, prev);
next = pick_next_task(rq);
clear_tsk_need_resched(prev);
clear_preempt_need_resched();
rq->skip_clock_update = 0;

if (likely(prev != next)) {
        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;

        context_switch(rq, prev, next); /* unlocks the rq */
        /*
         * The context switch have flipped the stack from under us
         * and restored the local variables which were saved when
         * this task called schedule() in the past. prev == current
         * is still correct, but it can be moved to another cpu/rq.
         */
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
} else
        raw_spin_unlock_irq(&rq->lock);
```

3.2 Read/Write Spin Lock

A read/write spin lock is a modified version of a spin lock.

By using a counter as its locking state, it allows multiple readers to access to the shared data simultaneously; however, a writer still needs to be mutually exclusive to readers to modify shared data.

In Linux, the read/write spin lock uses the counter variable as an indicator to allow or block an attempted access. the counter of the lock is used by a writer or a reader to test, acquire, or release the lock. Multiple readers can obtain the same lock by decreasing its counter field value.

In case there are many readers at a given time, the read/write version can be preferred to the normal spin lock.

A potential performance problem is that a flow of readers can starve a writer because it needs to wait for all readers to release the lock.

3.3 Seq Lock

A seq lock, or sequential lock, is a lock that allows a writer while there are readers, and is another version of a spin lock in that a process spins when the lock is contended.

A difference between a seq lock and a read/write spin lock is that a seq lock doesn't enforce the mutual exclusiveness for a writer.

In Linux, whenever a writer gets or releases the lock, the seq lock's counter is incremented so as to let the reader or readers holding the lock know that the data has been accessed by a writer and the data needs to be reread.

The benefit of a seq lock is that a writer can proceed while one or more readers are still holding the lock; in other words, a writer is never starved by readers.

A trade-off for this, however, is the overhead of the rereads by a reader when multiple writers consecutively acquire the lock before the reader releases the lock. The below Linux source code illustrates the use of a seq lock by a reader.

kernel/time/jiffies.c get_jiffies_64

```
#if (BITS_PER_LONG < 64)
u64 get_jiffies_64(void)
{
        unsigned long seq;
        u64 ret;

        do {
                seq = read_seqbegin(&jiffies_lock);
                ret = jiffies_64;
        } while (read_seqretry(&jiffies_lock, seq));
        return ret;
}
EXPORT_SYMBOL(get_jiffies_64);
#endif
```

3.4 Semaphore

When the wait time is known to be long, a lock called semaphore can be used. A semaphore, unlike a spin lock, allows the process that fails to acquire the lock to sleep.

A semaphore works by incrementing or decrementing its atomic counter. Unlike a spin lock, a semaphore allows multiple processes to be in the critical region. The initial value of the counter indicates how many processes can enter the region protected by the semaphore. When this value is one, the lock is called mutex, and ensures mutual exclusion.

Because a process can be blocked by the semaphore, Linux includes the wait field to hold a list of tasks waiting to get the lock.

Like spin lock, Linux implements a read/write version of semaphore that allows concurrent readers.

4. Other Synchronization

4.1 Completion Variable

A completion variable is used for a task or tasks to wait or sleep until a certain event occurs.

For this, the variable contains the field of a FIFO queue of tasks to be awakened.

A semaphore can be contended to make the same effect of making processes wait. However, it's not the intended use, and it's not optimized for the contended case, therefore completion variable was created.

kernel/fork.c complete_vfork_done

```
static void complete_vfork_done(struct task_struct *tsk)
{
    struct completion *vfork;

    task_lock(tsk);
    vfork = tsk->vfork_done;
    if (likely(vfork)) {
        tsk->vfork_done = NULL;
        complete(vfork);
    }
    task_unlock(tsk);
}
```

4.2 Read Copy Update

An RCU mechanism provides synchronization without overhead of traditional locks.

An RCU is a mechanism that allows multiple readers and a writer simultaneously. In that sense, it's yet another version of a read/write spin lock, and, unlike a seq lock, it allows multiple writers to proceed.

An RCU can be used to synchronize pointers to a data structure across readers and a writer.

For example, in the following code run by a writer, the order between the first assignment and the last assignment matters. If the compiler changed the order so that the last assignment occurs before the first assignment, concurrent readers can read the old data [7].

```
p->a = 1;
gp = p;
```

This situation can be prevented by an RCU function as following:

```
p->a = 1;
rcu_assign_pointer(gp, p);
```

4.3 Per-CPU Variable

Per-CPU variable is a variable that has an entry for each CPU of the system.

Whenever data is accessed by one specific CPU of the system, it makes sense to declare such data as part of a per-CPU variable.

The advantages of using per-cpu variables are:

It has simpler implementation; because there is no unsafe access to the data from parallel processing CPUs, no need to protect the code that accesses per-CPU variable.

It has better performance; since only one CPU can access the data, it doesn't involve cache snooping overhead.

However, a rescheduling resulted from kernel preemption can still create problems of incorrect behaviors. Thus, disabling kernel preemption is one way to safely access the data.

5. Cache Snooping

A shared memory, multi-processor system can have one or more per-processor caches.

The processor model of the computer on which this paper was written is Intel® Core™ i7-4750HQ. It has four hyperthreaded cores with a total of 8 threads.

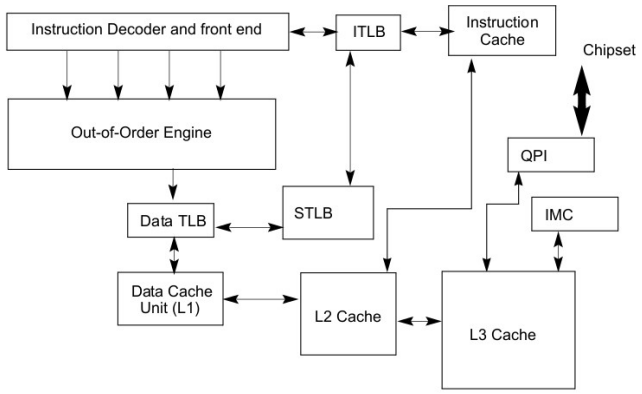

Figure 1. Cache Diagram For Intel Core i7 [3].

Because the memory is shared, and caches are not shared among processors, there can be a case that caches are not consistent, and need to be updated.

Cache snooping is a technique that can be used in this situation to synchronize caches.

6. Load Balancing

In computing, load balancing is distribution of the workload of a computer system across its available resources.

Load balancing can be applied to different types of resources in different contexts. For example, in big data management, a resource can be as big as a computer node.

In the context of Linux scheduler subsystem, the general goal of load balancing is that the computer system, at any given moment, takes maximum computational power of available CPUs via parallel processing.

This is done by moving runnable processes on a given CPU to a non-busy CPU. Evidently, if one CPU has many runnable processes while other CPUs are idle, load balancing needs to be done.

Linux periodically both with a timer tick and with some other entry points tries to apply load balancing at various points.

Firstly, it's done on every timer tick. Each processor's timer device triggers a timer interrupt which calls the corresponding timer handler.

kernel/timer.c

```
/*
 * Called from the timer interrupt handler to charge one tick to the current
 * process.  user_tick is 1 if the tick is user time, 0 for system.
 */
void update_process_times(int user_tick)
{
        struct task_struct *p = current;
        int cpu = smp_processor_id();

        /* Note: this timer irq context must be accounted for as well. */
        account_process_tick(p, user_tick);
        run_local_timers();
        rcu_check_callbacks(cpu, user_tick);
#ifdef CONFIG_IRQ_WORK
        if (in_irq())
                irq_work_run();
#endif
        scheduler_tick();
        run_posix_cpu_timers(p);
}
```

6.1 Implementation Considerations

One caveat for load balancing is that the cost of moving a process from CPU A to CPU B can differ from the cost of moving it to CPU C. The cost is such as latency, the overhead of remote memory access or cache invalidation.

For example, there is less latency in scheduling a process across CMP cores than moving it across NUMA nodes. If an active process is migrated to another processor, it's no longer cache hot in the new processor and must reestablish the cache.

Linux supports SMT processors, SMP systems as well as NUMA architectures, so Linux considers both timing and the processor topology of the system to try to avoid such costs.

For example, the number of cache invalidation overheads can be lowered by invoking the load balancer when a new process needs to be scheduled. Linux implements sched_exec for this end.

kernel/sched/core.c sched_exec

```
/*
 * sched_exec - execve() is a valuable balancing opportunity, because at
 * this point the task has the smallest effective memory and cache footprint.
 */
void sched_exec(void)
{
        struct task_struct *p = current;
        unsigned long flags;
        int dest_cpu;
```

```
        raw_spin_lock_irqsave(&p->pi_lock, flags);
        dest_cpu = p->sched_class->select_task_rq(p, task_cpu(p),
                                                  SD_BALANCE_EXEC, 0);
        if (dest_cpu == smp_processor_id())
                goto unlock;

        if (likely(cpu_active(dest_cpu))) {
                struct migration_arg arg = { p, dest_cpu };

                raw_spin_unlock_irqrestore(&p->pi_lock, flags);
                stop_one_cpu(task_cpu(p), migration_cpu_stop, &arg);
                return;
        }
unlock:
        raw_spin_unlock_irqrestore(&p->pi_lock, flags);
}

#endif
```

Implementing the detection of CPU configurations requires a little more thinking and code. To be concise, Linux creates the representation of the CPU topology of the system by encapsulating CPUs using data structures called scheduling domains and scheduling groups.

A scheduling domain contains scheduling groups that share same attributes. A scheduling group, in turn, contains a set of CPUs. Load balancing occurs between scheduling groups.

For example, as mentioned, the processor of this computer has 4 hyperthreding cores with a total of 8 threads. Then, there is one domain for each core, and each domain has one scheduling group pointing to each thread. Then, there is one scheduling domain at the chip level with 4 scheduling groups each pointing to two threads of one of the cores.

The below diagram illustrates a simpler case.

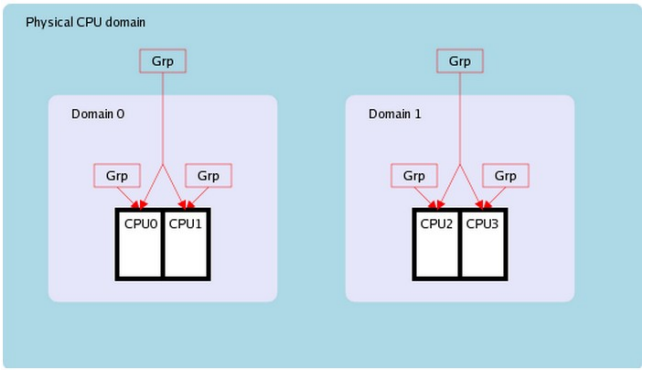

Figure 2. A package with two HT processors [2].

Below is the flags used for scheduling domains:

include/linux/sched.h

```
/*
 * sched-domains (multiprocessor balancing) declarations:
 */
#ifdef CONFIG_SMP
#define SD_LOAD_BALANCE                 0x0001
/* Do load balancing on this domain. */
#define SD_BALANCE_NEWIDLE   0x0002
/* Balance when about to become idle */
#define SD_BALANCE_EXEC                 0x0004
/* Balance on exec */
#define SD_BALANCE_FORK                 0x0008
/* Balance on fork, clone */
#define SD_BALANCE_WAKE                 0x0010
/* Balance on wakeup */
#define SD_WAKE_AFFINE                  0x0020
/* Wake task to waking CPU */
#define SD_SHARE_CPUPOWER    0x0080
/* Domain members share cpu power */
#define SD_SHARE_PKG_RESOURCES          0x0200
/* Domain members share cpu pkg resources */
#define SD_SERIALIZE                0x0400
/* Only a single load balancing instance */
#define SD_ASYM_PACKING                 0x0800
/* Place busy groups earlier in the domain */
#define SD_PREFER_SIBLING       0x1000
/* Prefer to place tasks in a sibling domain */
#define SD_OVERLAP                  0x2000
/* sched_domains of this level overlap */
#define SD_NUMA                         0x4000
/* cross-node balancing */
```

include/linux/topology.h #define SD_SIBLING_INIT

```
.flags          = 1*SD_LOAD_BALANCE                             \
                | 1*SD_BALANCE_NEWIDLE                          \
                | 1*SD_BALANCE_EXEC                             \
                | 1*SD_BALANCE_FORK                             \
                | 0*SD_BALANCE_WAKE                             \
                | 1*SD_WAKE_AFFINE                      \
                | 1*SD_SHARE_CPUPOWER                           \
                | 1*SD_SHARE_PKG_RESOURCES              \
                | 0*SD_SERIALIZE                        \
                | 0*SD_PREFER_SIBLING                           \
                | arch_sd_sibling_asym_packing()        \
```

include/linux/topology.h #define SD_MC_INIT

```
.flags          = 1*SD_LOAD_BALANCE                             \
                | 1*SD_BALANCE_NEWIDLE                          \
                | 1*SD_BALANCE_EXEC                             \
                | 1*SD_BALANCE_FORK                             \
                | 0*SD_BALANCE_WAKE                             \
                | 1*SD_WAKE_AFFINE                      \
                | 0*SD_SHARE_CPUPOWER                           \
                | 1*SD_SHARE_PKG_RESOURCES              \
                | 0*SD_SERIALIZE                        \
```

include/linux/topology.h #define SD_CPU_INIT

```
.flags          = 1*SD_LOAD_BALANCE                             \
                | 1*SD_BALANCE_NEWIDLE                          \
                | 1*SD_BALANCE_EXEC                             \
                | 1*SD_BALANCE_FORK                             \
                | 0*SD_BALANCE_WAKE                             \
                | 1*SD_WAKE_AFFINE                      \
                | 0*SD_SHARE_CPUPOWER                           \
```

```
| 0*SD_SHARE_PKG_RESOURCES        \
| 0*SD_SERIALIZE                  \
| 1*SD_PREFER_SIBLING                    \
```

6.2 Interrupt

An interrupt is a way for a hardware device to tell the kernel that a certain event has happened.

Interrupt requests must be distributed before arriving at a CPU for execution. Evidently, in a multi-processor system, the way interrupts are distributed is more complicated than it is in a uni-processor system.

In a multi-processor system, interrupts must be distributed using methods so as not to overload one of many available CPUs; otherwise, they cam lead to great latency of interrupt processing.

6.3 Power Saving

Increasing performance of a computer system has a cost. More processors or a higher CPU frequency means more power consumption.

Maximizing the performance of a system shouldn't be done on some systems, especially for mobile devices. In this case, the power of a computer must be saved.

This can be achieved by modifying the load balancer to deliberately make idle CPUs or leave idle CPUs idle, even if many tasks are pending. The system can then save power by putting the CPUs to idle state or lowering clock cycles.

Implementing this methods in Linux load balancer must consider the overhead of switching CPU states needs to be considered.

7. SUMMARY/CONCLUSION

To make parallel processing possible, Linux takes advantage of various concepts and implementations that are different from those used for uni-processors.

Synchronization must be done for a computer system to be free from data corruption. Different parts of Linux are implemented with different synchronization techniques to avoid unnecessary performance overhead.

Load balancing is the methods for maximizing the performance of a system, and corresponding parts of Linux are implemented with various performance considerations in mind.

One may encounter a situation that one implementation of Linux doesn't perform as well as expected on a parallel processing system. By understanding Linux at the implementation level, he or she is able to fine tune the code relevant to parallel processing.

8. REFERENCES

[1] Bovet, Daniele, and Marco Cesati. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, 2006.

[2] Covert. *Scheduling domains*. 2004.

http://lwn.net/Articles/80911/

[3] *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*.

[4] Linux Kernel http://www.kernel.org

[5] Love, Robert. *Linux Kernel Development, Third Edition*. Addison-Wesle, 2010.

[6] Mauerer, Wolfgang. Professional Linux Kernel Architecture. Wiley Publishing, Inc., 2008.

[7] McKenney, Paul, Walpole, Jonathan. *What is RCU, Fundamentally?* 2007.

http://lwn.net/Articles/262464/

[8] Torvalds, Linus. *Re: 2.4.7-pre9.. 2001.*

http://lwn.net/2001/0802/a/lt-completions.php3

[9] *Variable SMP – A Multi-CoreCPU Architecture for Low Power and High Performance*. 2011.

http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf.

Columns on Last Page Should Be Made As Close As Possible to Equal Length