Red Hat Enterprise Linux Phase 1

Ben Andrews, Jimmy Hickey, Erika Jensen

October 20, 2017

Contents

1 Introduction **2**
- 1.1 Structure . . . 2
- 1.2 Design Goals . . . 2
- 1.3 Benefits and Limitations . . . 3

2 Kernel Process Synchronization **3**
- 2.1 Normal Critical Sections . . . 3
- 2.2 Interrupt Critical Section . . . 4
- 2.3 PseudoCode . . . 5
  - 2.3.1 Spin Lock . . . 5
  - 2.3.2 Semaphore Locks . . . 5

3 CPU Scheduling **6**
- 3.1 Realtime Scheduling Policies . . . 6
  - 3.1.1 `SCHED_FIFO` . . . 6
  - 3.1.2 `SCHED_RR` . . . 7
- 3.2 Normal Scheduling Policies . . . 7
  - 3.2.1 `SCHED_OTHER` . . . 8
  - 3.2.2 CFS . . . 8

4 Memory Management **8**
- 4.1 Physical Memory . . . 9
  - 4.1.1 Zones of Physical Memory . . . 9
  - 4.1.2 Page Allocator . . . 9
  - 4.1.3 Other Memory Allocations . . . 9
- 4.2 Virtual Memory . . . 10
- 4.3 Swapping of Memory . . . 11

1 Introduction

1.1 Structure

Red Hat is built on the Linux Kernel. The Linux Kernel is layered into subsystems. At a high-level view, it can be divided into three subsystems: the system call interface, architecture-independent kernel code, and the architecture-dependent kernel code. At lower levels it can be divided into even more subsystems: the system call interface, process management, virtual file system, memory management, network stack, arch, and device drivers. However, it can also be viewed as a monolithic structure because it packages all of its most basic services into the kernel. This is not the same as a microkernel structure, where more specific services would be plugged into a microkernel layer.

Red Hat has made changes to the core Linux Kernel over the years. Some of the most recent changes to the kernel (version 3.10) in Red Hat Enterprise Linux 7.X include updates allowing dynamic kernel patching, swap memory compression, NUMA-aware scheduling and memory allocation, a hardware error reporting mechanism, AMD Microcode and AMD Opteron Support, and more. These updates improve the kernel in different ways. For example, Swap Memory Compression is used to reduce I/O. AMD Microcode and AMD Opteron Support increased flexibility and portability. One big change from a previous kernel (version 6.X) includes a tickless kernel. Prior to this update, the kernel would wake up hundreds to thousands of times per second, and query for tasks. The new, tickless kernel is driven by interrupts, meaning the system sleeps until it is asked to wake up and do something. This change increases power savings.

1.2 Design Goals

One of the things that makes Red Hat stand out from other Linux based systems is that it aims to bring Linux into a more enterprise-focused environment. The main design goals

for Red Hat Linux Enterprise are quite simple: Reliability, Availability, Scalability, and Manageability (RASM). These goals are in line with what large companies typically want out of any enterprise-edition software.

1.3 Benefits and Limitations

There are many advantages of using Red Hat Enterprise Linux. One example is that it is compatible with Microsoft products, specifically Windows Server. Red Hat supports many versions of Red Hat Enterprise Linux, allowing for stable deployments. One of the biggest downsides of Red Hat is that it is a commercial product and you have to pay for it, which is not true of all Linux distributions. However, there are Linux distros that use the Red Hat Linux Package Management System (RPM), and are free to use, such as Fedora and CentOS.

2 Kernel Process Synchronization

Red Hat Enterprise Linux is a commercial Linux Distribition. Linux supports threads by treating them as any other process. Linux uses 'task' as a general term to refer to processes and threads. Since version 2.6 the Linux kernel has been preemptive, which means it needs a synchronization mechanism. There are two cases when synchronization is needed, for critical code sections during normal kernel activity and critical sections during interrupt handling.

2.1 Normal Critical Sections

For critical sections during normal kernel activity synchronization, Linux provides short term and long term locking strategies. On SMP systems Linux uses Spin Locks for its short term locks, and for single processor machines it replaces spinlocks with turning

preemption on and off. For locks that need to be held for a longer term, Semaphores are used to lock critical sections of code.

2.2 Interrupt Critical Section

During interrupt handling activity synchronization, interrupts need to be prevented from entering critical sections being used by on going interrupts. The primitive way to achieve this is to take advantage of a processor's interrupt disabling features. However, while interrupts are disabled all I/O is disabled, all devices are left stranded, and performance worsens. To prevent this, Linux uses a special synchronization strategy, by putting interrupt routines into two sections.

These sections are called the top half and the bottom half. Interrupt services in the top half have a number, and all other interrupt services that share that number are disabled. All others stay enabled. The interrupt services that run in the bottom half run with all interrupts enabled. To keep synchronization there is a mini scheduler that runs the bottom half that ensures all the services running in the bottom half do not interrupt each other. Some bottom half services are disabled while foreground kernel code is running, which allows the foreground code to keep its critical sections safe.

2.3 PseudoCode

2.3.1 Spin Lock

```
// derived from /include/linux/spinlock.h
void spinlock(lock_t *lock)
{
        for (;;) {
                preempt_disable();
                if (can_aquire(lock))
                    break_out_of_for_loop;

                preempt_enable();

                if (!lock->break_lock)
                    lock->break_lock = 1;

                while (can_get_lock(lock) && lock->break_lock)
                    ;
        }
        lock->break_lock = 0;
}
```

2.3.2 Semaphore Locks

```
// derived from /include/linux/semaphore.h
void sema_init(struct semaphore *sem, int val){
   struct lock_class_key key;
   *sem = initialize_semaphore(*sem, val);
}

// derived from kernel/locking/semaphore.c
void down(struct semaphore *sem){
        spinlock(&sem->lock);

        if (sem->count > 0)
            sem->count--;
        else
            down(sem); // wait
}

// derived from /include/linux/semaphore.h
void up(struct semaphore *sem){
    if (list_empty(&sem->wait_list))
        sem->count++;
    else
        up(sem); // let the first waiting task exit its wait
}
```

3 CPU Scheduling

Red Hat combats schedule processing by using two distinct policies to allocate processes/ threads to particular cores. These are the realtime and normal policies.

3.1 Realtime Scheduling Policies

Realtime processes are always scheduled to run before normal processes. This policy is used for time sensitive, atomic tasks. With their increased priority, these processes should not run for long to avoid monopolizing CPU time. Realtime policies are implemented with using a FIFO algorithm (`SCHED_FIFO`) and a round robin algorithm (`SCHED_RR`).

Both of these policies assign priority to each process in the ready queue. Priorities range from 1 and 99, with higher numbers representing a more important job. It is advised to start with lower priorities, only increasing with reason. Setting priorities too high can result in a myriad of issues. Setting a thread's priority to 99 will set it at the same level as watchdog and migration processes. Since the realtime threads are blocking, these important system processes will not be able to preempt the user processes and will not run. This will send the processor into a computational loop and can lock uniprocessor systems.

3.1.1 `SCHED_FIFO`

This policy also includes a bandwidth cap to prevent one thread taking an unfair amount of resources. This cap can be adjusted by the user.

- `/proc/sys/kernel/sched_rt_period_us` controls the time period that will be considered 100% of the CPU bandwidth. By default this is $1\,000\,000\,\mu s$ or $1\,s$.
- `/proc/sys/kernel/sched_rt_runtime_us` controls how the time period that will be given to the realtime processes. By default this is $9\,500\,000\,\mu s$ or $0.95\,s$.

Through these parameters, the monopolization problem is addressed. The `SCHED_FIFO` algorithm is simple. It scans the ready queue for the process with the highest priority and then runs that process.

```
while True{
        process_to_run = 0
        process_to_run_priority = queue[0].priority

        for (i in 0 to queue.length) {
                if (queue[i].priority > process_to_run_priority){
                        process_to_run = i
                        process_to_run_priority = queue[i].priority
                }
        }
        run queue[process_to_run]
}
```

This algorithm is useful for optimizing the response time.

3.1.2 SCHED_RR

This is a round robin variant of the previously described `SCHED_FIFO` algorithm. The user can view the time quantum using the `sched_rr_get_interval(2)` system call, but this cannot be editted. The `SCHED_RR` algorithm is useful when there are multiple processes of the same priority in the ready queue.

3.2 Normal Scheduling Policies

Normal policies are used for lower priority jobs that have limited interest in performance tuning. These policies have better throughput because they do not need to reschedule for preemption as often as realtime policies. This scheduling is implemented through three algorithms: `SCHED_OTHER`, `SCHED_BATCH`, and `SCHED_IDLE`.

3.2.1 `SCHED_OTHER`

This algorithm uses the Completely Fair Scheduler (CFS) to provide access to all processes. Its priority can be indirectly controlled by the user, but its true dynamic priority is determined by the "niceness" of the process is only directly changeable by the CFS.

3.2.2 CFS

The primary goal of the CFS is to provide fair use of the CPU to each process. The less time that a task has been permitted to access the processor, the more its need is to access the processor. It also ensures that waiting tasks (such as tasks waiting for I/O) are allotted their processor time when they need it. This is the concept of sleeper fairness.

Instead of using a queue to organize the ready processes, the CFS uses a time-ordered red-black tree. A red-black tree offers many desired characteristics. First, it self balances, so no path will be twice as long as any other (or the shortest path). Also, tree traversal is quick running with $O \log(n)$ efficiency. This allows for quick insertion and deletion.

Tasks with the highest need to run are stored on the left side while those with the least need sorted to the right. The scheduler selects the left-most node. It is then reinserted into the tree.

4 Memory Management

The Linux Kernel separates memory management into two parts. One part deals with physical memory, being pages, groups of pages, and blocks of RAM. The other manages virtual memory.

4.1 Physical Memory

4.1.1 Zones of Physical Memory

Linux separates physical memory into four zones, `ZONE_DMA`, `ZONE_DMA32`, `ZONE_NORMAL`, `ZONE_HIGHMEM`. The functionality of the zones is dependent on the architecture. The `ZONE_DMA` and the `ZONE_DMA32` are designated for devices that can only access the lower addressed bytes of memory. `ZONE_HIGHMEM` is memory that is not mapped to the kernel address space, and `ZONE_NORMAL` is everything else. `ZONE_NORMAL` consists of regular mapped pages of memory. The kernel keeps a list of free pages for each zone. When physical memory is requested the kernel gives it pages in the appropriate zone.

4.1.2 Page Allocator

Linux's physical memory is primarily managed by its page allocators. Each of the four zones has its own allocator which handles allocating and freeing the physical pages of memory. They are also able to allocate physically contiguous pages of memory if required. Free pages of memory are tracked of using a buddy system. If two contiguous units are both free, they are combined to create a larger piece of memory, creating a buddy heap. This buddy heap also has a buddy; if its buddy is free they are combined to create a larger buddy heap. If a request for a small amount of memory comes in, these heaps can be divided into their respective smaller units, recursively, until a unit small enough is created. A linked list keeps track of units of each size. One physical page of memory is the smallest available size. If a request comes in that is smaller than a page, Linux uses a specialized memory allocator.

4.1.3 Other Memory Allocations

However, not all memory is allocated by the memory allocator. Some memory is statically allocated by drivers during bootup. Further, some memory is allocated using slab alloca-

tion. Slabs are used for memory holding the kernel's data structures, and are made up of contiguous pages of memory. These slabs are organized into caches, which may hold one or more memory slabs. A cache holds a single type of kernel data structure. Caches are organized into objects, single instantiations of a kernel data structure. Objects that are not being used are considered free; the allocator is allowed to populate them in the cache. Depending on the freeness state of the objects in the slab, the slab can be:

- Full - all objects are marked as not free.
- Partial - some, but not all, of the objects are marked as free.
- Empty - all the objects in the slab are marked as free.

When the slab allocator receives a request it first tries to allocate the object into a partially free slab. If there are no partially free slabs the allocator will assign the object to an empty slab. If there are no appropriate empty slabs a new slab will be allocated from contiguous pages.

4.2 Virtual Memory

In the Linux Kernel the virtual memory system keeps track of each process's address space. Pages of virtual memory are created. Then the virtual memory manager swaps the pages to and from disk as needed. New virtual address spaces are created under two instances. The first is when `exec()` is called. In this case the process is given a new empty virtual address space. The second is when `fork()` is called. This requires copying a process's complete virtual address space. There are two views associated with virtual memory: the logical view and the physical view. The logical view describes the layout of the address space. In this view the memory is made up of a contiguous block of memory. The physical view is stored in the page tables of the hardware. These tables keep track of where a process's logical pages are located in physical memory.

4.3 Swapping of Memory

Swapping of memory is needed to use memory efficiently. Linux implements page swapping instead of whole process swapping. To decide which page to swap out of memory, Linux uses its pageout policy which uses a multipass clock. Every time the clock ticks all pages of memory are viewed and their age is updated based on whether or not they have been used. The pageout policy swaps out the least frequently used pages. Linux uses its paging mechanism that can swap pages out to files or dedicated swap partitions. Swapping to swap partitions is preferred; swapping to files is less efficient because of the overhead incurred by the file system.

Sources

1. IBM developerWorks
2. Linux Kernel
3. Network World
4. Operating Systems Concepts
5. Red Hat Website