GNU/Linux Kernel Scheduling Design and Alternatives

Kernel Design and Schedule Models

Ruben Safir 2015

Abstract:

The default scheduler for the 3.0 trunk of the GNU/Linux operating system is called the Completely Fair Operating System. Instead of a confusing and complex set of heuristics to try to keep a multiple of system types balanced and in order to avoid deadlocks, feuding scheduler project managers have evolved a largely elegant design that can be outlined and understood within 10 pages text.[1] *In the grand scheme of things, that is quite an accomplishment. In truth, the scheduler can be outlined in even simpler terms then that. A lot of confusion has developed in discussing the CFS because of the history of schedulers, but the product itself is extremely simple to abstract. All the running-ready tasks are listed on a RB-Binary tree and process that has seen the least amount of CPU active time gets plucked for CPU run time. After running it gets rotated back to the tree and placed inside the tree in order of its cumulative run time statistics.*[1] *The simplicity of this design has opened the Linux Scheduler up for advanced development. And today, there are schedulers and research into many avenues of development from Real Time Operating Systems to handhold devices.*

This paper will explore the code foundation for the Linux CFS and look at related software architectural support and then investigate an test the commonly substituted BFS as an exercise to show how easily one scheduler can be alternated for another within the Kernel code base. Outline will be methods, resources and utilities for kernel developers to code their own schedulers.

1. Inside the Linux 2.6 Completely Fair Scheduler: Jones, Tim December 2009. IMB DeveopmetWorks

Basic Linux Kernel Scheduling System Structure

The default Gnu/Linux scheduler within the current 3.0 tree of the kernel development is called the Completely Fair Scheduler. It is a Red and Black BTREE where the left most task is plucked off the tree and put in a run time state, handed to a CPU for running, and when interupted, accounted for the time that was spent on the CPU and if needed, put back on the red and Black Btree for later scheduling. The order of the RB BTREE is dependent on a variable called **u64 vruntime** in the **struct sched_entity**. [1] While this description is really this simple, direct and correct, like a good novel, scheduling can be explored and understood as an example of operating system design at a variety of intellectual levels depending on what one brings to the story. When examining the code base for the kernel, one can see how the scheduler in embedded in an environment of wait queues, system, run queues, standardized kernel specific software containers, task signaling, and preemption theory, unix process control, and scheduler theory.

Scheduling goes right at the heart of operating systems. Since the Gnu/Linux scheduler is both code available, well studied and relatively easy to understand, the best part of studying it is that it intersects so many other part of the operating system. This makes the study of the Scheduler an ideal place for the advanced undergraduate Computer Science University student to begin their studies of more difficult topics of operating system design and kernel coding. It gives students a practical demonstration of various algorithms that would be otherwise learned only in theory, and one can see the real world affects of these algorithms at the level of ones keyboard and mouse. The code base also demonstrates the advanced use of signals and waiting, which one would be hard pressed to find understandable code to evaluate otherwise, and to see advanced structs in action. It transforms the student from observer to participant in multiple areas of theoretical knowledge and puts them into one of the great collaborative efforts in the history of mankind.

Before looking at the details of the code of the schedule and how we can manipulate it, lets first look at the overall structure of the scheduler design and its relationship with tasks. In Linux, threads and processes are differentiated tasks. They are, on the kernel level, fundamentally the same.[2] Each is described in the Kernel with struct task_struct. Instances of task_struct's are strong along in linked lists in runtime queues. They can be running or or sleeping, and there are different kernel queues for each. task_struct is defined in located linux-3.19.3/include/linux/sched.h defined from lines from lines 1274 to 1704. Since tasks_struct is so core to the system it is defined with quite a few processor directives within it and is work looking at as a coding lesson. Application programmers will rarely run into structure declarations nearly 500 lines long and with preprocessor commands.

---

1 Love, Robert: “Linux Kernel Development 3rd Edition” Page pg 50
2 Love, Robert: “Linux Kernel Development 3rd Edition” Pg 35

Struct task_struct in sched.h

```
struct task_struct {
        volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;   /* per process flags, defined below */
        unsigned int ptrace;

#ifdef CONFIG_SMP
        struct llist_node wake_entry;
        int on_cpu;
        struct task_struct *last_wakee;
        unsigned long wakee_flips;
        unsigned long wakee_flip_decay_ts;

        int wake_cpu;
#endif
        int on_rq;

        int prio, static_prio, normal_prio;
        unsigned int rt_priority;
        const struct sched_class *sched_class;
        struct sched_entity se;
        struct sched_rt_entity rt;
#ifdef CONFIG_CGROUP_SCHED
        struct task_group *sched_task_group;
#endif
        struct sched_dl_entity dl;

#ifdef CONFIG_PREEMPT_NOTIFIERS
        /* list of struct preempt_notifier: */
        struct hlist_head preempt_notifiers;
#endif

#ifdef CONFIG_BLK_DEV_IO_TRACE
        unsigned int btrace_seq;
#endif

        unsigned int policy;
        int nr_cpus_allowed;
        cpumask_t cpus_allowed;

#ifdef CONFIG_PREEMPT_RCU
        int rcu_read_lock_nesting;
        union rcu_special rcu_read_unlock_special;
        struct list_head rcu_node_entry;
#endif /* #ifdef CONFIG_PREEMPT_RCU */
#ifdef CONFIG_PREEMPT_RCU
        struct rcu_node *rcu_blocked_node;
#endif /* #ifdef CONFIG_PREEMPT_RCU */
#ifdef CONFIG_TASKS_RCU
        unsigned long rcu_tasks_nvcsw;
        bool rcu_tasks_holdout;
        struct list_head rcu_tasks_holdout_list;
        int rcu_tasks_idle_cpu;
#endif /* #ifdef CONFIG_TASKS_RCU */

#if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
        struct sched_info sched_info;
#endif
```

```
        struct list_head tasks;
#ifdef CONFIG_SMP
        struct plist_node pushable_tasks;
        struct rb_node pushable_dl_tasks;
#endif

        struct mm_struct *mm, *active_mm;
#ifdef CONFIG_COMPAT_BRK
        unsigned brk_randomized:1;
#endif
        /* per-thread vma caching */
        u32 vmacache_seqnum;
        struct vm_area_struct *vmacache[VMACACHE_SIZE];
#if defined(SPLIT_RSS_COUNTING)
        struct task_rss_stat  rss_stat;
#endif
/* task state */
        int exit_state;
        int exit_code, exit_signal;
        int pdeath_signal;  /*  The signal sent when the parent dies  */
        unsigned int jobctl;  /* JOBCTL_*, siglock protected */

        /* Used for emulating ABI behavior of previous Linux versions */
        unsigned int personality;

        unsigned in_execve:1; /* Tell the LSMs that the process is doing an
                               * execve */
        unsigned in_iowait:1;

        /* Revert to default priority/policy when forking */
        unsigned sched_reset_on_fork:1;
        unsigned sched_contributes_to_load:1;

#ifdef CONFIG_MEMCG_KMEM
        unsigned memcg_kmem_skip_account:1;
#endif

        unsigned long atomic_flags; /* Flags needing atomic access. */

        pid_t pid;
        pid_t tgid;

#ifdef CONFIG_CC_STACKPROTECTOR
        /* Canary value for the -fstack-protector gcc feature */
        unsigned long stack_canary;
#endif
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->real_parent->pid)
         */
        struct task_struct __rcu *real_parent; /* real parent process */
        struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() reports */
        /*
         * children/sibling forms the list of my natural children
         */
        struct list_head children;    /* list of my children */
        struct list_head sibling;     /* linkage in my parent's children list */
        struct task_struct *group_leader;    /* threadgroup leader */

        /*
```

```
         * ptraced is the list of tasks this task is using ptrace on.
         * This includes both natural children and PTRACE_ATTACH targets.
         * p->ptrace_entry is p's link on the p->parent->ptraced list.
         */
        struct list_head ptraced;
        struct list_head ptrace_entry;

        /* PID/PID hash table linkage. */
        struct pid_link pids[PIDTYPE_MAX];
        struct list_head thread_group;
        struct list_head thread_node;

        struct completion *vfork_done;              /* for vfork() */
        int __user *set_child_tid;          /* CLONE_CHILD_SETTID */
        int __user *clear_child_tid;        /* CLONE_CHILD_CLEARTID */

        cputime_t utime, stime, utimescaled, stimescaled;
        cputime_t gtime;
#ifndef CONFIG_VIRT_CPU_ACCOUNTING_NATIVE
        struct cputime prev_cputime;
#endif
#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
        seqlock_t vtime_seqlock;
        unsigned long long vtime_snap;
        enum {
                VTIME_SLEEPING = 0,
                VTIME_USER,
                VTIME_SYS,
        } vtime_snap_whence;
#endif
        unsigned long nvcsw, nivcsw; /* context switch counts */
        u64 start_time;              /* monotonic time in nsec */
        u64 real_start_time;  /* boot based time in nsec */
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
        unsigned long min_flt, maj_flt;

        struct task_cputime cputime_expires;
        struct list_head cpu_timers[3];

/* process credentials */
        const struct cred __rcu *real_cred; /* objective and real subjective task
                                     * credentials (COW) */
        const struct cred __rcu *cred;       /* effective (overridable) subjective task
                                     * credentials (COW) */
        char comm[TASK_COMM_LEN]; /* executable name excluding path
                                    - access with [gs]et_task_comm (which lock
                                      it with task_lock())
                                    - initialized normally by setup_new_exec */
/* file system info */
        int link_count, total_link_count;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
        struct sysv_sem sysvsem;
        struct sysv_shm sysvshm;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
/* hung task detection */
        unsigned long last_switch_count;
#endif
/* CPU-specific state of this task */
        struct thread_struct thread;
/* filesystem information */
```

```
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
/* namespaces */
        struct nsproxy *nsproxy;
/* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;

        sigset_t blocked, real_blocked;
        sigset_t saved_sigmask;       /* restored if set_restore_sigmask() was used */
        struct sigpending pending;

        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;
        struct callback_head *task_works;

        struct audit_context *audit_context;
#ifdef CONFIG_AUDITSYSCALL
        kuid_t loginuid;
        unsigned int sessionid;
#endif
        struct seccomp seccomp;

/* Thread group tracking */
        u32 parent_exec_id;
        u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings, mems_allowed,
 * mempolicy */
        spinlock_t alloc_lock;

        /* Protection of the PI data structures: */
        raw_spinlock_t pi_lock;

#ifdef CONFIG_RT_MUTEXES
        /* PI waiters blocked on a rt_mutex held by this task */
        struct rb_root pi_waiters;
        struct rb_node *pi_waiters_leftmost;
        /* Deadlock detection and priority inheritance handling */
        struct rt_mutex_waiter *pi_blocked_on;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
        /* mutex deadlock detection */
        struct mutex_waiter *blocked_on;
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
        unsigned int irq_events;
        unsigned long hardirq_enable_ip;
        unsigned long hardirq_disable_ip;
        unsigned int hardirq_enable_event;
        unsigned int hardirq_disable_event;
        int hardirqs_enabled;
        int hardirq_context;
        unsigned long softirq_disable_ip;
        unsigned long softirq_enable_ip;
        unsigned int softirq_disable_event;
        unsigned int softirq_enable_event;
        int softirqs_enabled;
```

```
        int softirq_context;
#endif
#ifdef CONFIG_LOCKDEP
# define MAX_LOCK_DEPTH 48UL
        u64 curr_chain_key;
        int lockdep_depth;
        unsigned int lockdep_recursion;
        struct held_lock held_locks[MAX_LOCK_DEPTH];
        gfp_t lockdep_reclaim_gfp;
#endif

/* journalling filesystem info */
        void *journal_info;

/* stacked block device info */
        struct bio_list *bio_list;

#ifdef CONFIG_BLOCK
/* stack plugging */
        struct blk_plug *plug;
#endif

/* VM state */
        struct reclaim_state *reclaim_state;

        struct backing_dev_info *backing_dev_info;

        struct io_context *io_context;

        unsigned long ptrace_message;
        siginfo_t *last_siginfo; /* For ptrace use.  */
        struct task_io_accounting ioac;
#if defined(CONFIG_TASK_XACCT)
        u64 acct_rss_mem1;     /* accumulated rss usage */
        u64 acct_vm_mem1;      /* accumulated virtual memory usage */
        cputime_t acct_timexpd;       /* stime + utime since last update */
#endif
#ifdef CONFIG_CPUSETS
        nodemask_t mems_allowed;      /* Protected by alloc_lock */
        seqcount_t mems_allowed_seq;  /* Seqence no to catch updates */
        int cpuset_mem_spread_rotor;
        int cpuset_slab_spread_rotor;
#endif
#ifdef CONFIG_CGROUPS
        /* Control Group info protected by css_set_lock */
        struct css_set __rcu *cgroups;
        /* cg_list protected by css_set_lock and tsk->alloc_lock */
        struct list_head cg_list;
#endif
#ifdef CONFIG_FUTEX
        struct robust_list_head __user *robust_list;
#ifdef CONFIG_COMPAT
        struct compat_robust_list_head __user *compat_robust_list;
#endif
        struct list_head pi_state_list;
        struct futex_pi_state *pi_state_cache;
#endif
#ifdef CONFIG_PERF_EVENTS
        struct perf_event_context *perf_event_ctxp[perf_nr_task_contexts];
        struct mutex perf_event_mutex;
        struct list_head perf_event_list;
#endif
```

```
#ifdef CONFIG_DEBUG_PREEMPT
        unsigned long preempt_disable_ip;
#endif
#ifdef CONFIG_NUMA
        struct mempolicy *mempolicy;  /* Protected by alloc_lock */
        short il_next;
        short pref_node_fork;
#endif
#ifdef CONFIG_NUMA_BALANCING
        int numa_scan_seq;
        unsigned int numa_scan_period;
        unsigned int numa_scan_period_max;
        int numa_preferred_nid;
        unsigned long numa_migrate_retry;
        u64 node_stamp;                      /* migration stamp  */
        u64 last_task_numa_placement;
        u64 last_sum_exec_runtime;
        struct callback_head numa_work;

        struct list_head numa_entry;
        struct numa_group *numa_group;

        /*
         * numa_faults is an array split into four regions:
         * faults_memory, faults_cpu, faults_memory_buffer, faults_cpu_buffer
         * in this precise order.
         *
         * faults_memory: Exponential decaying average of faults on a per-node
         * basis. Scheduling placement decisions are made based on these
         * counts. The values remain static for the duration of a PTE scan.
         * faults_cpu: Track the nodes the process was running on when a NUMA
         * hinting fault was incurred.
         * faults_memory_buffer and faults_cpu_buffer: Record faults per node
         * during the current scan window. When the scan completes, the counts
         * in faults_memory and faults_cpu decay and these values are copied.
         */
        unsigned long *numa_faults;
        unsigned long total_numa_faults;

        /*
         * numa_faults_locality tracks if faults recorded during the last
         * scan window were remote/local. The task scan period is adapted
         * based on the locality of the faults with different weights
         * depending on whether they were shared or private faults
         */
        unsigned long numa_faults_locality[2];

        unsigned long numa_pages_migrated;
#endif /* CONFIG_NUMA_BALANCING */

        struct rcu_head rcu;

        /*
         * cache last used pipe for splice
         */
        struct pipe_inode_info *splice_pipe;

        struct page_frag task_frag;

#ifdef CONFIG_TASK_DELAY_ACCT
        struct task_delay_info *delays;
#endif
```

```
#ifdef CONFIG_FAULT_INJECTION
	int make_it_fail;
#endif
	/*
	 * when (nr_dirtied >= nr_dirtied_pause), it's time to call
	 * balance_dirty_pages() for some dirty throttling pause
	 */
	int nr_dirtied;
	int nr_dirtied_pause;
	unsigned long dirty_paused_when; /* start of a write-and-pause period */

#ifdef CONFIG_LATENCYTOP
	int latency_record_count;
	struct latency_record latency_record[LT_SAVECOUNT];
#endif
	/*
	 * time slack values; these are used to round up poll() and
	 * select() etc timeout values. These are in nanoseconds.
	 */
	unsigned long timer_slack_ns;
	unsigned long default_timer_slack_ns;

#ifdef CONFIG_FUNCTION_GRAPH_TRACER
	/* Index of current stored address in ret_stack */
	int curr_ret_stack;
	/* Stack of return addresses for return function tracing */
	struct ftrace_ret_stack	*ret_stack;
	/* time stamp for last schedule */
	unsigned long long ftrace_timestamp;
	/*
	 * Number of functions that haven't been traced
	 * because of depth overrun.
	 */
	atomic_t trace_overrun;
	/* Pause for the tracing */
	atomic_t tracing_graph_pause;
#endif
#ifdef CONFIG_TRACING
	/* state flags for use by tracers */
	unsigned long trace;
	/* bitmask and counter of trace recursion */
	unsigned long trace_recursion;
#endif /* CONFIG_TRACING */
#ifdef CONFIG_MEMCG
	struct memcg_oom_info {
		struct mem_cgroup *memcg;
		gfp_t gfp_mask;
		int order;
		unsigned int may_oom:1;
	} memcg_oom;
#endif
#ifdef CONFIG_UPROBES
	struct uprobe_task *utask;
#endif
#if defined(CONFIG_BCACHE) || defined(CONFIG_BCACHE_MODULE)
	unsigned int	sequential_io;
	unsigned int	sequential_io_avg;
#endif
#ifdef CONFIG_DEBUG_ATOMIC_SLEEP
	unsigned long	task_state_change;
#endif
};
```

There are a few things worth noting in this code. First, most of the task_struct consists of optionally included data field surrounded by the #ifdef CONFIG.... #endif commands. These are defined and generated in the kernel compilation process. When compiling the GNU/Linux Kernel, the first step is to produce this kernel configuation file, generally running the command

ruben@host:$ make menuconfig

or some version, the object of which is to produce a working configuration file for the kernel compilation. Details for this exist in the Kernel source code Readme file which currently says:

```
   To configure and build the kernel, use:

     cd /usr/src/linux-3.X
     make O=/home/name/build/kernel menuconfig
     make O=/home/name/build/kernel
     sudo make O=/home/name/build/kernel modules_install install

   Please note: If the 'O=output/dir' option is used, then it must be
   used for all invocations of make.

CONFIGURING the kernel:

   Do not skip this step even if you are only upgrading one minor
   version.  New configuration options are added in each release, and
   odd problems will turn up if the configuration files are not set up
   as expected.  If you want to carry your existing configuration to a
   new version with minimal work, use "make oldconfig", which will
   only ask you for the answers to new questions.

 - Alternative configuration commands are:

     "make config"      Plain text interface.

     "make menuconfig"  Text based color menus, radiolists & dialogs.

     "make nconfig"     Enhanced text based color menus.

     "make xconfig"     X windows (Qt) based configuration tool.

     "make gconfig"     X windows (Gtk) based configuration tool.

     "make oldconfig"   Default all questions based on the contents of
                        your existing ./.config file and asking about
                        new config symbols.

     "make silentoldconfig"
                        Like above, but avoids cluttering the screen
                        with questions already answered.
                        Additionally updates the dependencies.
```

For the general purposes of the scheduler, the most important data point in task_struct is the entry struct **sched_entity se,** and **const struct sched_class *sched_class.** struct sched_entity is a struct defined in linux/struct.h

```
struct sched_entity {
        struct load_weight    load;          /* for load-balancing */
        struct rb_node        run_node;
        struct list_head      group_node;
        unsigned int          on_rq;

        u64                   exec_start;
        u64                   sum_exec_runtime;
        u64                   vruntime;
        u64                   prev_sum_exec_runtime;

        u64                   nr_migrations;

#ifdef CONFIG_SCHEDSTATS
        struct sched_statistics statistics;
#endif

#ifdef CONFIG_FAIR_GROUP_SCHED
        int                   depth;
        struct sched_entity   *parent;
        /* rq on which this entity is (to be) queued: */
        struct cfs_rq         *cfs_rq;
        /* rq "owned" by this entity/group: */
        struct cfs_rq         *my_q;
#endif

#ifdef CONFIG_SMP
        /* Per-entity load-tracking */
        struct sched_avg      avg;
#endif
};
```

One of the ingenious aspects designs of the new schedulers is that the task_struct instances are not themselves juggled around for schedule programming. Instead, this rather svelte structure is used to manipulate task movements through the Red Black Tree that controls the task scheduling.

The data point within sched_entity which directly represents the node position is **struct rb_node run_node** . rb_node is defined in rb_tree.h which declares the functions and data structures needed to control the Red and Black Tree that the CFS uses. The structure itself looks as follows:

```
struct rb_node {
   unsigned long  __rb_parent_color;
   struct rb_node *rb_right;
   struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
    /* The alignment might seem pointless, but allegedly CRIS needs it */

struct rb_root {
   struct rb_node *rb_node;
};
```

Overall, the data structures that control task scheduling and run queues are all interconnected in a hierarchical scheme. To truly understand who this system works, these broad relationships need to be understood and it helps to have a functional diagram of these relations such as the figure 1.

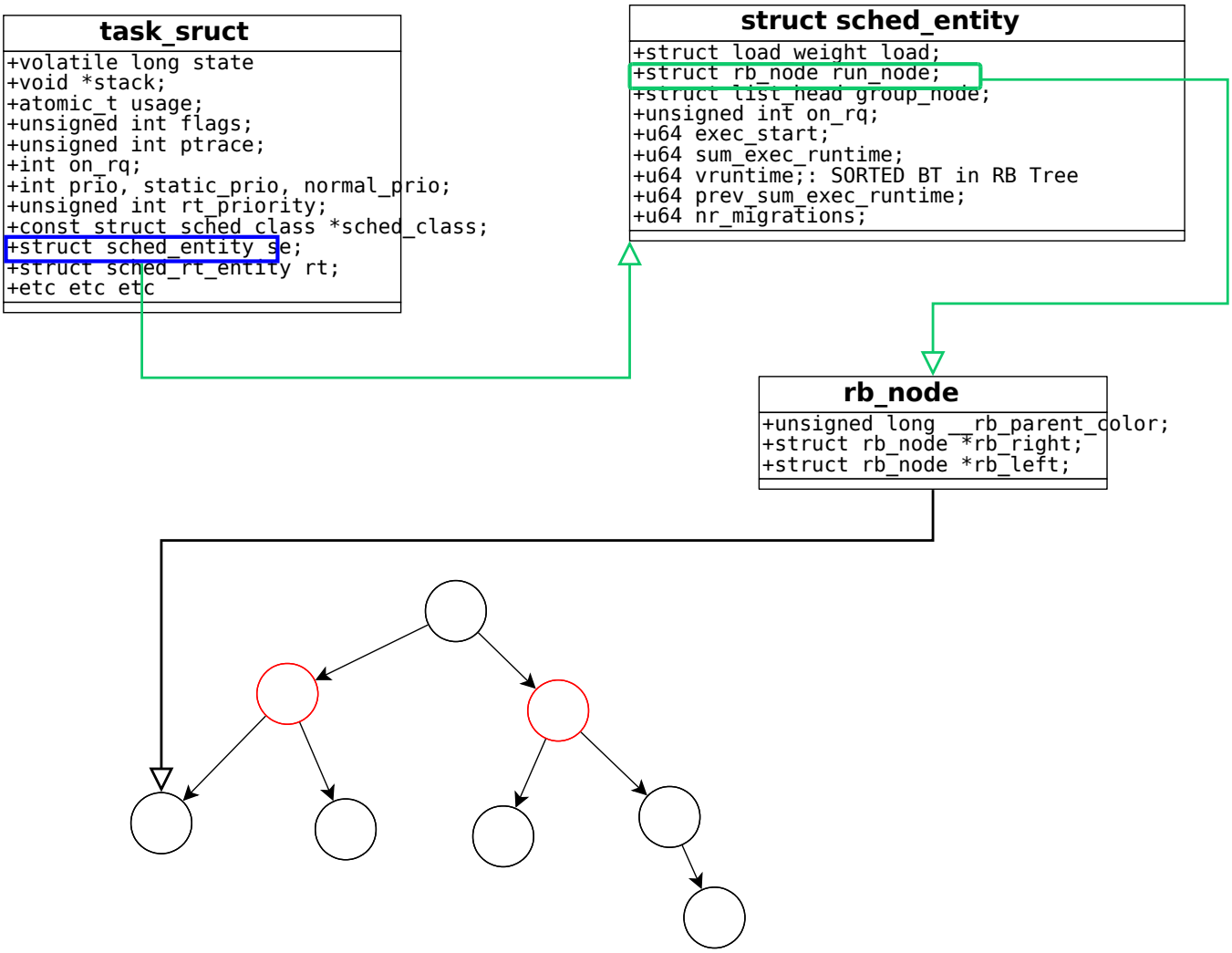


Figure 1

Problematically this means that the following is valid code to access the left child node of a any task within the CFS RB BTREE

```
struct rb_node my_rb_node;

struct task_struct mytask;

my_rb_node  = mytask->se->run_node->left;
```

The nodes themselves are sorted on:

```
mytask->se->vruntime
```

We will review how these relationships are managed in code but before this, let's review some of the basics of a RB BTREE and then how it is specifically implemented in the Gnu/Linux Kernel.

Red and Black trees have a specific kernel and GNU/Linux library located under

linux-3.xx.x/include/linux/rbtree.h and also documented in the under

linux-3.xx.x/Doumentation/rbtree.txt . A Red and Black tree is a semi-balanced binary tree. The classic explanation for Red Black Trees is the series of lectures given by Erik Demaine of MIT whose opencourseware videos explains the properties far better than any other resource I'd yet discovered. His main lector on Red Black trees is at http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-10-red-black-trees-rotations-insertions-deletions/ , and his more advanced lecture on the topic is addressed at https://courses.csail.mit.edu/6.851/spring12/lectures/L05.html . The second video is a very advanced look at how binary trees fit in with a large swath of computer science and mathematical theories and I would recommend reviewing both in one ever wants to understand the arguments and discussions around and about optimization of searches, storage and data access time.

The principles behind a Red Black tree are based on the 2-3-4 multikey tree which can best be described diagrammatically. A 2-3-4 multikey tree (also known as an order 4 tree) is a tree that can have a maximum of 3 indexes per node and therefor a maximum of 4 child nodes. Additionally, we constrain the tree such that tree must be balances, and all the lowest nodes are on the same level. We insert new values from the bottom and push up the root as needed, to keep the lowest level balanced. 2-3-4 trees are isomers of Red And Black Trees as show in the diagram below in figure 2.[3]

Red Black trees have a number of rules and advantages:

1. The Root Node is Black
2. Red Nodes are the 'wings' of the 2-3-4 multikey arraignment
3. Every Red Node has two black child nodes.
4. Leaf Nodes are black and nil
5. You can not have 2 Red Nodes in a row on the same path
6. The number of black nodes in any path is equal to any other path from the lowest level to the root
7. Insertions, for our purposes go from bottom to top and new entries always enter the bottom.
8. Insertions are of the color Red.
9. If an insertion is to replace a black null child whose parent is red, what happens next can

3 Drozdek, Adam: Data Structures and Alloirthms in C++ 4th Edition 2012 pg 309-390

depend on the algorithm used. Either the parents color is flipped or one of a series of node rotations takes place.

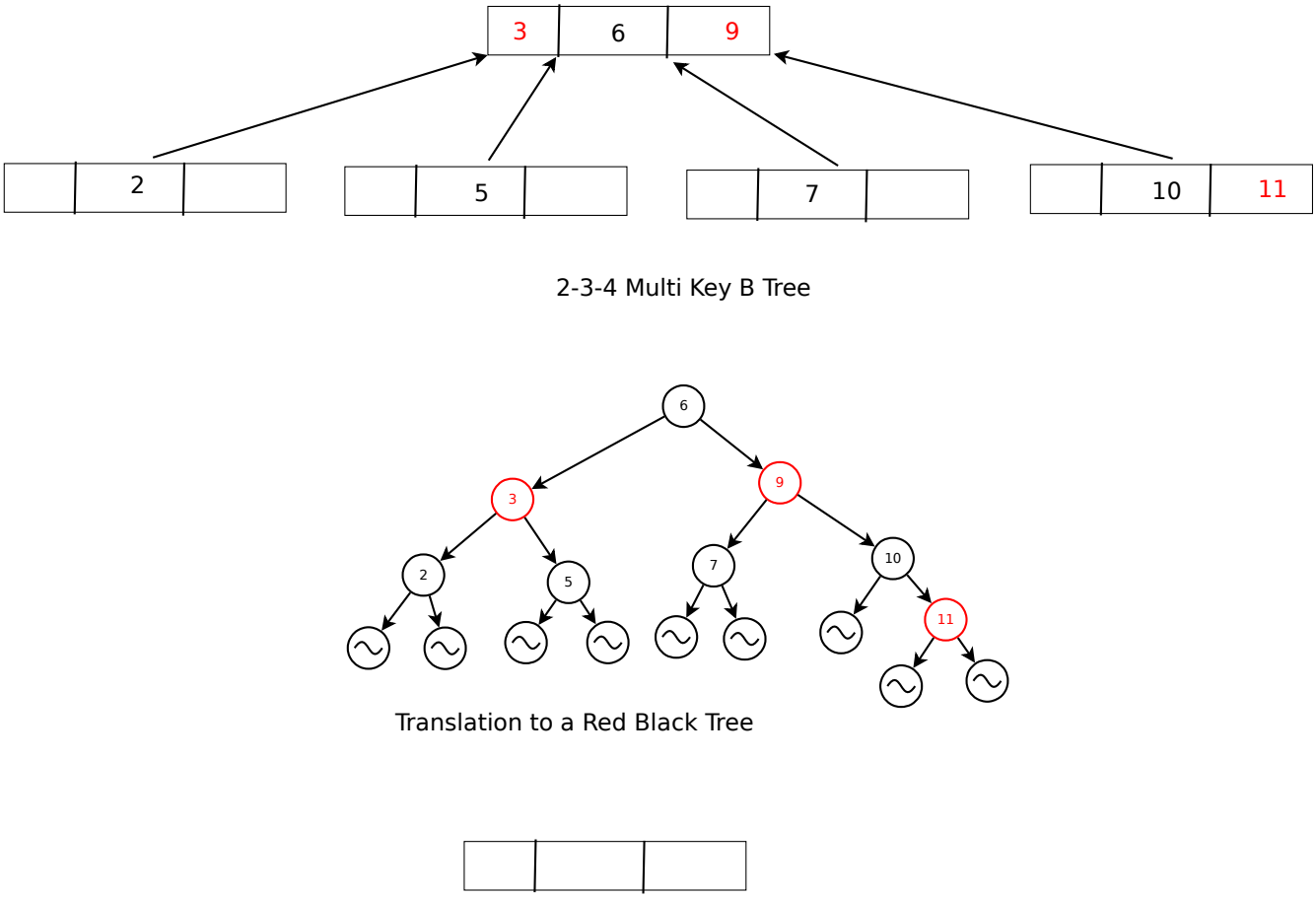


Figure 2: Red Black Tree

Rotation of nodes is needed when the path which is affected by the addition is longer one more in length than sister paths, or when two red nodes come together. We have the specific algorithms for insertion behaviors in the RB Tree. Adding a process to the tree involves calling **static void enqueue_entity()** which calls __enqueue_entity() which in turn calls rb_insert_color and finally _rb_insert() which does the rotations and the insertions. rb_insert() is in linux.x.xx/include/rbtree.c . By examining this code, we can determine exactly the algorithm that the CFS uses to determine the order of which which processes to run.[4]

Below is the code for the rbtree insertion. What is interesting here, aside from seeing a rbtree insertion being used in the wild, is the careful internal documentation that the coders left behind within the code.

4 See Love, Rober: Chapter 4

In order to make sure they didn't lose one of the necessary conditional recipes that an rbtree would use, instead the documented all the possibilities right in the code.

```
 __rb_insert(struct rb_node *node, struct rb_root *root,
           void (*augment_rotate)(struct rb_node *old, struct rb_node *new))
{
        struct rb_node *parent = rb_red_parent(node), *gparent, *tmp;

        while (true) {
                /*
                 * Loop invariant: node is red
                 *
                 * If there is a black parent, we are done.
                 * Otherwise, take some corrective action as we don't
                 * want a red root or two consecutive red nodes.
                 */
                if (!parent) {
                        rb_set_parent_color(node, NULL, RB_BLACK);
                        break;
                } else if (rb_is_black(parent))
                        break;

                gparent = rb_red_parent(parent);

                tmp = gparent->rb_right;
                if (parent != tmp) {  /* parent == gparent->rb_left */
                        if (tmp && rb_is_red(tmp)) {
                                /*
                                 * Case 1 - color flips
                                 *
                                 *       G            g
                                 *      / \          / \
                                 *     p   u  -->   P   U
                                 *    /            /
                                 *   n            n
                                 *
                                 * However, since g's parent might be red, and
                                 * 4) does not allow this, we need to recurse
                                 * at g.
                                 */
                                rb_set_parent_color(tmp, gparent, RB_BLACK);
                                rb_set_parent_color(parent, gparent, RB_BLACK);
                                node = gparent;
                                parent = rb_parent(node);
                                rb_set_parent_color(node, parent, RB_RED);
                                continue;
                        }
```

```
        tmp = parent->rb_right;
        if (node == tmp) {
                /*
                 * Case 2 - left rotate at parent
                 *
                 *      G             G
                 *     / \           / \
                 *    p   U  -->    n   U
                 *     \             /
                 *      n           p
                 *
                 * This still leaves us in violation of 4), the
                 * continuation into Case 3 will fix that.
                 */
                parent->rb_right = tmp = node->rb_left;
                node->rb_left = parent;
                if (tmp)
                        rb_set_parent_color(tmp, parent,
                                            RB_BLACK);
                rb_set_parent_color(parent, node, RB_RED);
                augment_rotate(parent, node);
                parent = node;
                tmp = node->rb_right;
        }

        /*
         * Case 3 - right rotate at gparent
         *
         *        G           P
         *       / \         / \
         *      p   U  -->  n   g
         *     /                 \
         *    n                   U
         */
        gparent->rb_left = tmp;  /* == parent->rb_right */
        parent->rb_right = gparent;
        if (tmp)
                rb_set_parent_color(tmp, gparent, RB_BLACK);
        __rb_rotate_set_parents(gparent, parent, root, RB_RED);
        augment_rotate(gparent, parent);
        break;
} else {
        tmp = gparent->rb_left;
        if (tmp && rb_is_red(tmp)) {
                /* Case 1 - color flips */
                rb_set_parent_color(tmp, gparent, RB_BLACK);
                rb_set_parent_color(parent, gparent, RB_BLACK);
                node = gparent;
                parent = rb_parent(node);
                rb_set_parent_color(node, parent, RB_RED);
                continue;
```

```
                        }

                        tmp = parent->rb_left;
                        if (node == tmp) {
                                /* Case 2 - right rotate at parent */
                                parent->rb_left = tmp = node->rb_right;
                                node->rb_right = parent;
                                if (tmp)
                                        rb_set_parent_color(tmp, parent,
                                                            RB_BLACK);
                                rb_set_parent_color(parent, node, RB_RED);
                                augment_rotate(parent, node);
                                parent = node;
                                tmp = node->rb_left;
                        }

                        /* Case 3 - left rotate at gparent */
                        gparent->rb_right = tmp;  /* == parent->rb_left */
                        parent->rb_left = gparent;
                        if (tmp)
                                rb_set_parent_color(tmp, gparent, RB_BLACK);
                        __rb_rotate_set_parents(gparent, parent, root, RB_RED);
                        augment_rotate(gparent, parent);
                        break;
                }
        }
}
```

Studying this code, one will notice that the scheduler allows for an augmented algorithm function to be used for rbtree placement. The arguments for the function include an anonymous function, *augment_rotate, which has a default set within the kernel, but which is designed as a callback function. The function is called on line 138 of rbtree.c, again on line 157 and finally on 180. The function call is set up by default within rbtree.c under the following instructions

```
void rb_insert_color(struct rb_node *node, struct rb_root *root)
{
        __rb_insert(node, root, dummy_rotate);
}
EXPORT_SYMBOL(rb_insert_color);
```

and

```
static inline void dummy_rotate(struct rb_node *old, struct rb_node *new) {}
```

Flow of Information and Data Typing:

If we return to the graphs of the data hierarchy for kernel tasks, scheduling entities and rbtree nodes, we can identify a few C programming oddities and perplexing problems. First of all, in C

programming, data objected embedded in structs do not know what their container object is. For example, if you code:

```
  struct autombile{
        char [500] wheels;
       int pressure;
} olds1978;
int flat;
strcpy(olds1978.wheels, “Firestone”);
olds1978.pressure=17;

flat = olds1978.pressure;
printf(“%i \n”, flat);
```

The one thing that neither “flat” or “pressure” know is that their data came from olds1978 nor does “pressure” know it is even a member of olds1978. This is not object oriented code and there is not concept of “this”. So how does rbnodes and sched_entity's know which instances of tasks that they are associated with?

Let's examine a specific example of this problem within the kernel code. The kernel scheduler will always run the left most process of the RBTree, which is the process with the least amount of CPU time as measured by vruntime which is a data record described in struct sched_entity and would therefore be instantiated by the creation of a task_struct. So we are sorting on a field within the parent data structure. Function __pick_first_entity() plucks that leftmost node from the cfs run queue in code that looks like this:

__pick_first_entity in ../kernel/sched/fair.c

```
struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
{
        struct rb_node *left = cfs_rq->rb_leftmost;

        if (!left)
                return NULL;

        return rb_entry(left, struct sched_entity, run_node);[5]
```

5 Note that this function has been changed. It was called `__pick_next_entity()` but now is that function is different. Also, this file has been moved to a new location. I mention this because the source for this information is in the Love text, and since its printing, this part of the kernel has undergone substantial change.

```
}
```

This function takes an argument of struct cfs_rq and returns a pointer to a struct sched_entity. How dies it aquire that data structure from an operation that seems to take as its only unique information and rb_node. rb_nodes only that information about their color of its parents and pointers to its left and right children nodes in the rbtree. Furthermore, no C function can take a data type as an argument. And what is run_node. Obviously this is not standard C code. And so we are introduced to an important C Marco that the Gnu/Linux Kernel utilizes called **container_of**.

Linux-x.xx/include/linux/rbtree.h – Line 50 has the following macro.

```
#define rb_entry(ptr, type, member) container_of(ptr, type, member)
```

container_of comes directly kernel.h - Linux-x.xx/include/linux/kernel.h, line 798 in the 3.19.3 codebase.

```
#define container_of(ptr, type, member) ({                \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);        \
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

The container_of macro is a core GNU/Linux kernel component that comes up repeatedly. It was written initially by Greg Kroah-Hartman when he was trying to code a more stable method for recording and configuring hot plugin devices for what would develop into devfs. [6] It attempts to give what would be otherwise useful information in an object oriented programming language to C code by extracting the container object information for a child object. The macro is very difficult to understand even when explained by its author. In order to understand it correctly, one needs to know the proper meaning and usage of C macro language, gcc specific macros, in this case typeof and offsetof, and a detailed understanding of pointer arithmetic.

First lets take a quick review of macros syntax, particularly parameterized macros. Parameterized Marcos are defined as follows

```
#define identifier (x1 , x2 , … , xn ) replacement list [7]
```

The replacement list is critical because it defines the desired behavior of your macro. For example

```
#define TWO_PI (2*3.14159)
#define SCALE(x) ((x) * 10)
```

6 Oram, Andy and Wilson, Greg: Beautiful Code 2007 O'Reilly pg 271

7King, K.N.: C Programming a Modern Approach WW Norton and Company 1996: pg 279

```
j = SCALE(TWO_PI);
```

becomes

```
j = SCALE(2*3.14159);
j = ( (2.31459) * 10 )
```

It is important to note here the critical role of the parenthesis in the macro scale and its affect on the final order of operations of the evaluated C statement. Longer macros use curly braces. Doing though needs to done thoughtfully and with caution. The following example from Kings[8]

```
#define ECHO(s) {get(s); puts(s);}
```

when called as

```
if(echo_flag)
    ECHO(str);
else
  gets(str);
```

this is expanded to an unexpected statement in C

```
if(echo_flag)
{ gets(str); puts(str); } ;
else
gets(str)
```

and you end up with a null statement in your if construction. One can use commas in order to try to mitgate such errors. Be aware that you can embed other macros, such as in the replacement list. However a macro cannot generate a preprocessor directive, so #defines cannot nest in that sense. You need to use #ifdef or #ifndef

Now, lets examine this **typeof** marco command that is part of the gcc compiler, and which is the needed to understand this macro' role in our scheduler. Typeof is defined in the gcc documentation and is located on the GNU website[9]

> 6.6 Referring to a Type with typeof
>
> Another way to refer to the type of an expression is with typeof. The syntax of using of this keyword looks like sizeof, but the construct acts semantically like a type name defined with typedef.
>
> There are two ways of writing the argument to typeof: with an expression or with a type. Here is an example with an expression:
>
> typeof (x[0](1))

8 As above 286

9 https://gcc.gnu.org/onlinedocs/gcc/Typeof.html

This assumes that x is an array of pointers to functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to int.

If you are writing a header file that must work when included in ISO C programs, write __typeof__ instead of typeof. See Alternate Keywords.

A typeof construct can be used anywhere a typedef name can be used. For example, you can use it in a declaration, in a cast, or inside of sizeof or typeof.

The operand of typeof is evaluated for its side effects if and only if it is an expression of variably modified type or the name of such a type.

typeof is often useful in conjunction with statement expressions (see Statement Exprs). Here is how the two together can be used to define a safe "maximum" macro which operates on any arithmetic type and evaluates each of its arguments exactly once:

```
#define max(a,b) \
  ({ typeof (a) _a = (a); \
      typeof (b) _b = (b); \
    _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for a and b. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

Essentially, where ever you might need a type definition, such as in a variable definition or within a cast, one can use the **typeof** macro as a substitute. In our macro, we have the expression:

```
typeof( ((type *)0)->member )
```

whatever object type the preprocessor can evaluate and substitute for within the parenthesis is analyzed to determine its datatype and is then substituted into the C expression. In our case, **type** is the second parameter of **container_of** and **member** is the third parameter, and they are substituted in from the substitution list. This forms a null pointer of presumably a struct type which points to a member and can then be interpreted by the preprocessor as a data datatype of TYPENAME where TYPENAME can be any datatype for which member can be declared as.

Once we have the datatype of the member, we can create a pointer of that datatype and assign it the pointer value that our macro obtained from the first parameter **ptr,** which in theory is a member of a specific unknown instance of parent struct of **type.**

Offsetof, however, is a standard ansi C macro is stddef.h and has a standard manpage on most Gnu/Linux distributions. An except from the man page is as follows:

NAME

offsetof - offset of a structure member

SYNOPSIS

#include <stddef.h>

size_t offsetof(type, member);

DESCRIPTION

The macro offsetof() returns the offset of the field member from the start of the structure type.

Subtracting the offset in bytes from the pointer than points to the byte in memory of a member of a structure should return you to the first byte of the containing structure. This macro was written by Greg Kroah-Hartman, and he notes that the cause for confusion of this macro is likely to be a problem understanding pointer math. [10] The real problem is understanding the macro language, its requirements within the parameter list, and the details of its internal workings.

As applied to the scheduler, what it allows for is access of sched_entity's from simple rb_nodes, and that provides a considerable performance boost over having to move tasks from their run queues and wait queues to manipulate them in an RBTree.

Now that we can find our sched_entity for a related rb_nod, we can now examine the function that finds an tasks' location on the bottom of the rbtree. Recall that when working with a rbtree, the first step is to insert the node in a normal fashion for a Binary Search Tree. That is, first, we find the “unbalanced” location for the node, and then we can balance the tree. In this case, since we have a RBTree, we use previously described _rb_insert() function in order to balance the tree. The function that performs BST insertion is __enquue_entity() . This is the function that ties together many of the concepts on the CFS that we have thus far described:

The code is found in linux-3.19.3/kernel/sched/fair.c

10 Above: Oram and Wilson: Pg 268

```
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
        struct rb_node *parent = NULL;
        struct sched_entity *entry;
        int leftmost = 1;

        /*
         * Find the right place in the rbtree:
         */
        while (*link) {
                parent = *link;
                entry = rb_entry(parent, struct sched_entity, run_node);
                /*
                 * We dont care about collisions. Nodes with
                 * the same key stay together.
                 */
                if (entity_before(se, entry)) {
                        link = &parent->rb_left;
                } else {
                        link = &parent->rb_right;
                        leftmost = 0;
                }
        }

        /*
         * Maintain a cache of leftmost tree entries (it is frequently
         * used):
         */
        if (leftmost)
                cfs_rq->rb_leftmost = &se->run_node;

        rb_link_node(&se->run_node, parent, link);
        rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
```

```
static inline int entity_before(struct sched_entity *a,
                                struct sched_entity *b)
{
        return (s64)(a->vruntime - b->vruntime) < 0;
}
```

The key features of this function, as it lives in the 3.19.3 kernel, is the use of the container_of macro to get access to the sched_entity. The sched_entity has the vruntime data within it. Vruntime is the contains the adjusted value of time that a process has accumulated and more importantly for understanding our code, it is the key value of the rbtree. This line is highlighted in yellow. After retrieving the sched_entity, a static inline function is called to make the comparison and return a Boolean value. Note that vruntimes are 64 bit integers which are casted to a customized data type (s64). This function and its call is outlines in pink.

CFS Implementation Details

The heart of the CFS scheduler is the RBTree described above. However, there are a few details on implementation that are worth notating and learning in order to understand how the scheduler works.. The GNU/Linux operating system has run queues of tasks and wait queues of tasks.[11] The operating system is preemtive multitasking. This means that under most circumstances, the kernel can interrupt running processes at will, and that tasks are assigned to share CPU resources. When there is more than one CPU, which is true in the majority of cases in today's computing environment, CPU's are used simultaneously by tasks.

CFS logic to choosing tasks to run is based on p->se.vruntime where se is a sched_entity. It alsways tries to run the task with the smallest such value. Values are affected by nice priorities which can weigh down a vruntime value.[12]

The run queue is held in the struct rq. rq->cfs.min_vruntime is the monotonic increasing value tracking the smallest vruntime among all tasks in the queue. It is a measure of the comlete work the system has done. That value helps place new tasks as far left as possible on the RBTree.[13]

Scheduling begins with hardware clocks which translate into software interrupts which keep tracks of time and controls processes and scheduling. A processes gets placed on a CPU, and at schedule interrupt (a schedule tick) or other event that interrupts the process, an accounting is taken of the time spent on the cpu. That accounting is stored in p->se.vruntime. When vruntime rises above the left most process on the rbtree, then the processes are swapped out. There is a granulaity fudge factor built into the algorithm prevent thrashing of processes.[14] The granularity can be set on a live system by accessing /proce/sys/kernel/sched_min_granularity_ns.

The default Gnu/Linux scheduler also has scheduling classes. Everything described previously has been a description of the default fair scheduler. There are two other classes, SCHED_FIFO and SCHED_RR which are defined within sched/rt.c. They allow for real time scheduling within the framework of the rbtree based fair scheduler. Getting processes, therefor, from the scheduler, is not just as simple as plucking the most left process from the rbtree. There are a few higher level steps that are part of the gauntlet prior to that step. The kernel calls for __schedule() and the ancillary functions located in ../linux-3.xx.x/kernel/sched/core.c . The function that prioritized class selection is pick_next_task(struct rq) .

---

11 above: Love: Chapter 4
12 CFS Source Code Documention
13 ibid.
14 ibid.

Declared in ../linux-3.xx.x/kernel/sched/sched.h as

struct task_struct * (*pick_next_task) (struct rq *rq, struct task_struct *prev) if is defined at follows:

```
/*
 * Pick up the highest-prio task:
 */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev)
{
        const struct sched_class *class = &fair_sched_class;
        struct task_struct *p;

        /*
         * Optimization: we know that if all tasks are in
         * the fair class we can call that function directly:
         */
        if (likely(prev->sched_class == class &&
                   rq->nr_running == rq->cfs.h_nr_running)) {
                p = fair_sched_class.pick_next_task(rq, prev);
                if (unlikely(p == RETRY_TASK))
                        goto again;

                /* assumes fair_sched_class->next == idle_sched_class */
                if (unlikely(!p))
                        p = idle_sched_class.pick_next_task(rq, prev);

                return p;
        }

again:
        for_each_class(class) {
                p = class->pick_next_task(rq, prev);
                if (p) {
                        if (unlikely(p == RETRY_TASK))
                                goto again;
                        return p;
                }
        }

        BUG(); /* the idle class will always have a runnable task */
}
```

This is a bit of a rewriting of the previous versions of this function since its introduction. But the notable implementation here is the for_each_class that walks through classes until it find the class of the highest priority with waiting tasks. class->pick_next_task(rq, prev); calls the __pick_next_entity() function that we studied above.

for_each_class is a macro defined as follows

```
#define for_each_class(class) \
   for (class = sched_class_highest; class; class = class->next)
```

which is defined in linux/3.xx.x/kernel/sched/sched.h

Queue Manipulation of Tasks

We examined task_struct which stores task information and determined the relationship between task_struct and the rbtree of rbnodes, and the container struct sched_entity. Together this forms a system for runtime queues. In order to make access to such queues easy, and possibly backward compatible to previous scheduling systems, the Gnu/Linux Kernel also has a struct that that acts asa place holder and descriptor for the entire queue. This is struct cfs_rq and it is defined in

..linix-3.xx.x/kernel/sched/sched.h as follows:

```
/* CFS-related fields in a runqueue */
struct cfs_rq {
        struct load_weight load;
        unsigned int nr_running, h_nr_running;

        u64 exec_clock;
        u64 min_vruntime;
#ifndef CONFIG_64BIT
        u64 min_vruntime_copy;
#endif

        struct rb_root tasks_timeline;
        struct rb_node *rb_leftmost;

        /*
         * 'curr' points to currently running entity on this cfs_rq.
         * It is set to NULL otherwise (i.e when none are currently running).
         */
        struct sched_entity *curr, *next, *last, *skip;

#ifdef CONFIG_SCHED_DEBUG
        unsigned int nr_spread_over;
#endif

#ifdef CONFIG_SMP
        /*
         * CFS Load tracking
         * Under CFS, load is tracked on a per-entity basis and aggregated up.
         * This allows for the description of both thread and group usage (in
         * the FAIR_GROUP_SCHED case).
         */
        unsigned long runnable_load_avg, blocked_load_avg;
        atomic64_t decay_counter;
        u64 last_decay;
        atomic_long_t removed_load;

#ifdef CONFIG_FAIR_GROUP_SCHED
        /* Required to track per-cpu representation of a task_group */
        u32 tg_runnable_contrib;
        unsigned long tg_load_contrib;

        /*
```

```
         *   h_load = weight * f(tg)
         *
         * Where f(tg) is the recursive weight fraction assigned to
         * this group.
         */
        unsigned long h_load;
        u64 last_h_load_update;
        struct sched_entity *h_load_next;
#endif /* CONFIG_FAIR_GROUP_SCHED */
#endif /* CONFIG_SMP */

#ifdef CONFIG_FAIR_GROUP_SCHED
        struct rq *rq; /* cpu runqueue to which this cfs_rq is attached */

        /*
         * leaf cfs_rqs are those that hold tasks (lowest schedulable entity in
         * a hierarchy). Non-leaf lrqs hold other higher schedulable entities
         * (like users, containers etc.)
         *
         * leaf_cfs_rq_list ties together list of leaf cfs_rq's in a cpu. This
         * list is used during load balance.
         */
        int on_list;
        struct list_head leaf_cfs_rq_list;
        struct task_group *tg;/* group that "owns" this runqueue */

#ifdef CONFIG_CFS_BANDWIDTH
        int runtime_enabled;
        u64 runtime_expires;
        s64 runtime_remaining;

        u64 throttled_clock, throttled_clock_task;
        u64 throttled_clock_task_time;
        int throttled, throttle_count;
        struct list_head throttled_list;
#endif /* CONFIG_CFS_BANDWIDTH */
#endif /* CONFIG_FAIR_GROUP_SCHED */
};
```

Highlighted here are some of the entries which we have already discussed. A lot of the code for this structure is conditional for various hardware and configuration considerations. Functions which calculate load, vmruntime, run orders, and accounting all access this structure in order to get a snapshot of the current condition of the run queue or to inform the run queue of some change.

In addition to task_struct's being part of the run queue and on the rbtree, they can be otherwise sleeping and waiting. It is not the intention of this paper to get into detailed description of the wait queue, and how it can be signaled to wake tasks when a task comes off the CPU, for whatever reason, the vruntime is calculated but it is not necessarily the destiny for a task to be put right back into the rbtree. The task might be put onto a wait queue, and in fact, a wait queue might need to be created for the task. Then the task sleeps. When it awakes it needs to run schedule() and go back onto the rbtree.

Wait lists are simple linked lists such as this:

under kernel/sched/wait.c

```
void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
{
        unsigned long flags;
        wait->flags &= ~WQ_FLAG_EXCLUSIVE;
        spin_lock_irqsave(&q->lock, flags);
        __add_wait_queue(q, wait);
        spin_unlock_irqrestore(&q->lock, flags);
}
```

include/linux/wait.h

```
static inline void __add_wait_queue(wait_queue_head_t *head, wait_queue_t *new)
{
        list_add(&new->task_list, &head->task_list);
}
```

include/linux/lists.h

```
static inline void __list_add(struct list_head *new,
                              struct list_head *prev,
                              struct list_head *next)
{
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}
```

Kernel Development

Source code for the kernel is available from kernel.org. The latest stable release can be downloaded and compiled. The basics to kernel compilation is, configure, make/compile install and add to the boot loaders of your system. Years going by, this process has been simpler. Today there are more problems and the methodology is not as straight forward as it has been previously.

The Kernel comes with documentation on its installation and its use. These documents are a good starting point and one needs to read the README file in the kernel source tree. But some of the details have been outdated and distributions have increasingly customized installation tools, and our boot loaders have changed from lilo to grub to grub2, to UEFI. Configuration files have also become

very complicated. Every piece of hardware needs new module and has new option. Configuring the config file can get very bewildering even for experienced coders and long time GNU/Linux users. The more understanding you have about compiling and configuring the kernel, the more control you have of your system.

For developers there is now the advange of running GNU/Linux within a virtual machine. Previously, if you made a coding error in your development code, your computer would crash and you would need to do a hard reboot. Today you can just restart your virtual machine with a stable kernel binary. The residence of the kernel binaries is located under /boot

```
[ruben@stat13 boot]$ ls -l
total 71136
drwxr-xr-x 3 root root       80 Nov 13 05:03 EFI
drwxr-xr-x 6 root root      192 Mar 21 04:25 grub
-rw-r--r-- 1 root root 16108705 Dec 29 18:18 initramfs-310-x86_64-fallback.img
-rw-r--r-- 1 root root  2830044 Dec 29 18:18 initramfs-310-x86_64.img
-rw-r--r-- 1 root root 17887753 Dec 29 17:39 initramfs-314-x86_64-fallback.img
-rw-r--r-- 1 root root  2901576 Dec 29 17:39 initramfs-314-x86_64.img
-rw-r--r-- 1 root root 18271088 Mar 13 22:24 initramfs-319-x86_64-fallback.img
-rw-r--r-- 1 root root  2920830 Mar 13 22:24 initramfs-319-x86_64.img
-rw-r--r-- 1 root root       22 Dec 17 12:21 linux310-x86_64.kver
-rw-r--r-- 1 root root       22 Dec 17 14:55 linux314-x86_64.kver
-rw-r--r-- 1 root root       21 Mar  7 14:01 linux319-x86_64.kver
drwxr-xr-x 2 root root       80 Oct  6  2013 memtest86+
drwxr-xr-x 2 root root       80 Oct 13  2014 syslinux
-rw-r--r-- 1 root root  3793728 Dec 17 12:21 vmlinuz-310-x86_64
-rw-r--r-- 1 root root  3881680 Dec 17 14:55 vmlinuz-314-x86_64
-rw-r--r-- 1 root root  4146640 Mar  7 14:01 vmlinuz-319-x86_64
```

initrafs is a commonly used to work with tmpfs for booting. Partitioning of file systems and disks now need special attention for kernel developers if they chose to boot there computers. And grub2, which is the standard GNU boot loader, on x86/64 systems, often will look only for file names using the following patterns:

/boot/vmlinuz-* /vmlinuz-* /boot/kernel-*[15]

15 For example, in the grub config files generated by grub, most often have config scripts that include the following line in the shell script

```
machine=`uname -m`
case "x$machine" in
    xi?86 | xx86_64)
   list=`for i in /boot/vmlinuz-* /vmlinuz-* /boot/kernel-* ; do
                if grub_file_is_not_garbage "$i" ; then echo -n "$i " ; fi
            done` ;;
    *)
   list=`for i in /boot/vmlinuz-* /boot/vmlinux-* /vmlinuz-* /vmlinux-* /boot/kernel-* ; do
```

The kernel compile scripts will write out files with the name vmlinuz, which doesn't match the grub2 requirement. So there are now numerous hurdles that need to need to be worked through in order to make a comfortable environment for kernel development.

Another hurdle that now exists is the problems introduced with UEFI. UEFI as a specification adds so many complexities that they are beyond the scope of this document. Unfortunately, that means that the actual technique to install a custom kernel on the readers system has now become so complex that it can no longer be rationally covered within a paper that covers kernel development. It has perplexed advanced and ordinary users alike. Here is what Kernel developer Greg KH considers a **simple** method to boot signed custom kernels:

Booting a Self-signed Linux Kernel

September 2nd, 2013

Now that The Linux Foundation is a member of the UEFI.org group, I've been working on the procedures for how to boot a self-signed Linux kernel on a platform so that you do not have to rely on any external signing authority.

After digging through the documentation out there, it turns out to be relatively simple in the end, so here's a recipe for how I did this, and how you can duplicate it yourself on your own machine.

We don't need no stinkin bootloaders!

When building your kernel image, make sure the following options are set:

```
1  CONFIG_EFI=y
2  CONFIG_EFI_STUB=y
3  ...
4  CONFIG_FB_EFI=y
5  ...
6  CONFIG_CMDLINE_BOOL=y
7  CONFIG_CMDLINE="root=..."
8  ...
9  CONFIG_BLK_DEV_INITRD=y
10 CONFIG_INITRAMFS_SOURCE="my_initrd.cpio"
```

The first two options here enable EFI mode, and tell the kernel to build itself as a EFI binary that can be run directly from the UEFI bios. This means that no bootloader is involved at all in the system, the UEFI bios just boots the kernel, no "intermediate" step needed at all. As much as I love gummiboot, if you trust the kernel image you are running is "correct", this is the simplest way to boot a signed kernel.

As no bootloader is going to be involved in the boot process, you need to ensure that the kernel knows where the root partition is, what init is going to be run, and anything else that the bootloader normally passes to the kernel image. The option listed above, CONFIG_CMDLINE should be set to whatever you want the kernel to use as the command line.

Also, as we don't have an initrd passed by the bootloader to the kernel, if you want to use one, you need to build it into the kernel itself. The option CONFIG_INITRAMFS_SOURCE should be set to your pre-built cpio initramfs image you wish to use.

Note, if you don't want to use an initrd/initramfs, don't set this last option. Also, currently it's a bit of a pain to build the kernel, build the initrd using dracut with the needed dracut modules and kernel modules, and then rebuild the kernel adding the cpio image to the kernel image. I'll be working next on taking a pre-built kernel image, tearing it apart and adding a cpio image directly to it, no need to rebuild the kernel. Hopefully that can be done with only a minimal use of libbfd

After setting these options, build the kernel and install it on your boot partition (it is in FAT mode, so that UEFI can find it, right?) To have UEFI boot it directly, you can place it in /boot/EFI/boot/bootx64.efi, so that UEFI will treat it as the "default" bootloader for the machine.

```
                if grub_file_is_not_garbage "$i" ; then echo -n "$i " ; fi
        done` ;;
esac
```

Lather, rinse, repeat

After you have a kernel image installed on your boot partition, it's time to test it.

Reboot the machine, and go into the BIOS. Usually this means pounding on the F2 key as the boot starts up, but all machines are different, so it might take some experimentation to determine which key your BIOS needs. See this post from Matthew Garrett for the problems you might run into trying to get into BIOS mode on UEFI-based laptops.

Traverse the BIOS settings and find the place where UEFI boot mode is specified, and turn it the "Secure Boot" option OFF.

Save the option and reboot, the BIOS should find the kernel located at boot/EFI/boot/bootx64.efi and boot it directly. If your kernel command line and initramfs (if you used one) are set up properly, you should now be up and running and able to use your machine as normal.

If you can't boot properly, ensure that your kernel command line was set correctly, or that your initramfs has the needed kernel modules in it. This usually takes a few times back and forth to get all of the correct settings properly configured.

Only after you can successfully boot the kernel directly from the BIOS, in "insecure" mode should you move to the next step.

Keys to the system

Now that you have a working kernel image and system, it is time to start messing with keys. There are three different types of UEFI keys that you need to learn about, the "Platform Key" (known as a "PK"), the "Key-Exchange Keys" (known as a "KEK"), and the "Signature Database Key" (known as a "db"). For a simple description of what these keys mean, see the Linux Foundation Whitepaper about UEFI Secure boot, published back in 2011. For a more detailed description of the keys, see the UEFI Specification directly.

For a *very* simple description, the "Platform Key" shows who "owns and controls" the hardware platform. The "Key-Exchange keys" shows who is allowed to update the hardware platform, and the "Signature Database keys" show who is allowed to boot the platform in secure mode.

If you are interested in how to manipulate these keys, replace them, and do neat things with them, see James Bottomley's blog for descriptions of the tools you can use and much more detail than I provide here.

To manipulate the keys on the system, you need the the UEFI keytool USB image from James's website called sb-usb.img (md5sum 7971231d133e41dd667a184c255b599f). dd the image to a USB drive, and boot the machine into the image.

Depending on the mode of the system (insecure or secure), you will be dropped to the UEFI console, or be presented with a menu. If a command line, type KeyTool to run the keytool binary. If a menu, select the option to run KeyTool directly.

Save the keys

First thing to do, you should save the keys that are currently on the system, in case something "bad" ever happens and you really want to be able to boot another operating system in secure mode on the hardware. Go through the menu options in the KeyTool program and save off the PK, KEK, and db keys to the USB drive, or to the hard drive, or another USB drive you plug into the system.

Take those keys and store them somewhere "safe".

Clear the machine

Next you should remove all keys from the system. You can do this from the KeyTool program directly, or just reboot into the BIOS and select an option to "remove all keys", if your BIOS provides this (some do, and some don't.)

Create and install your own keys

Now that you have an "empty" machine, with the previous keys saved off somewhere else, you should download the sbsigntool and efiutil packages and install them on your development system. James has built all of the latest versions of these packages in the openSUSE build system for all RPM and DEB-based Linux distros. If you have a Gentoo-based system, I have checked the needed versions into portage, so just grab them directly from there.

If you want to build these from source, the sbsigntool git tree can be found here, and the efitools git tree is here.

The efitools README is a great summary of how to create new keys, and here is the commands it says to follow in order to create your own set of keys:

```
1
2
3
4
5
6
7
8
```

```
# create a PK key
openssl req -new -x509 -newkey rsa:2048 -subj "/CN=my PK name/" -keyout PK.key -out PK.crt -days 3650 -nodes -sha256

# create a KEK key
openssl req -new -x509 -newkey rsa:2048 -subj "/CN=my KEK name/" -keyout KK.key -out KK.crt -days 3650 -nodes -sha256

# create a db key
openssl req -new -x509 -newkey rsa:2048 -subj "/CN=my db name/" -keyout db.key -out db.crt -days 3650 -nodes -sha256
```

The option -subj can contain a string with whatever name you wish to have for your key, be it your company name, or the like. Other fields can be specified as well to make the key more “descriptive”.

Then, take the PK key you have created, turn it into a EFI Signature List file, and add a GUID to the key:

```
1 cert-to-efi-sig-list -g <my random guid> PK.crt PK.esl
```

Where my random guid is any valid guid you wish to use (I’ve seen some companies use all ‘5’ as their guid, so I’d recommend picking something else a bit more “random” to make look like you know what you are doing with your key…).

Now take the EFI Signature List file and create a signed update file:

```
1 sign-efi-sig-list -k PK.key -c PK.crt PK
PK.esl PK.auth
```

For more details about the key creation (and to see where I copied these command lines from), see James’s post about owning your own Windows 8 platform.

Take these files you have created, put them on a USB disk, run the KeyTool program and use it to add the db, KEK, and PK keys into the BIOS. Note, apply the PK key last, as once it is installed, the platform will be “locked” and you should not be able to add any other keys to the system.

Fail to boot

Now that your own set of keys is installed in the system, flip the BIOS back into “Secure boot” mode, and try to boot your previous-successful Linux image again.

Hopefully it should fail with some type of warning, the laptop I did this testing on provides this “informative” graphic:

Sign your kernel

Now that your kernel can’t boot, you need to sign it with the db key you placed in your bios:

```
1 sbsign --key db.key --cert db.crt --output bzImage
bzImage.signed
```

Take the bzImage.signed file and put it back in the boot partition, copying over the unsigned /boot/EFI/boot/bootx64.efi file.

Profit!

Now, rebooting the machine should cause the UEFI bios to check the signatures of the signed kernel image, and boot it properly.[16]

With regard to visualization, which makes kernel development easier, GNU systems have several options. We use Oracle's Virtual Box. There is also Vmware's system that has been available for many years. Most interesting is the development of KVM, which is a free software implementation of virtualization that utilizes the kvm.ko loadable kernel module. [17] [18] Visualization requires both host and client components. There can be considerable integration of mouse and keyboard controls. When you make a new kernel in your virtual machine, you alter the balance of that integration. That is one of the disadvantages of performing these tasks in virtual space.

The methodology that we used was to use oracle's virtualbox, although I would encourage use of the developing kvm. Installing virtualbox can best be done through your distributions package management system since they integrate the host and client tools for the kernels they deliver. It is important to remember that you can not run virtualization without starting host needed kernel modules that allow for virtualization.

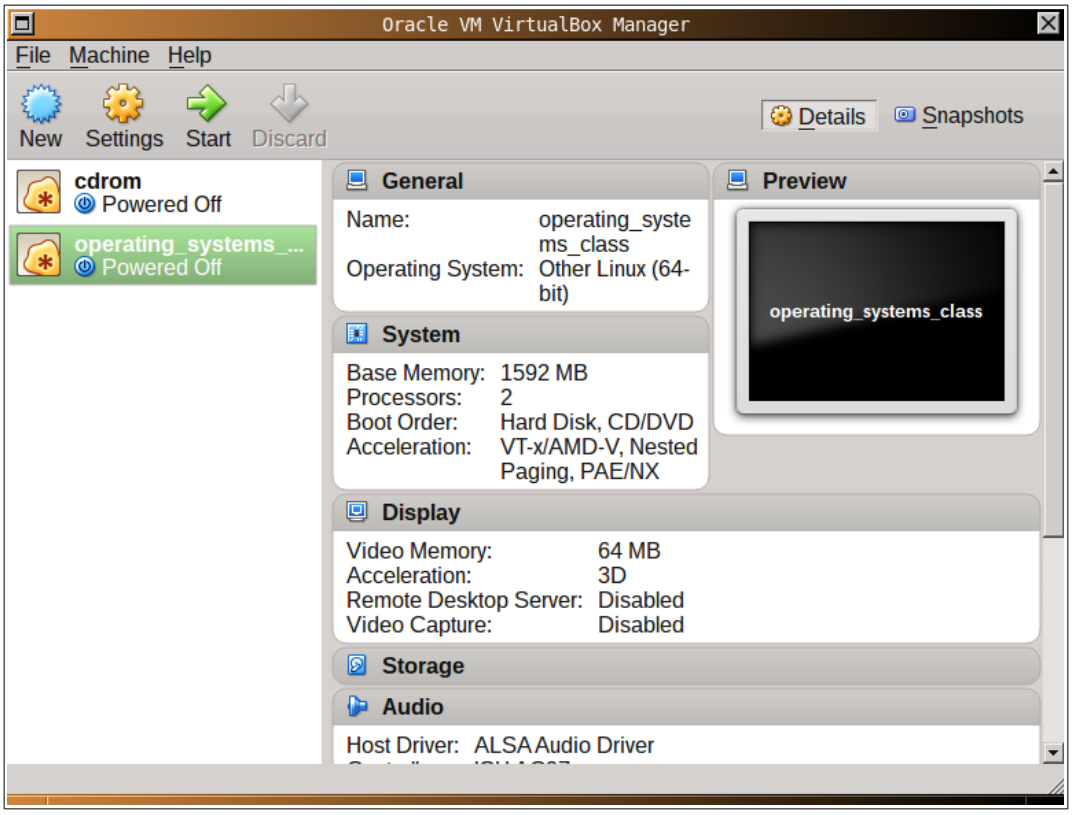
16 Kroah-Hartman, Greg from his website: http://www.kroah.com/log/ current to April, 2015
17 http://www.linux-kvm.org/page/Main_Page
18 http://virt-tools.org/learning/index.html

```
sudo  modprobe vboxdrv
```

you are now ready to run your virtual machine and set up a fresh virtual machine:

```
ruben@host$ virtualbox
```

Setting up virtualization is fairly straightforward through the GUI of the virtual machine. You do need an installation disk medium. We used a flash drive attached to a host workstation with an Manjaro Distribution that runs openrc. [19] This was chosen for a number of reasons but largely it is do to easy of use, minimal resource usage and access to openrc for an initiation system, which just makes it simpler for me to manipulate. The principles presented here should function with any standard distribution.



For the purposes of this paper, we downloaded source code directly from kernel.org. This has a disadvantage of not have the build tools available for the specifics of a distribution, but since we will be swapping out Kernel schedulers, it seemed better to attempt this entirely by hand from a plain vanilla kernel.

19 https://forum.manjaro.org/index.php?PHPSESSID=akh2eqcvbjnmfirkk89l6e1im0&board=50.0

The latest stable kernel is the newly released 4.0 trunk. We downloaded to late 3.0 truck since the 4.0 truck has been released just a few days prior to this writing. We download it to our home directory. Examine the /usr/src directory and assure that it has enough permissions to allow for the user compiling the kernel and for copying the source files into /usr/src. The reason for this is because compiling should be done as a regular user, and not root. Decompress the release you downloaded from kernel.org, which for us is the linux-3.19.5.tar.xz file which is a stable release from 4-19-2015.

```
ruben@host:~/Downloads/ $ unxv linux-3.19.5.tar.xv

ruben@host:~/Downloads/ $ cd /usr/src

ruben@host:/usr/src/ $ tar -xvf /home/ruben/Downloads/linux-3.19.5.tar

ruben@host:/usr/src/ $ cd linix-3.19.5.tar

[ruben@manjaro src]$ ls -al

total 8

drwxrwxrwx  4 root  root    56 Apr 20 13:05 .

drwxr-xr-x  9 root  root   109 Mar  1 05:57 ..

lrwxrwxrwx  1 ruben users   12 Mar 22 14:30 linux -> linux-3.19.2

drwxr-xr-x 24 ruben users 4096 Mar 22 20:32 linux-3.19.2

drwxr-xr-x 23 ruben users 4096 Apr 19 04:11 linux-3.19.5
```

Here we can see a symbolic link between the a source tree and the linux directory. This link helps with the installation, configuring and compiling of kernel and kernel versions.

At this point cd into the proper directory and make mrproper just to clean out dependencies and permision problems possibly left behind by the development team.

```
[ruben@manjaro src]$ cd linux-3.19.5/

[ruben@manjaro linux-3.19.5]$ make mrproper
```

The next step is to create a working configuration file for the kernel compilation. The most straight forward way of do this is to run make menuconfig. This will walk you through hundreds of modules that can be compiled options in software and hardware for every occasion. to say it is a challenge is an understatement. Be prepared to take a few hours to walk through all the choices that the GNU/Linux present to you. In addition, it possible for choices to be in conflict. And just when you think you understand a hardware choice, the specificity of the questions that the kernel configuration is asking can make one go scurrying for detailed documentation about chipset and hardware versions that you run.

For example, you can be working on a Intel Dual Core Lenova think Center, all fairly standard GNU friendly hardware. On kernel configuration though, you discover that some dual cores are related to Xeon chips and others are not. How brushed up are you on your history of Intel chip development? This is not easy information to find, unless you know where to look. It can lead to conversations with online collaborators like that in the boxes below. If you are luck, you develop a relationship with many individuals on the internet who have real knowledge. It takes time to weed such people out, but they do exist. There is a strong element of oral learning to kernel development and computing technology. Everything is not written down, nor is there ope access to much of what is needed to know to accomplish a desired result.

An alternate means of acquiring a decent configuration file can be by asking your current running kernel for one based on its current operations. This has its obvious drawbacks, but it gives you a good starting reference for a configuration file. [20]

```
[ruben@manjaro linux-3.19.5]$ zcat  /proc/config.gz > .config
```

This gives you a very good base level configuration file and we can load it with make menuconfig and load the config file. If done properly it will look like *figure 3*.

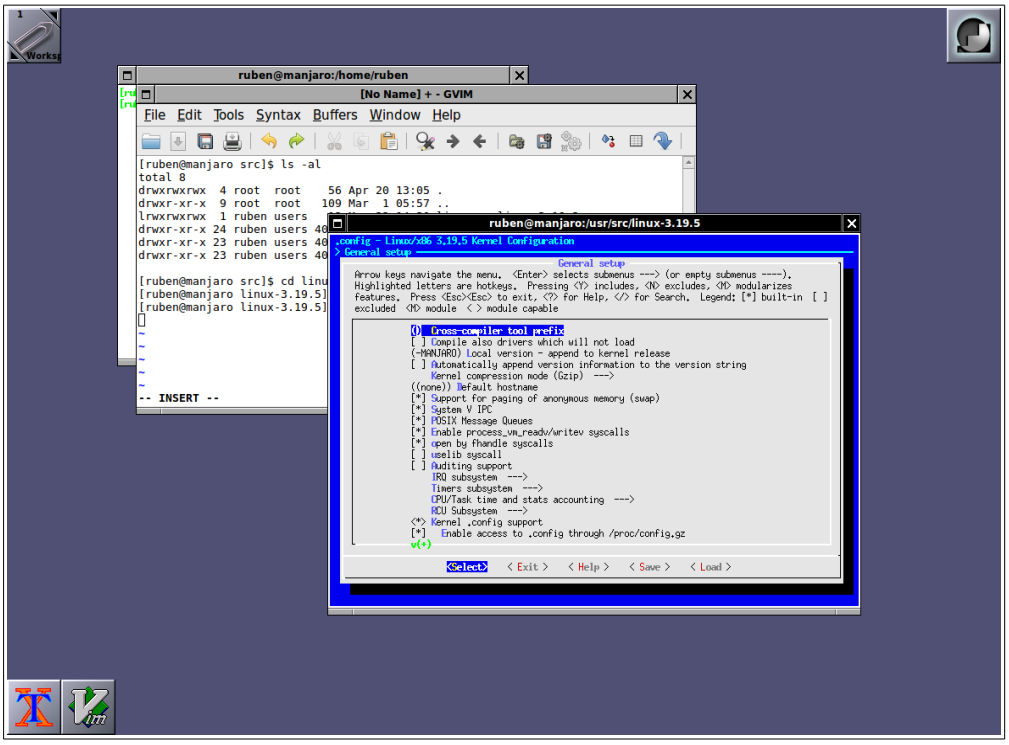


Figure 3 – make menuconfig Kernel Configuration Screen

20 Cronwell, Bob: http://cromwell-intl.com/linux/linux-kernel.html - From the Notes on GNU/Linux kernel configuration. This is a very upto date and complete one man job to document fulling many areas of Unix and Linux administration.

```
   ruben safir conveyed the following to
```

```
comp.os.linux.hardware...
```

```
I'm compiling my own kernel and I'm confused at what I am reading in
the
configuration file  This machine is a Intel Core 2 Duo CPU E8500
and my choices in the menuconfig is:

 |
       | ┌──────────────────────────────────────────────────────────┐
       | │                                                          │
       | │         ( ) Opteron/Athlon64/Hammer/K8                   │
       | │                                                          │
       | │         ( ) Intel P4 / older Netburst based Xeon         │
       | │                                                          │
       | │         ( ) Core 2/newer Xeon                            │
       | │                                                          │
       | │         ( ) Intel Atom                                   │
       | │                                                          │
       | │         (X) Generic-x86-64                               │
       | │                                                          │
       | │                                                          │
       | │                                                          │
       | └──────────────────────────────────────────────────────────┘


So, I think I need the core 2/new Xeon but the help message
 CONFIG_MCORE2:
  Select this for Intel Core 2 and newer Core 2 Xeons (Xeon 51xx and
 53xx) CPUs. You can distinguish newer from older Xeons by the CPU
 family in /proc/cpuinfo. Newer ones have 6 and older ones 15
 (not a typo) Symbol: MCORE2 [=y]
 Type  : boolean
  Prompt: Core 2/newer Xeon
Location:
   -> Processor type and features
  -> Processor family (<choice> [=y])
   Defined at arch/x86/Kconfig.cpu:254
  Depends on: <choice>
```

```
Yes, you need to select "Core2/newer Xeon".  The distinction "newer
Xeons" is because Intel has also made Xeon versions with 64-bit support
based upon the Netburst (Pentium 4) architecture.

--
= Aragorn =
```