

The logo consists of the letters 'NY' stacked above 'LXS' in a bold, white, sans-serif font. The text is centered within a black square that has a thin white border.

**NY
LXS**

NYLXS Journal
April 2015



Published By NYLXS <http://www.mrbrklyn.com>

Author: Fernando Gutierrez, Ruben Safir

ISBN 978-1-329-11030-4

Copyright 2015 under the GPL FDL by NYLXS

Spring is here, and after a very cold winter, activities are heating up in the GNU/Linux community. The key word, there is community. It wasn't long ago that I was listening to a podcast on "Linux" and among the items that was discussed was the VMWARE - Free Software community lawsuit in Germany for violations of the GPL. Evidently, VMWARE had decided to not be such a good member of the community. There has been a long standing problem with VMWARE and the issues have come to boil over. Nor is the problems being sighted against VMWARE exclusive to them. Everyone want their cake and to eat it too. Everyone wants the benefits of a powerful software community based on sharing, but they don't share. They want to pay wall a significant portion of the shared knowledge base. Whats is yours is mine and what is mine is mine. Free Software can't work like that.



Now to be honest, VMWARE is not the worst actor. There is a broader issue of cutting around the core values that the GPL, and to leave the community with only the slimiest pickings. This is the business tactic which is causing a broad problem. This is a problem with Objective C, CLANG, JAVA, Oracle, and also VMWARE. Simon Phipps wrote a fairly comprehensive review of the problem in infoworld ¹ and hits the nail on the head when he writes:

At first glance, this seemed like a conflict over "module loading." Vendors commonly write code their legal teams tell them will "insulate" them from the GPL so that they can use Linux drivers in proprietary code. This is a controversial topic among free software advocates, and a lawsuit clarifying the effectiveness of that strategy would be helpful.

¹ <http://www.infoworld.com/article/2893695/open-source-software/vmware-heading-to-court-over-gpl-violations.html>
VMware heads to court over GPL violations May 5th 2015

Unfortunately, it may not be so simple. It looks like VMware started out originally with an insulating layer to allow its kernel to use unmodified Linux drivers. But analysis published by LWN (sadly paywalled) suggests to me that, while this may have been where VMware started, code from GPL-licensed modules may have migrated into the "insulation layer."

So first, the questionable usage of hide and seek methods designed to insulate closed sourced code from a GPL layer is being used, and this is a primary problem. Then, of course, since they have used our shared code so frequency, and learn so much from our Free Software community, our code magically seems to have now appeared in the parent product.

This is not as terrible as it seems. It is just plain human to use the GPLed free software library of free speech and free code within the free software community, and for it to migrate into all the other projects that one creates. The VMWARE coders learned techniques and resources from the Free Software community. These are intellectual resources that without the GPL, would not be available and which would be a loss to not only the VMWARE developers, but also the rest of society. I would expect to see the GPL code to migrate into the main product and everywhere else, for that matter. And that is why this lawsuit is necessary. The lawsuit and the GPL assure that these benefits remain available to all of us. Everyone remains responsible to the community. The GPL is not about what you TAKE, but about what you GIVE. You can not have freedom without responsibility. And you can not exercise freedom without being willing to guarantee that right to others.

This podcaster seems to fail Freedom and Civics 101. He sneered at the GPL as being tyrannical, and did so with hatefulness. He hates the GPL and raged on and on about it. The GPL protects everyone by making sure that everyone gives to the benefit of the community. Like any other free community, freedom is not a measure of what you TAKE, but a MEASURE of what you GIVE. Freedom is something you give. This is a problem that exceeds just software. Maybe problems in the world come from this basic lack of understanding of the fact that Freedom is a responsibility and a buffet. Dictators around the world coop the language of freedom and democracy to cover for their personal goals while brutalizing their neighbors. Without responsibility to the community, nobody has freedom.



Table of Contents

The NYLXS Journal April 2015.....	1	Kernel Development.....	34
GNU/Linux Kernel Scheduling Design and Alternatives.....	5	Setting Your customized Kernel to Boot....	42
Abstract:.....	5	Testing Exploring Your New Kernel and Scheduler.....	49
Basic Linux Kernel Scheduling System Structure.....	6	Changing Scheduler.....	52
Flow of Information and Data Typing:.....	23	Patching the Kernel.....	63
CFS Implementation Details.....	29	Conclusion.....	70
Queue Manipulation of Tasks.....	31	Open Source Hardware: A Technical Review of the Arduino.....	73

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 Scheduler. Instead of a confusing and complex set of heuristics to try to keep a multiple of system types balanced and in order and to avoid deadlocks, the feuding scheduler project managers who have worked on GNU/Linux 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 than 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 first for CPU run time.** After running, the processes new cumulative CPU resource time gets accounted for, and the task is then rotated back to the tree and placed 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 handheld 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. We will outline the methods, resources and utilities for kernel development code and explore the code base for the scheduler and other implementations. Hopefully, this paper succeeds in teaching about the process of Free Software scheduler development, more than the result. When the process is sound, both human and technical, the results happen. I take that as an axiom.

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 3.0 tree, (the 4.0 kernel was just introduced as this project was being written), 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 interrupted, accounted for the time that was spent on the CPU. If needed the task is then 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**.² 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 is embedded in an environment of wait queues, system design decisions, run queues, standardized kernel specific software containers, task signaling, preemption theory, unix process control, and scheduler theory. We will cover all of this.

Scheduling goes right at the heart of operating systems. Since the Gnu/Linux scheduler is a) code available, b) well studied and c) relatively easy to understand, the best part of studying it is that it intersects so many other part of the operating system that you can draw on many areas of your knowledge base, and use it to learn something new, or to develop something new. **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 GNU/Linux, threads and

² Love, Robert: "Linux Kernel Development 3rd Edition" Page pg 50

processes are differentiated tasks. They are, on the kernel level, fundamentally the same.³ Each is described in the Kernel with **struct task_struct**. Instances of task_struct's are strung along in linked lists in run time queues. They can be **running** 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 1274 to 1704. Since tasks_struct is so core to the system it is defined with quite a few processor directives and is worth looking at its definition as a coding lesson. Application programmers will rarely run into structure declarations nearly 500 lines long and with processor commands.

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;
#endif
};
```

```

        struct list_head rcu_node_entry;
#endif /* #ifndef CONFIG_PREEMPT_RCU */
#ifdef CONFIG_PREEMPT_RCU
        struct rcu_node *rcu_blocked_node;
#endif /* #ifndef 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 /* #ifndef CONFIG_TASKS_RCU */

#ifdef 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];
#ifdef CONFIG_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
#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;
#ifdef 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 minflt, majflt;

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;
#ifdef 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; /* Sequence 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;
#endif
#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;
    };
#endif

```

```

        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, which we will look at in detail. When compiling the GNU/Linux Kernel, the first step is to produce this kernel configuration file. You do this generally running the command

```
ruben@host:$ make menuconfig
```

or some version of this. The object of this command is to produce a working configuration file for the kernel compilation. Details for this exist in the Kernel's 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 <code>./.config</code> 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 `struct sched_entity se`, and `const struct sched_class *sched_class`. `struct sched_entity` is a struct defined in `linux/sched.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_rq;
#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 instances of `task_struct` 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 struct sched_entity which directly represents the node position is **struct rb_node** **run_node** (no that is not a typo). struct 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, and other parts of the kernel 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 below.

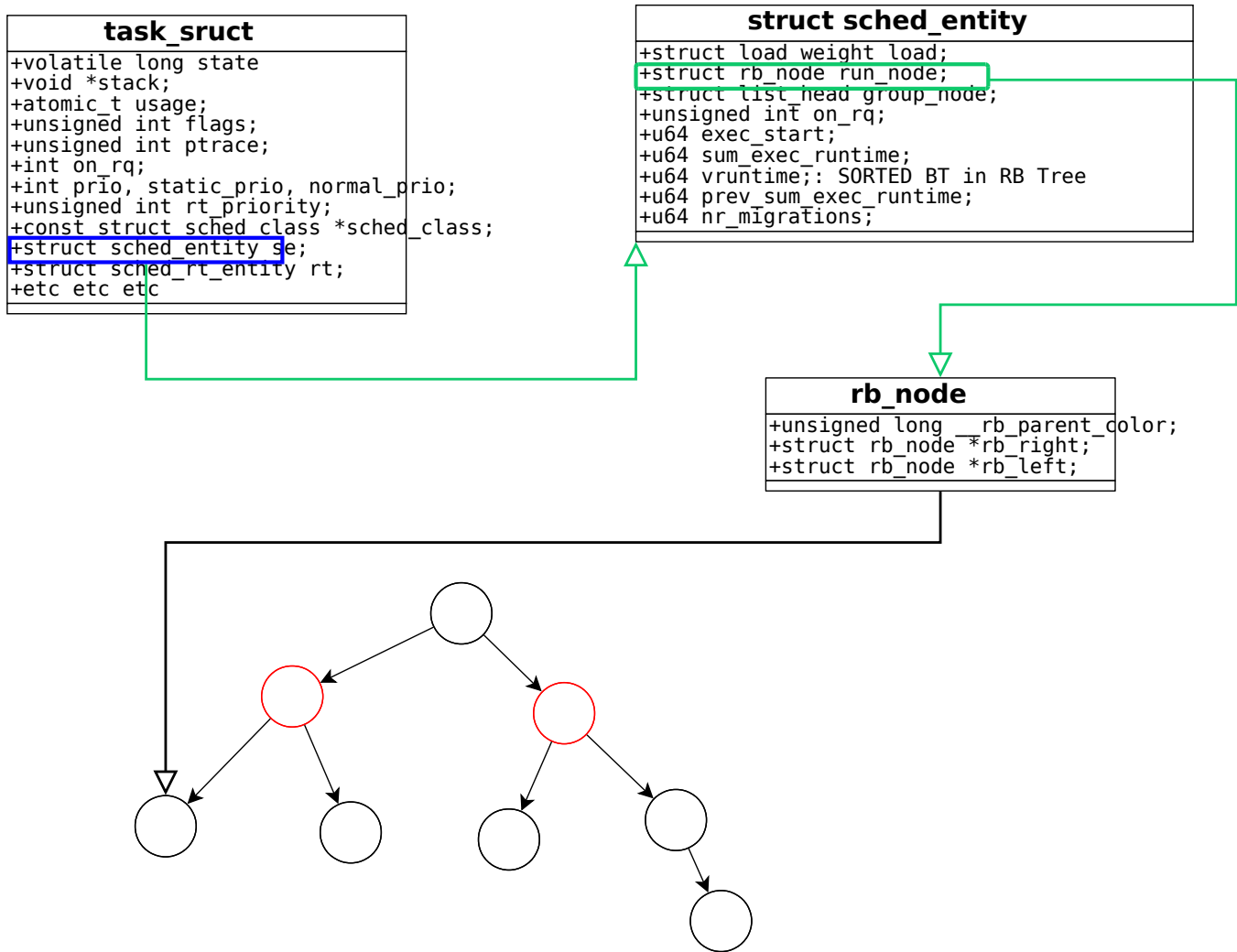


Figure 1 the relations between struct involving the CFS scheduler in GNU/Linux

Programatically 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_to_run_next = 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/Documentation/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 lecture 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.

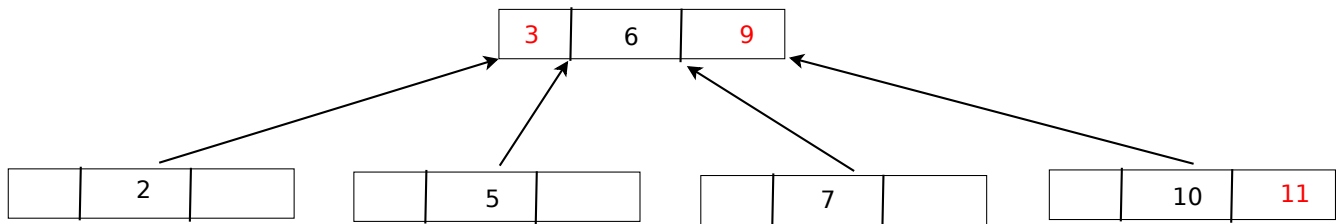
The principles behind a Red Black tree are based on the 2-3-4 multikey tree which can best be described diagrammatically. A 2-3-4 multikey tree (also known as an order 4 tree) is a tree that can have a maximum of 3 indexes per node and therefore a maximum of 4 child nodes. Additionally, we constrain the tree such that tree must be height balanced, 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 2.⁴

Red Black trees have a number of rules and advantages:

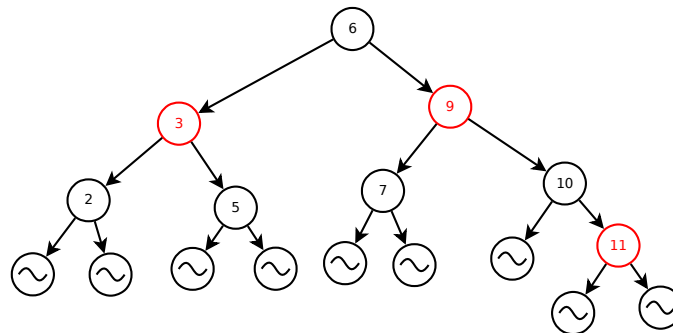
1. The Root Node is Black
2. Red Nodes are the 'wings' of the 2-3-4 multikey arraignment
3. Every Red Node has two black child nodes.
4. Leaf Nodes are black and nil
5. You can not have 2 Red Nodes in a row on the same path
6. The number of black nodes in any path is equal to any other path from the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

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

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



2-3-4 Multi Key B Tree



Translation to a Red Black Tree



Figure 2: Red Black Tree

Rotation of nodes is needed when the path which is affected by the addition where we have end up with a height length one more in length than sister paths(1). Each new entry is Red, inserted in a standard way as with any BST tree(2). When we have two red nodes come together we need to do a rotation(3). Additionally, when we have a black parent node with two red nodes, we need to flip the colors(4). Flipping the colors can cause double reds and require more rotations(4). If we do this correctly, height imbalances shouldn't happen (the first condition), and the red Black Algorithm should prevent such an imbalance. We have specific algorithms for insertion behaviors for 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 processes to run.⁵

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. In order to make sure they didn't lose one of the necessary conditional recipes that an rbtree would use, they diagrammed 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);
            }
        }
    }
}

```

⁵ See Love, Rober: Chapter 4

```

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

```

struct automobile{
    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 no concept of “this”.* **So how does rbnodes and sched_entity's know which instances of tasks that they are associated with?** If you are pulling the left most rbnode, you need the associated task to put into run time.

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);6
}

```

This function takes an argument of struct cfs_rq and returns a pointer to a struct sched_entity. How does it acquire that data structure from an operation that seems to take as its only unique information and rb_node. rb_nodes only contain information about their color, who's its parents node is, and pointers to its left and right children nodes in the rbtree. Furthermore, the second argument is a data type. **There is no C function that 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 Macro 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.⁷ 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, and a couple of 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 Macros are defined as follows

⁶ 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.

⁷ Oram, Andy and Wilson, Greg: Beautiful Code 2007 O'Reilly pg 271


```
#define identifier (x1 , x2 , ... , xn ) replacement list 8
```

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)
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 Kings⁹

```
#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 mitigate 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

⁸King, K.N.: C Programming a Modern Approach WW Norton and Company 1996: pg 279

⁹ As above 286

located on the [GNU website](https://gcc.gnu.org/onlinedocs/gcc/Typeof.html)¹⁰

6.6 Referring to a Type with typedef

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))
```

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 the object type “*type*” is, the preprocessor can evaluate and substitute for it within the parenthesis, to determine its datatype and is then substituted it 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.

¹⁰ <https://gcc.gnu.org/onlinedocs/gcc/Typeof.html>

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, until now, unknown instance of parent struct of **type**.

Offsetof, however, is a standard ANSI C macro in `stddef.h` and has a standard manpage on most GNU/Linux distributions. An excerpt 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 then 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.¹¹ 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` from a simple `rb_nodes`, and that provides a considerable performance boost over having to move tasks directly inserted into run queues and wait queues or to manipulate them in an RBTree.

Now that we can find our `sched_entity` from a related `rb_nod`, we can now examine the function that finds a task's 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 `__enqueue_entity()`. This is the function that ties together many of the concepts on the CFS that we have thus far described:

¹¹ Above: Oram and Wilson: Pg 268

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

```
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); //SEE DEFINITION ABOVE - Ruben
        /*
         * 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 (via `rb_entry`) to get access to the `sched_entity`. The `sched_entity` has the `vruntime` data within it. `Vruntime` contains the adjusted value of time that a task has accumulated and more importantly for understanding our code, it is the key value of the rbtrees. 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.¹² The operating system is preemptive and 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 always tries to run the task with the smallest such `vruntime` value. Values are affected by nice priorities which can weigh down a `vruntime` value by making them larger or smaller with a coefficient.¹³

The run queue is held in the struct `rq`. `rq->cfs.min_vruntime` is a monotonic constantly increasing value tracking the smallest `vruntime` among all tasks in the queue. It is a measure of the complete work the system has done from the time the system was turned on. That value helps place new tasks as far left as possible on the RBTree.¹⁴

Scheduling begins with hardware clock which is translate into software interrupt probes 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 a process is spent on the cpu. That accounting is stored in `p->se.vruntime`. When `vruntime` rises above the left most process on the `rbtree`, plus a standardized little bit more time, then the processes are swapped out. There is a granularity fudge factor built into the algorithm prevent thrashing of processes.¹⁵ 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

12 above: Love: Chapter 4

13 CFS Source Code Documention

14 *ibid.*

15 *ibid.*

are part of the gauntlet prior to that step. The kernel calls for `__schedule()` and their are ancillary functions located in `../linux-3.xx.x/kernel/sched/core.c` . These functions prioritized class selection using `pick_next_task(struct rq)` .

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

`struct task_struct * (*pick_next_task) (struct rq *rq, struct task_struct *prev)` 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 run time 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 acts as a place holder and descriptor for the entire queue. This is struct `cfs_rq` and it is defined in

`..linux-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;
#ifdef 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
#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 */
#endif

#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 if it is not the destiny for a task to be put right back into the rbtree, then it has to be placed within the other queues. 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, A) configure, B) make/compile install and C) add to the boot loaders of your system. Years going by, this process has been simpler. Today there are numerous 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**. However, Some of the details in the Readme file have been outdated and distributions have increasingly relied on customized installation tools for kernel development. Boot loaders have changed dramatically from lilo to grub to grub2 and now is a series of tools to help manipulate to the UEFI system and secure boot. Configuration files have also become very complicated. Every piece of hardware needs new modules and has new options. Configuring the config file can get to be 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 advantage 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 kernel binaries are 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
```

initramfs is a commonly used to work with tmpfs for booting. Partitioning of file systems and disks now need special attention for kernel developers. 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-*16
```

The kernel compile scripts will write out files with the name vmlinux, which doesn't match the grub2 requirement. Compressed versions that are most likely used for booting are under the arch subdirectory of your system under the name bzlinux. 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 have 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 excerpt, for example, of 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
2
3
4
```

```
CONFIG_EFI=y
CONFIG_EFI_STUB=y
...
CONFIG_FB_EFI=y
```

¹⁶ 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 "$machine" in
  xi?86 | xx86_64)
    list=`for i in /boot/vmlinuz-* /vmlinuz-* /boot/kernel-* ; do
      if grub_file_is_not_garbage "$i" ; then echo -n "$i " ; fi
    done` ;;
  *)
    list=`for i in /boot/vmlinuz-* /boot/vmlinux-* /vmlinuz-* /vmlinux-* /boot/kernel-* ; do
      if grub_file_is_not_garbage "$i" ; then echo -n "$i " ; fi
    done` ;;
esac
```

5
6
7
8
9
10

```
...
CONFIG_CMDLINE_BOOL=y
CONFIG_CMDLINE="root=..."
...
CONFIG_BLK_DEV_INITRD=y
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.

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 efutil 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 efutools git tree is [here](#).

The efutools [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
```

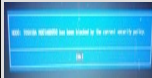
For more details about the key creation (and to see where I copied these command lines from), see [James’s post about owning your own Windows 8 platform](#).

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.¹⁷

So as you can see, kernel compiling, set up and booting has evolved considerably over the past few years and has become quite complex. And for this reason, I will include in this paper an outline of the method for kernel installation that I used. It is not complete, but it should allow for anyone to copy my methods with and with little bit of work do so yourself. I will also outline some of the pitfalls.

Virtualization can make kernel development easier. On GNU systems you have several options. Here I chose to use Oracle's Virtual Box just because of familiarity. But there is also others. Vmware's virtualization system has been available for many years. And most interesting to me is the development of KVM, which is a free software implementation of virtualization that utilizes the `kvm.ko` loadable kernel module.^{18 19} Virtualization requires both host and client components, also known as guest and host. There can be considerable integration of mouse and keyboard controls between the two environments (thank god for the sake of my cut and pasting for this paper). 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.

Although the methodology that we used was to use oracle's virtualbox, 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.

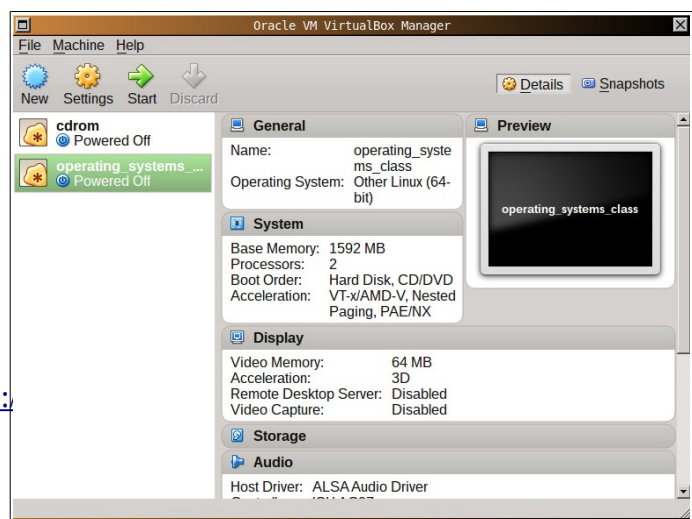
```
sudo modprobe vboxdrv
```

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`.²⁰ This was chosen for a number of reasons but largely due to its easy of

¹⁷ Kroah-Hartman, Greg from his website: <http://www.kernel.org/doc/Documentation/serial-console.txt>
¹⁸ http://www.linux-kvm.org/page/Main_Page
¹⁹ <http://virt-tools.org/learning/index.html>



use, minimal resource requirements and access to openrc as the system init, 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.

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 the kernel source code to our home directory. I want to put the kernel code into /usr/src which is the traditional location for kernel source code. 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. Compiling should be done as a regular user, and not root. It don't often say this anywhere, but it is true. Always compile as a a regular user so that you have a completely and functioning environment that you are working in. When you compile as root, things don't work.

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. Use these commands:

```

ruben@host:~/Downloads/ $ unxz linux-3.19.5.tar.xz
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 linux-3.19.5
[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. If it is not there, makeis with

the `ln -S` command.

At this point `cd` into the proper directory and run **make mrproper** just to clean out dependencies and permission 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, choosing through option after option in software and hardware, an option for every occasion. To say it is a challenge is an understatement. I tried it and it took several hours and it didn't produce a working config file. So 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 Lenovo Think Center, such as I am, 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 lucky, you develop a relationship with many individuals on the internet who have real knowledge.*** It takes time to weed

Sample of usenet Kernel conversation

```

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.

--

```


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 open 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. ²¹

```
[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*. When finished we run **make**, and then **make modules_install**

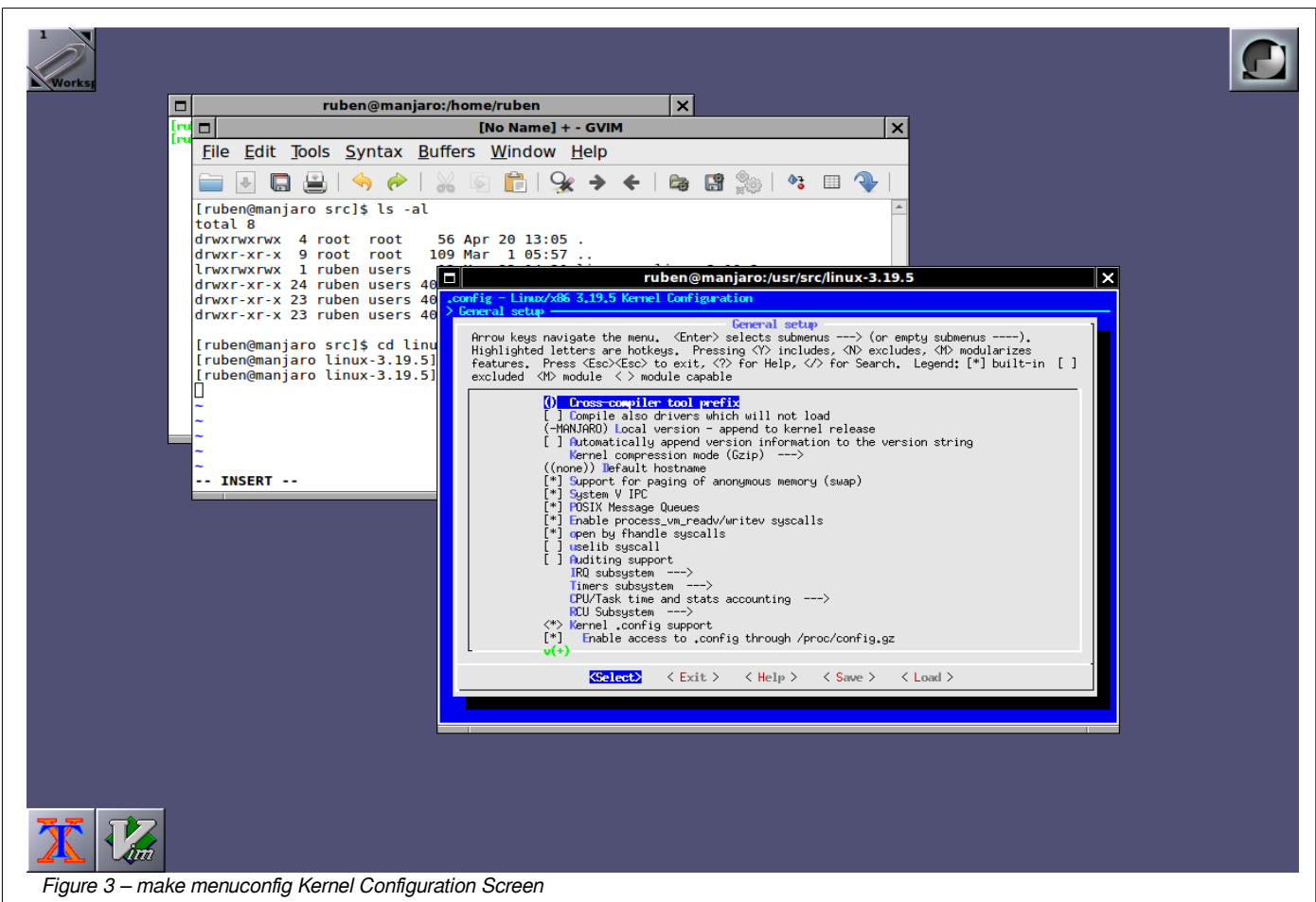


Figure 3 – make menuconfig Kernel Configuration Screen

²¹ 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.

Setting Your customized Kernel to Boot

When all you compiling is finished, your /usr/src directory will have a new freshly minted GNU/Linux kernel

```
[ruben@manjaro linux-3.19.5]$ ls -l|grep "vmlinuz"
-rwxr-xr-x  1 ruben users 18112680 Apr 20 18:23 vmlinuz
-rw-r--r--  1 ruben users 22180056 Apr 20 18:23 vmlinuz.o
```

Unfortunately, the next few steps are the ones that have changed dramatically over the few years and it appears that they will continue to change, and become more distribution specific. This might be one of the reasons why the instructions within the README file is not up to date for the placement of the location of the kernel, the configuration of the boot loader, or the methods for creating ramdisk or tmpfs boot environments. For whatever the reasons, the documentation within the GNU/Linux kernel is no longer up to date for actually installing your kernel. So we cheat and we steal the shell scripts from Manjaro that they use to make new kernels and customize their use for our own purposes. Such are the gifts of Free Software.

A sample build script from Manjaro Linux can be seen here from the GITS repository:

<https://github.com/manjaro/packages-core/blob/master/linux319/PKGBUILD>

and <https://github.com/manjaro/packages-core/blob/master/linux319/linux319.install>

The later link has important instructions that help us create the needed files, grub2 setup and, initiation ram or tmpfs disks. First, lets see how grub is set up on a live system.

Grub2 is the most popular GNU/Linux boot manager and can be used in standard bios and efi environments, although its role is efi can be reduced to the secondary boot loader since the hardware usually comes with a primary boot loaded and efi shell. It is configured through a set of config files under the /etc/ sub-directory

```
[ruben@stat13 etc]$ ls grub*
grub-customizer:
grub.cfg  viewOptions.cfg

grub.d:
00_header  20_linux_xen  40_custom  60_memtest86+
10_linux   30_os-prober  41_custom  README
[ruben@stat13 etc]$ ls /etc/default/|grep grub
/etc/default/grub
```

In general, these files should not be hand hacked, although many experienced kernel hackers do just that. Grub configuration is automatically updated by the `update-grub` command as we will see shortly.

It is, however, useful to take a peek at those files and see what they do. Learning how grub2 works in detail is a useful skill but, but outside of the scope of this paper.

In our live `/boot` directory is the current kernel binary files and tmpfs disks that we use of booting. In theory, we can boot without the tmpfs disks. Distributions are bias towards booting first with the tmpfs disks because it gives them flexibility. Our `/boot` directory looks as follows:

```
[ruben@manjaro boot]$ ls -al /boot
total 85100
drwxr-xr-x  8 root root    4096 Apr  6 22:32 .
drwxr-xr-x 17 root root    4096 Apr 21 05:04 ..
drwxr-xr-x  2 root root    4096 Dec 31 1969 efi
drwxr-xr-x  3 root root    4096 Mar 21 01:44 EFI
drwxr-xr-x  6 root root    4096 Apr  6 22:32 grub
-rw-r--r--  1 root root 18575110 Apr  6 22:31 initramfs-318-x86_64-fallback.img
-rw-r--r--  1 root root  3387082 Apr  6 22:31 initramfs-318-x86_64.img
-rw-r--r--  1 root root 19104682 Mar 22 20:48 initramfs-3192-RS-x86_64-fallback.img
-rw-r--r--  1 root root  4052743 Mar 22 20:47 initramfs-3192-RS-x86_64.img
-rw-r--r--  1 root root 18648066 Apr  6 22:32 initramfs-319-x86_64-fallback.img
-rw-r--r--  1 root root  3388714 Apr  6 22:31 initramfs-319-x86_64.img
-rw-r--r--  1 root root   663040 Feb  9 10:42 intel-ucode.img
```

```

-rw-r--r-- 1 root root      22 Apr  4 17:52 linux318-x86_64.kver
-rw-r--r-- 1 root root      21 Mar 26 15:19 linux319-x86_64.kver
drwx----- 2 root root    16384 Mar 22 11:30 lost+found
drwxr-xr-x 2 root root     4096 Oct  6 2013 memtest86+
drwxr-xr-x 2 root root     4096 Apr  6 22:49 syslinux
-rw-r--r-- 1 root root   2667040 Mar 22 20:08 System.map
-rw-r--r-- 1 root root   4116496 Mar 22 20:08 vmlinuz
-rw-r--r-- 1 root root   4087328 Apr  4 17:52 vmlinuz-318-x86_64
-rw-r--r-- 1 root root   4116496 Mar 22 20:38 vmlinuz-3192-RS-x86_64
-rw-r--r-- 1 root root   4148048 Mar 26 15:19 vmlinuz-319-x86_64

```

This is a testbed virtualbox and so we see many kernel versions and a customized kernel already in the boot tree. The boot loader, grub, makes each kernel available through a menu when you boot. The initramfs are the boot images, and the vmlinuz are the corresponding kernels.

When we ran `make module_install`, the make file probed our system and automatically created the directory `/usr/lib/modules/3.19.5-MANJARO` and placed the modules under that directory. we now need to build the modules database for module loading with the `depmod` command as seen in the manjaro configuration file.

sudo depmod

This command allows for the loadable modules to be found, not only at boot, but throughout the operations of your computer. Later, if you want to boot a kernel module you can just run `modprobe` and it will find, run and install the module.

The next step is to create a tmpfs image disk. The tmpfs is in an archive format of cpio and the file `/etc/mkinitcpio.conf` instructs the system how to use the initramfs when it boots. There is a subdirectory `/etc/mkinitcpio.d` and it has **preset** files for each kernel that is built and has a initramfs boot image. When we create the image this preset file will allow for us to make a proper version according to the specifics of our system.

We are going to steal the Manjaro `/etc/mkinitcpio.conf` file which looks like this:

```
# mkinitcpio preset file for the 'linux' package

ALL_config="/etc/mkinitcpio.conf"
ALL_kver="/boot/vmlinuz-3.X-X-manjaro"

PRESETS=('default' 'fallback')

#default_config="/etc/mkinitcpio.conf"
default_image="/boot/initramfs-3.X-X-manjaro.img"
#default_options=""

#fallback_config="/etc/mkinitcpio.conf"
fallback_image="/boot/initramfs-3.X-X-manjaro-fallback.img"
fallback_options="-S autodetect"
```

and we can edit it a little bit for our own purposes.

```
# mkinitcpio preset file for the 'linux' package

ALL_config="/etc/mkinitcpio.conf"
ALL_kver="/boot/vmlinuz-319-liu_test"

PRESETS=('default' 'fallback')

#default_config="/etc/mkinitcpio.conf"
default_image="/boot/initramfs-319-liu_test.img"
#default_options=""

#fallback_config="/etc/mkinitcpio.conf"
fallback_image="/boot/initramfs-319-liu_test-fallback.img"
fallback_options="-S autodetect"
```

Now we copy the new kernel image to the /boot directory. The compressed kernel image is buried in the source code tree and is called bzImage. We can use the find command to discover it and then copy it to the /boot directory as follows, and create the initramfs disk. Run depmod on the target kernel first. I know this is twice but better to do it now with the source directory as outlined. Our source created modules under /lib/modules/3.19.5-MANJARO/

```
$ find /usr/src/linux-3.19.5/ -name "bzImage*" -print
/usr/src/linux-3.19.5/arch/x86/boot/bzImage
/usr/src/linux-3.19.5/arch/x86_64/boot/bzImage

[ruben@manjaro linux]$ sudo depmod 3.19.5-MANJARO
[ruben@manjaro linux]$ sudo mkinitcpio -p linux319_liutest
==> Building image from preset: /etc/mkinitcpio.d/linux319_liutest.preset: 'default'
-> -k /boot/vmlinuz-319_liu_test -c /etc/mkinitcpio.conf -g /boot/initramfs-319_liu_test.img
```

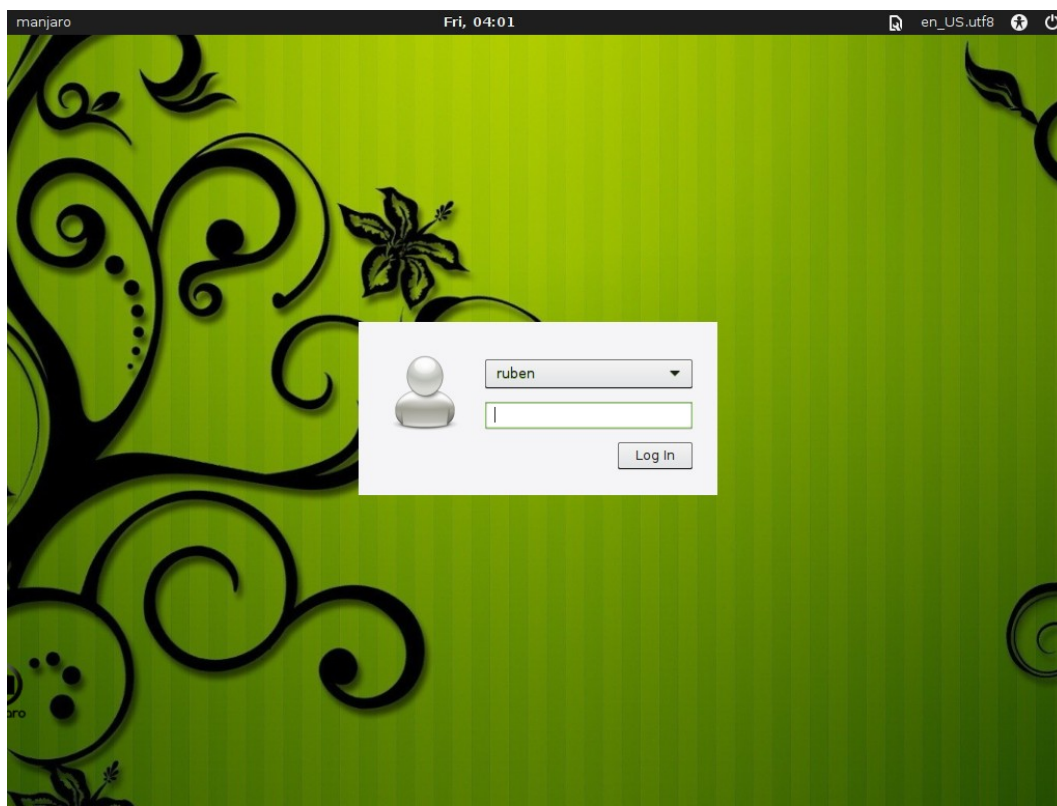
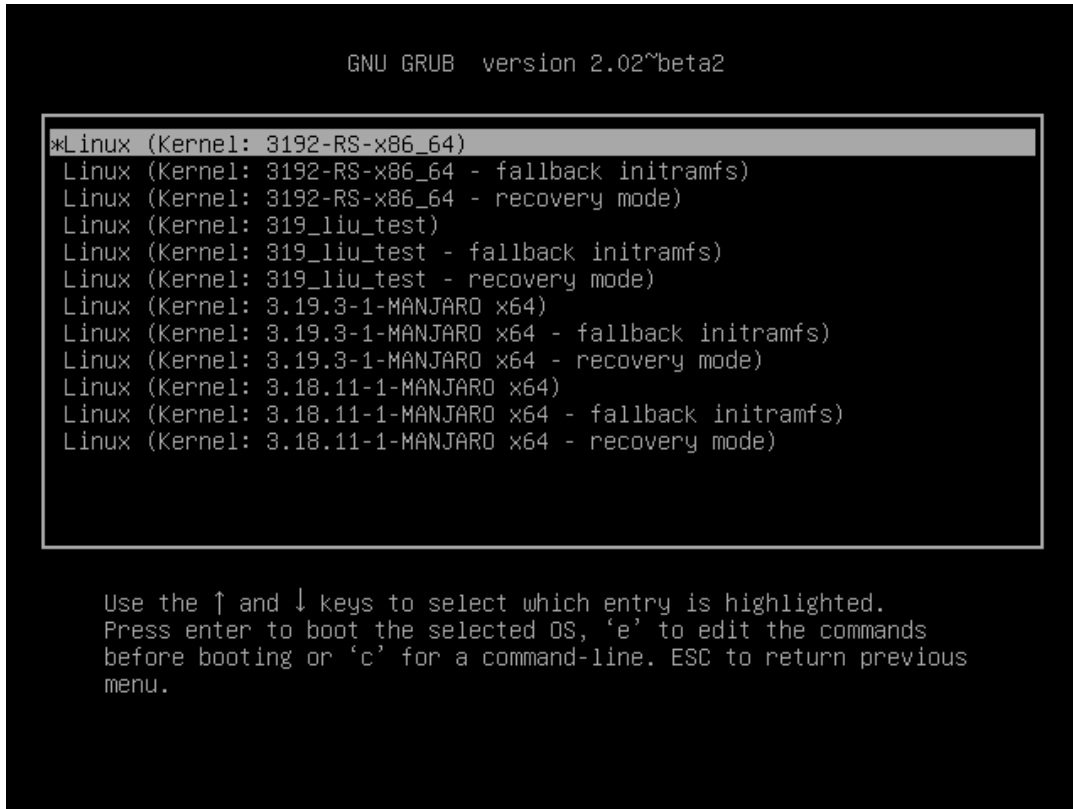
```
==> Starting build: 3.19.5-MANJARO
-> Running build hook: [base]
-> Running build hook: [udev]
-> Running build hook: [autodetect]
-> Running build hook: [modconf]
-> Running build hook: [block]
-> Running build hook: [filesystems]
-> Running build hook: [keyboard]
-> Running build hook: [keymap]
-> Running build hook: [fsck]
-> Running build hook: [usr]
-> Running build hook: [shutdown]
==> Generating module dependencies
==> Creating gzip-compressed initcpio image: /boot/initramfs-319_liu_test.img
==> Image generation successful
==> Building image from preset: /etc/mkinitcpio.d/linux319_liutest.preset: 'fallback'
-> -k /boot/vmlinuz-319_liu_test -c /etc/mkinitcpio.conf -g /boot/initramfs-319_liu_test-fallback.img
-S autodetect
==> Starting build: 3.19.5-MANJARO
-> Running build hook: [base]
-> Running build hook: [udev]
-> Running build hook: [modconf]
-> Running build hook: [block]
==> WARNING: Possibly missing firmware for module: aic94xx
==> WARNING: Possibly missing firmware for module: wd719x
-> Running build hook: [filesystems]
-> Running build hook: [keyboard]
-> Running build hook: [keymap]
-> Running build hook: [fsck]
-> Running build hook: [usr]
-> Running build hook: [shutdown]
==> Generating module dependencies
==> Creating gzip-compressed initcpio image: /boot/initramfs-319_liu_test-fallback.img
==> Image generation successful
```

Now we can copy the system map to the boot directory. It is located in the root source directory,

/usr/src/linux.3.19.5/ in our case. I make a backup of the current system map first and then copy. Then run `update_grub`.

```
[ruben@manjaro linux-3.19.5]$ sudo cp /boot/System.map /boot/System.map.bak
[ruben@manjaro linux-3.19.5]$ sudo cp ./System.map /boot/
[ruben@manjaro linux-3.19.5]$ sudo update
update-ca-trust          update-desktop-database  update-mime-database    update-pciids
updatedb                update-grub              update-patterndb        update-smart-drivedb
[ruben@manjaro linux-3.19.5]$ sudo update-grub
Generating grub configuration file ...
Found Intel Microcode image
Found linux image: /boot/vmlinuz-3192-RS-x86_64
Found initrd image: /boot/initramfs-3192-RS-x86_64.img
Found initrd fallback image: /boot/initramfs-3192-RS-x86_64-fallback.img
Found linux image: /boot/vmlinuz-319_liu_test
Found initrd image: /boot/initramfs-319_liu_test.img
Found initrd fallback image: /boot/initramfs-319_liu_test-fallback.img
Found linux image: /boot/vmlinuz-319-x86_64
Found initrd image: /boot/initramfs-319-x86_64.img
Found initrd fallback image: /boot/initramfs-319-x86_64-fallback.img
Found linux image: /boot/vmlinuz-318-x86_64
Found initrd image: /boot/initramfs-318-x86_64.img
Found initrd fallback image: /boot/initramfs-318-x86_64-fallback.img
  /run/lvm/lvmetad.socket: connect failed: No such file or directory
  WARNING: Failed to connect to lvmetad. Falling back to internal scanning.
Found memtest86+ image: /boot/memtest86+/memtest.bin
done
```

You can ignore the last warning from grub inside of a virtualbox. It's a false alarm. But you are now ready to reboot with the new Kernel.



Testing Exploring Your New Kernel and Scheduler

Now that we have a customized hand compiled kernel installed and booted, we can now investigate swapping out different schedulers and test them. Ideally, one would have a standardized test where the components and the conditions of the running test system can be controlled. I don't have that. And furthermore, one should have a tool to profile performance benchmarks of different schedulers and run such an analysis through R and GNUPLOT in order to make a definitive report. There was a wonderful tool called LinSched which was written by John M. Calandrino, Dan P. Baumberger, Tong Li, Jessica C. Young, and Scott Hahn, however it seems to be no longer supported. At this time, I have not yet found a means to replace its functionality, which is a blot on my abilities, since I'm not able to write such software, and I couldn't find any other software available to test the Linux Scheduler.

However, I did write up software which I think can demonstrate some the schedulers ability and is simple enough in design to enhance it over time. Again, any deficiencies in this software is due to my own limitations. And perhaps with time this stress program can be improved. Initially, I attempted to do a simple recursion algorithm with pthreads. I found that when I ran it with large numbers it would segfault, which ideally it wouldn't do. It attempts to count the number of threads created and currently running and to reboot threads as they come home and rejoin, which in theory should allow it the program to run indefinitely. I didn't quite succeed in this. The counter, for one thing, needs to be a critical area, but doing so makes all the threads wait, upsetting the point of the program. I need to give this more thought as well. Here is the code as it is.

```
1 #ifndef STRESS
2 #define STRESS
3 #define MAX 1000
4 #include <stdint.h>
5 #include <inttypes.h>
6 void * fib(void * x);
```

50

```
7
8 int64_t counter;
9
10 void * fib(void * x)
11 {
12     fflush(stdout);
13     printf("New Thread ...");
14     counter++;
15     printf("counter => %" PRId64 "\n", counter);
16     int64_t a;
17     int64_t b;
18     int64_t * x_one = &a;
19     int64_t * x_two = &b;
20     int64_t sum = *(int64_t *)x;
21     if (counter > MAX){
22         printf("No more Threads, we are over MAX\n");
23         //counter--;
24         return 0;
25     }
26
27     *x_one = (sum + 1);
28     *x_two = (sum + 2);
29     printf("Sum => %" PRId64 "\n", sum);
30     printf("x_one => %" PRId64 "\n", *x_one);
31     printf("x_two => %" PRId64 "\n", *x_two);
32     printf("MAX => %d\n", MAX);
33
34     pthread_attr_t attr;
35     pthread_attr_init(&attr);
36     pthread_t mytd;
37     pthread_t mytd2;
38     pthread_create(&mytd, &attr, fib, x_one);
39     pthread_create(&mytd2, &attr, fib, x_two);
40     pthread_join(mytd, NULL);
41     printf("Thread A rejoined\n");
42     //counter--;
43     printf("counter => %" PRId64 "\n", counter);
44     pthread_join(mytd2, NULL);
45     printf("Thread B rejoined\n");
46     counter--;
47     printf("counter => %" PRId64 "\n", counter);
48     return 0;
49
50
51 // return (x + fib(x+1) + fib(x+2) );
52 }
53
54
55 #endif
```

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "stress.h"
5
```

51

```
6
7 int main(int argc, char * argv[])
8 {
9     int64_t root = 0;
10    counter = 0;
11    int64_t * start = &root;
12    fib( (void *) start);
13    return 1;
14 }
15
```

This code will rapidly create a large number of threads, in a geometric progression. In theory, you should be able to maintain a number of threads, say 1000 threads, popping in and out, but the code needs more work. As it is, it produces twice as many threads as MAX because of a race condition, until it settles down to the MAX count. There may also be memory issues within the pthread allocations which are set up in each recursion. pthread_attr_t attr is generated over and over and then passed forward, which is a large consumption of memory, most likely stored on the stack, although I'm not certain.

A simple use of this program is as follows:

```
[ruben@stat13 scheduler_stress_test]$ ./stress.exe >out&
[ruben@stat13 scheduler_stress_test]$ ps -eLf|grep stress
ruben    7458  4789 20668  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20669  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20670  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20671  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20672  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20674  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20675  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20676  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20677  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20678  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20679  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20680  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20681  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20682  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben    7458  4789 20683  0 9641 03:50 pts/2    00:00:00 ./stress.exe
```

```

ruben  7458  4789 20684  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben  7458  4789 20685  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben  7458  4789 20686  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben  7458  4789 20687  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben  7458  4789 20688  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben  7458  4789 20689  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben  7458  4789 20690  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben  7458  4789 20692  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben  7458  4789 20693  0 9641 03:50 pts/2    00:00:00 ./stress.exe
ruben  16549 4789 16549  0  1 03:50 pts/2    00:00:00 grep stress

```

I can track down a memory problem which shows up in the Kernel reporting mechanism as follows:

```

$ dmesg
[95924.952889] stress.exe[5206]: segfault at 7f2b632d49d0 ip 00007f4656b62410 sp
00007f38784fae88 error 4 in libpthread-2.20.so[7f4656b5a000+17000]

```

The next step would be to find the corruption with a standard tool like valgrind, unfortunately, at the moment that would require a major update to the working system that can't be done at this time because of system obligations. A further review of the program is certainly worthwhile.²²

Changing Scheduler

The second most common scheduler used in the GNU/linux kernel is called the BFS scheduler written by Con Kolvas. For one thing, it is the default scheduler used for the Manjaro distribution as can be seen in the source code for the default manjaro kernel. It is also being used experimentally for android and for Cyanogen, the free software version of Android.²³

Patches for the BFS scheduler are available from the Kolvas's website at <http://ck.kolivas.org/patches/bfs/>. We are using a derivative of the 3.19.5 kernel on our virtual box and so we will download the patch <http://ck.kolivas.org/patches/bfs/3.0/3.19/3.19-sched-bfs-461.patch>. The patch file can be used to patch the kernel and form an upgrade. Patches are performed on the kernel using the patch command: `man patch`

²² <http://marc.info/?l=debian-devel&m=132009480129145&w=2> for details on error 4

²³ <http://forum.cyanogenmod.org/topic/18575-cfs-vs-bfs-kernel-scheduler/>

PATCH(1)

General Commands Manual

PATCH(1)

NAME

patch - apply a diff file to an original

SYNOPSIS

patch [options] [originalfile [patchfile]]

but usually just

patch -pnum <patchfile

DESCRIPTION

patch takes a patch file patchfile containing a difference listing produced by the diff program and applies those differences to one or more original files, producing patched versions. Normally the patched versions are put in place of the originals. Backups can be made; see the -b or --backup option. The names of the files to be patched are usually taken from the patch file, but if there's just one file to be patched it can be specified on the command line as originalfile.

Upon startup, patch attempts to determine the type of the diff listing, unless overruled by a -c (--context), -e (--ed), -n (--normal), or -u (--unified) option. Context diffs (old-style, new-style, and unified) and normal diffs are applied by the patch program itself, while ed diffs are simply fed to the ed(1) editor via a pipe.

patch tries to skip any leading garbage, apply the diff, and then skip any trailing garbage. Thus you could feed an article or message containing a diff listing to patch, and it should work. If the entire diff is indented by a consistent amount, if lines end in CRLF, or if a diff is encapsulated one or more times by prepending "- " to lines starting with "- " as specified by Internet RFC 934, this is taken into account. After removing indenting or encapsulation, lines beginning with # are ignored, as they are considered to be comments.

Specifically, patching the kernel is documented in the Linux Documentation Project under Kernel Howto's²⁴

6.1 Applying a patch

Incremental upgrades of the kernel are distributed as patches. For example, if you have version 1.1.45, and you notice that there's a 'patch46.gz' out there for it, it means you can upgrade to version 1.1.46 through application of the patch. You might want to make a backup of the source tree first ('make clean' and then 'cd /usr/src; tar zcvf old-tree.tar.gz linux' will make a compressed tar archive for you.).

So, continuing with the example above, let's suppose that you have 'patch46.gz' in /usr/src. cd to /usr/src and do a 'zcat patch46.gz | patch -p0' (or 'patch -p0 < patch46' if the patch isn't compressed). You'll see things whizz by (or flutter by, if your system is that slow) telling you that it is trying to apply hunks, and whether it succeeds or not. Usually, this action goes by too quickly for you to read, and you're not too sure whether it worked or not, so you might want to use the -s flag to patch, which tells patch to only report error messages (you don't get as much of the 'hey, my computer is actually doing something for a change!' feeling, but you may prefer this..). To look for parts which might not have gone smoothly, cd to /usr/src/linux and look for files with a .rej extension. Some versions of patch (older versions which may have been compiled with on an inferior filesystem) leave the rejects with a # extension. You can use 'find' to look for you;

²⁴ <http://linuxdocs.org/HOWTOs/Kernel-HOWTO-6.html>

```
find . -name '*.rej' -print
```

prints all files who live in the current directory or any subdirectories with a .rej extension to the standard output. If everything went right, do a `make clean`, `config`, and `dep` as described in sections 3 and 4.

Before we do that, however, let us look at what is in the patch, because after all, the ability to just switch a central component of an operating system, such as what we are looking at, is really quite remarkable, and is one of the most powerful aspects of Free Software, and GNU/Linux specifically. As one becomes more proficient with their coding skills and administration skills, they can increasingly participate in development, and even write their own system parts.

The first part of the patch opens with this line:

```
---
Documentation/scheduler/sched-BFS.txt      | 347 +
Documentation/sysctl/kernel.txt            | 26
arch/powerpc/platforms/cell/spufs/sched.c | 5
arch/x86/Kconfig                           | 22
drivers/cpufreq/cpufreq.c                  | 7
drivers/cpufreq/cpufreq_conservative.c    | 4
drivers/cpufreq/cpufreq_ondemand.c        | 4
drivers/cpufreq/intel_pstate.c             | 9
fs/proc/base.c                             | 2
include/linux/init_task.h                  | 66
include/linux/ioprio.h                     | 2
include/linux/jiffies.h                    | 2
include/linux/sched.h                      | 89
include/linux/sched/prio.h                 | 12
include/uapi/linux/sched.h                 | 9
init/Kconfig                               | 57
init/main.c                                | 3
kernel/delayacct.c                         | 2
kernel/exit.c                              | 2
kernel/sched/Makefile                      | 11
kernel/sched/bfs.c                         | 7413 +++++
kernel/sched/bfs_sched.h                   | 161
kernel/sched/idle.c                        | 4
kernel/sched/stats.c                       | 4
kernel/stop_machine.c                     | 3
kernel/sysctl.c                            | 31
kernel/time/Kconfig                        | 2
kernel/time/posix-cpu-timers.c            | 10
lib/Kconfig.debug                          | 2
29 files changed, 8235 insertions(+), 76 deletions(-)
```

This is a summary of the patch files findings when it was created with the diff command. Notice that the major alterations in the kernel will take place with the addition of kernel/sched/bfs.c

and kernel/sched/bfs_scheule.h, although alterations in source code is taking place all around the kernel

code including the drivers for the CPU, `init_task`, and within the header files of the core linux source `include/linux/init_task.h` etc.

Lets look at the patch on `init_task.h`. + marks mean to stitch that code into the tree, and – removes code.

```

Index: linux-3.19-ck1/include/linux/init_task.h
=====
--- linux-3.19-ck1.orig/include/linux/init_task.h      2015-02-27 17:03:41.668631704 +1100
+++ linux-3.19-ck1/include/linux/init_task.h      2015-02-27 17:03:41.661631704 +1100
@@ -156,8 +156,6 @@ extern struct task_group root_task_group
 # define INIT_VTIME(tsk)
 #endif

-#define INIT_TASK_COMM "swapper"
-
-#ifdef CONFIG_RT_MUTEXES
 # define INIT_RT_MUTEXES(tsk) \
 .pi_waiters = RB_ROOT, \
@@ -179,6 +177,68 @@ extern struct task_group root_task_group
 * INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x1ffffff (=2MB)
 */
+#ifdef CONFIG_SCHED_BFS
+#define INIT_TASK_COMM "BFS"
+#define INIT_TASK(tsk) \
+{ \
+ .state = 0, \
+ .stack = &init_thread_info, \
+ .usage = ATOMIC_INIT(2), \
+ .flags = PF_KTHREAD, \
+ .prio = NORMAL_PRIO, \
+ .static_prio = MAX_PRIO-20, \
+ .normal_prio = NORMAL_PRIO, \
+ .deadline = 0, \
+ .policy = SCHED_NORMAL, \
+ .cpus_allowed = CPU_MASK_ALL, \
+ .mm = NULL, \
+ .active_mm = &init_mm, \
+ .run_list = LIST_HEAD_INIT(tsk.run_list), \
+ .time_slice = HZ, \
+ .tasks = LIST_HEAD_INIT(tsk.tasks), \
+ INIT_PUSHABLE_TASKS(tsk) \
+ .ptraced = LIST_HEAD_INIT(tsk.ptraced), \
+ .ptrace_entry = LIST_HEAD_INIT(tsk.ptrace_entry), \
+ .real_parent = &tsk, \
+ .parent = &tsk, \
+ .children = LIST_HEAD_INIT(tsk.children), \
+ .sibling = LIST_HEAD_INIT(tsk.sibling), \
+ .group_leader = &tsk, \
+ RCU_POINTER_INITIALIZER(real_cred, &init_cred), \
+ RCU_POINTER_INITIALIZER(cred, &init_cred), \
+ .comm = INIT_TASK_COMM, \
+ .thread = INIT_THREAD, \
+ .fs = &init_fs, \
+ .files = &init_files, \
+ .signal = &init_signals, \
+ .sighand = &init_sighand, \
+ .nsproxy = &init_nsproxy, \
+ .pending = { \
+ .list = LIST_HEAD_INIT(tsk.pending.list), \
+ .signal = {{0}}, \
+ .blocked = {{0}}, \
+ .alloc_lock = __SPIN_LOCK_UNLOCKED(tsk.alloc_lock), \
+ .journal_info = NULL, \
+ .cpu_timers = INIT_CPU_TIMERS(tsk.cpu_timers), \
+ .pi_lock = __RAW_SPIN_LOCK_UNLOCKED(tsk.pi_lock), \
+ .timer_slack_ns = 50000, /* 50 usec default slack */ \
+ .pids = { \
+ [PIDTYPE_PID] = INIT_PID_LINK(PIDTYPE_PID), \
+ [PIDTYPE_PGID] = INIT_PID_LINK(PIDTYPE_PGID), \
+ [PIDTYPE_SID] = INIT_PID_LINK(PIDTYPE_SID), \
+ }, \
+ .thread_group = LIST_HEAD_INIT(tsk.thread_group), \
+ .thread_node = LIST_HEAD_INIT(init_signals.thread_head), \
+ INIT_IDS \
+ INIT_PERF_EVENTS(tsk) \
+ INIT_TRACE_IRQFLAGS \
+ INIT_LOCKDEP \
+ INIT_FTRACE_GRAPH \
+ INIT_TRACE_RECURSION \
+ INIT_TASK_RCU_PREEMPT(tsk) \
+}
+#else /* CONFIG_SCHED_BFS */

```

```

#define INIT_TASK_COMM "swapper"
#define INIT_TASK(tsk) \
{
    .state = 0, \
    @@ -248,7 +308,7 @@ extern struct task_group root_task_group \
    INIT_VTIME(tsk) \
    INIT_NUMA_BALANCING(tsk) \
}
-
#endif /* CONFIG_SCHED_BFS */

#define INIT_CPU_TIMERS(cpu_timers) \
{

```

Here is a Key section of the scheduler code that is being inserted into the Kernel source tree

```

+/*
+ * schedule() is the main scheduler function.
+ *
+ * The main means of driving the scheduler and thus entering this function are:
+ *
+ * 1. Explicit blocking: mutex, semaphore, waitqueue, etc.
+ *
+ * 2. TIF_NEED_RESCHED flag is checked on interrupt and userspace return
+ *    paths. For example, see arch/x86/entry_64.S.
+ *
+ *    To drive preemption between tasks, the scheduler sets the flag in timer
+ *    interrupt handler scheduler_tick().
+ *
+ * 3. Wakeups don't really cause entry into schedule(). They add a
+ *    task to the run-queue and that's it.
+ *
+ *    Now, if the new task added to the run-queue preempts the current
+ *    task, then the wakeup sets TIF_NEED_RESCHED and schedule() gets
+ *    called on the nearest possible occasion:
+ *
+ *    - If the kernel is preemptible (CONFIG_PREEMPT=y):
+ *
+ *    - in syscall or exception context, at the next outmost
+ *    preempt_enable(). (this might be as soon as the wake_up()'s
+ *    spin_unlock(!))
+ *
+ *    - in IRQ context, return from interrupt-handler to
+ *    preemptible context
+ *
+ *    - If the kernel is not preemptible (CONFIG_PREEMPT is not set)
+ *    then at the next:
+ *
+ *    - cond_resched() call
+ *    - explicit schedule() call
+ *    - return from syscall or exception to user-space
+ *    - return from interrupt-handler to user-space
+ */
+asmlinkage __visible void __sched schedule(void)
+{
+    struct task_struct *prev, *next, *idle;
+    unsigned long *switch_count;
+    bool deactivate;
+    struct rq *rq;
+    int cpu;
+
+    +need_resched:
+    preempt_disable();
+    cpu = smp_processor_id();
+    rq = cpu_rq(cpu);
+    rcu_note_context_switch();
+    prev = rq->curr;
+
+    deactivate = false;
+    schedule_debug(prev);
+
+    /*
+     * Make sure that signal_pending_state()->signal_pending() below
+     * can't be reordered with __set_current_state(TASK_INTERRUPTIBLE)
+     * done by the caller to avoid the race with signal_wake_up().
+     */
+    smp_mb__before_spinlock();
+    grq_lock_irq();
+
+    switch_count = &prev->nivcsw;
+    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
+        if (unlikely(signal_pending_state(prev->state, prev))) {
+            prev->state = TASK_RUNNING;
+        } else {
+            deactivate = true;

```



```

+         prev->on_rq = 0;
+
+         /*
+         * If a worker is going to sleep, notify and
+         * ask workqueue whether it wants to wake up a
+         * task to maintain concurrency. If so, wake
+         * up the task.
+         */
+         if (prev->flags & PF_WQ_WORKER) {
+             struct task_struct *to_wakeup;
+
+             to_wakeup = wq_worker_sleeping(prev, cpu);
+             if (to_wakeup) {
+                 /* This shouldn't happen, but does */
+                 if (unlikely(to_wakeup == prev))
+                     deactivate = false;
+                 else
+                     try_to_wake_up_local(to_wakeup);
+             }
+         }
+         switch_count = &prev->nvcsw;
+     }
+ }
+
+ /*
+ * If we are going to sleep and we have plugged IO queued, make
+ * sure to submit it to avoid deadlocks.
+ */
+ if (unlikely(deactivate && blk_needs_flush_plug(prev))) {
+     grq_unlock_irq();
+     preempt_enable_no_resched();
+     blk_schedule_flush_plug(prev);
+     goto need_resched;
+ }
+
+ update_clocks(rq);
+ update_cpu_clock_switch(rq, prev);
+ if (rq->clock - rq->last_tick > HALF_JIFFY_NS)
+     rq->dither = false;
+ else
+     rq->dither = true;
+
+ clear_tsk_need_resched(prev);
+ clear_preempt_need_resched();
+
+ idle = rq->idle;
+ if (idle != prev) {
+     /* Update all the information stored on struct rq */
+     prev->time_slice = rq->rq_time_slice;
+     prev->deadline = rq->rq_deadline;
+     check_deadline(prev);
+     prev->last_ran = rq->clock_task;
+
+     /* Task changed affinity off this CPU */
+     if (likely(!needs_other_cpu(prev, cpu))) {
+         if (!deactivate) {
+             if (!queued_notrunning()) {
+                 /*
+                  * We now know prev is the only thing that is
+                  * awaiting CPU so we can bypass rechecking for
+                  * the earliest deadline task and just run it
+                  * again.
+                  */
+                 set_rq_task(rq, prev);
+                 check_smt_siblings(cpu);
+                 grq_unlock_irq();
+                 goto rerun_prev_unlocked;
+             } else
+                 swap_sticky(rq, cpu, prev);
+         }
+     }
+     return_task(prev, rq, deactivate);
+ }
+
+ if (unlikely(!queued_notrunning())) {
+     /*
+     * This CPU is now truly idle as opposed to when idle is
+     * scheduled as a high priority task in its own right.
+     */
+     next = idle;
+     schedstat_inc(rq, sched_goidle);
+     set_cpuidle_map(cpu);
+ } else {
+     next = earliest_deadline_task(rq, cpu, idle);
+     if (likely(next->prio != PRIO_LIMIT))
+         clear_cpuidle_map(cpu);
+     else
+         set_cpuidle_map(cpu);
+ }
+
+ if (likely(prev != next)) {
+     /*

```

```

+      * Don't reschedule an idle task or deactivated tasks
+      */
+      if (prev != idle && !deactivate)
+          resched_suitable_idle(prev);
+
+      /*
+       * Don't stick tasks when a real time task is going to run as
+       * they may literally get stuck.
+       */
+      if (rt_task(next))
+          unstick_task(rq, prev);
+      set_rq_task(rq, next);
+      if (next != idle)
+          check_smt_siblings(cpu);
+      else
+          wake_smt_siblings(cpu);
+      grq.nr_switches++;
+      prev->on_cpu = false;
+      next->on_cpu = true;
+      rq->curr = next;
+      ++switch_count;
+
+      rq = context_switch(rq, prev, next); /* unlocks the grq */
+      cpu = cpu_of(rq);
+      idle = rq->idle;
+  } else {
+      check_smt_siblings(cpu);
+      grq_unlock_irq();
+  }
+
+rerun_prev_unlocked:
+  sched_preempt_enable_no_resched();
+  if (unlikely(need_resched()))
+      goto need_resched;
+}
+
+EXPORT_SYMBOL(schedule);
+
+#ifdef CONFIG_CONTEXT_TRACKING
+asmlinkage __visible void __sched schedule_user(void)
+{
+    /*
+     * If we come here after a random call to set_need_resched(),
+     * or we have been woken up remotely but the IPI has not yet arrived,
+     * we haven't yet exited the RCU idle mode. Do it here manually until
+     * we find a better solution.
+     *
+     * NB: There are buggy callers of this function. Ideally we
+     * should warn if prev_state != IN_USER, but that will trigger
+     * too frequently to make sense yet.
+     */
+    enum ctx_state prev_state = exception_enter();
+    schedule();
+    exception_exit(prev_state);
+}
+#endif
+
+/**
+ * schedule_preempt_disabled - called with preemption disabled
+ *
+ * Returns with preemption disabled. Note: preempt_count must be 1
+ */
+void __sched schedule_preempt_disabled(void)
+{
+    sched_preempt_enable_no_resched();
+    schedule();
+    preempt_disable();
+}

```

In many regards, the BFS scheduler is MUCH harder to understand and describe than the CFS scheduler. The CFS scheduler has clearly defined and separated data structures and methods to handle scheduling. We can easily identify the Rbtree, macros and class stratification methods that run the scheduler. With BFS, the implementation is very minimized. It has a single run queue which is added to the task entity directly with this code in the patch. Its comments are largely useless to understand what is happening.

```

589 @@ -1278,9 +1276,11 @@ struct task_struct {
590     unsigned int flags; /* per process flags, defined below */

```

```

591 unsigned int ptrace;
592
593 #ifndef CONFIG_SMP
594 #if defined(CONFIG_SMP) || defined(CONFIG_SCHED_BFS)
595 struct llist_node wake_entry;
596 int on_cpu;
597 #endif
598 #ifdef CONFIG_SMP
599 struct task_struct *last_wakee;
600 unsigned long wakee_flips;
601 unsigned long wakee_flip_decay_ts;
602 @@ -1288,12 +1288,29 @@ struct task_struct {
603 int wake_cpu;
604 #endif
605 int on_rq;
606 -
607 int prio, static_prio, normal_prio;
608 unsigned int rt_priority;
609 #ifdef CONFIG_SCHED_BFS
610 + int time_slice;
611 + u64 deadline;
612 + struct list_head run_list;
613 + u64 last_ran;
614 + u64 sched_time; /* sched_clock time spent running */
615 #ifdef CONFIG_SMT_NICE
616 + int smt_bias; /* Policy/nice level bias across smt siblings */
617 #endif
618 #ifdef CONFIG_SMP
619 + bool sticky; /* Soft affined flag */
620 #endif
621 #ifdef CONFIG_HOTPLUG_CPU
622 + bool zerobound; /* Bound to CPU0 for hotplug */
623 #endif

```

Here is `list_head run_list`.

```

+ #ifdef CONFIG_SCHED_BFS
+     int time_slice;
+     u64 deadline;
+     struct list_head run_list;
+     u64 last_ran;
+     u64 sched_time; /* sched_clock time spent running */
+ #ifdef CONFIG_SMT_NICE
+     int smt_bias; /* Policy/nice level bias across smt siblings */
+ #endif
+ #ifdef CONFIG_SMP
+     bool sticky; /* Soft affined flag */
+ #endif
+ #ifdef CONFIG_HOTPLUG_CPU
+     bool zerobound; /* Bound to CPU0 for hotplug */
+ #endif
+     unsigned long rt_timeout;
+ #else /* CONFIG_SCHED_BFS */

```

It is added to sched.h and is the head of the run queue linked list. Here are the functions that add tasks to the run queue.

```

+static inline bool task_queued(struct task_struct *p)
+{
+     return (!list_empty(&p->run_list));
+}
+
+/*
+ * Removing from the global runqueue. Enter with grq locked.
+ */
+static void dequeue_task(struct task_struct *p)
+{
+     list_del_init(&p->run_list);
+     if (list_empty(grq.queue + p->prio))

```

```

+         __clear_bit(p->prio, grq.prio_bitmap);
+         sched_info_dequeued(task_rq(p), p);
+     }
+
+

```

The theory with the BFS scheduler is that it is minimalist and has a single run queue for handling all tasks which are marked running and awaiting a CPU resource. This minimalist design, however, actually makes the BFS scheduler hard to understand. Instead of a nicely segregated differentiation of classes, tasks, sched_entities, and rbtree nodes, the functionality of the BFS scheduler is spread around and scattered, and poorly documented.

There is supposed to be one run queue, but there are now 103 queues in the BFS scheduler, with 100 for RT processes used FIFO. And then there are the three standard priority queues, SCHED_ISO, SCHED_NORMAL, and SCHED_IDLEPRIO. Priorities are added into a calculation to produce a virtual deadline, that determines which tasks on the run queue is give resources. This deadline includes a calculation for a time slice which helps determine the virtual deadline. This is defined in the global_rq structures.

```

+ * The global runqueue data that all CPUs work off. Data is protected either
+ * by the global grq lock, or the discrete lock that precedes the data in this
+ * struct.
+ */
+struct global_rq {
+    raw_spinlock_t lock;
+    unsigned long nr_running;
+    unsigned long nr_uninterruptible;
+    unsigned long long nr_switches;
+    struct list_head queue[PRIO_LIMIT];
+    DECLARE_BITMAP(prio_bitmap, PRIO_LIMIT + 1);
+#ifdef CONFIG_SMP
+    unsigned long qnr; /* queued not running */
+    cpumask_t cpu_idle_map;
+    bool idle_cpus;
+#endif
+    int noc; /* num_online_cpus stored and updated when it changes */

```

```

+     u64 niffies; /* Nanosecond jiffies */
+     unsigned long last_jiffy; /* Last jiffy we updated niffies */
+
+     raw_spinlock_t iso_lock;
+     int iso_ticks;
+     bool iso_refractory;
+};

```

The functions that calculate the virtual deadline are as follows:

```

+/*
+ * Deadline is "now" in niffies + (offset by priority). Setting the deadline
+ * is the key to everything. It distributes cpu fairly amongst tasks of the
+ * same nice value, it proportions cpu according to nice level, it means the
+ * task that last woke up the longest ago has the earliest deadline, thus
+ * ensuring that interactive tasks get low latency on wake up. The CPU
+ * proportion works out to the square of the virtual deadline difference, so
+ * this equation will give nice 19 3% CPU compared to nice 0.
+ */
+static inline u64 prio_deadline_diff(int user_prio)
+{
+     return (prio_ratios[user_prio] * rr_interval * (MS_TO_NS(1) / 128));
+}
+
+static inline u64 task_deadline_diff(struct task_struct *p)
+{
+     return prio_deadline_diff(TASK_USER_PRIO(p));
+}
+
+static inline u64 static_deadline_diff(int static_prio)
+{
+     return prio_deadline_diff(USER_PRIO(static_prio));
+}
+

```

```
+static inline int longest_deadline_diff(void)
+{
+    return prio_deadline_diff(39);
+}
+
```

These functions return the deadline member of `task_struct`, listed above, in blue in the code.

While it might be minimal, the BFS is difficult to understand as compared to the CFS. The CFS can be easily described in terms of its structures, classes, and run time functions. The BFS scheduler, however, is anything but easy to understand precisely or by example. It is supposed to have a single run time double linked list runqueue. But in the next paragraph of its documentation, it makes the point that it has 103 queues, all of which are reviewed with each task lookup. The methods of handling the CPUs, and the method of assigning tasks to local CPUs in order to take advantage of NUMA, is far from immediately clear. The flow of data is hard to follow. There are “wakee” queues, and complexities for tasks priority and how time splices are calculated. In my literature search on the BFS scheduler, many articles are written on experimentation with the BFS scheduler that measurements of latency and benchmarks for performance. So many that it is not even worth doing yet another one. I haven't yet found any on the details of the working source code, although there are commonly available such descriptions for the CFS scheduler. You have to be a very accomplished and knowledgeable kernel C programmer to understand the BFS code. I still have much to learn on its specifics.

Patching the Kernel

We have the kernel 3.19.5 so we downloaded the kernel patch for the 3.19 kernel version from the BFS website. Copy the patch file to the `/usr/src/linuxxxx` directory where your source code is. Then use the `patch` command and compile as follow:

```
[ruben@manjaro linux-3.19.5]$ patch -p1 < 3.19-sched-bfs-461.patch
patching file arch/powerpc/platforms/cell/spufs/sched.c
patching file Documentation/scheduler/sched-BFS.txt
patching file Documentation/sysctl/kernel.txt
patching file fs/proc/base.c
patching file include/linux/init_task.h
```

```
patching file include/linux/ioprio.h
patching file include/linux/sched.h
patching file init/Kconfig
patching file init/main.c
patching file kernel/delayacct.c
patching file kernel/exit.c
patching file kernel/sysctl.c
patching file lib/Kconfig.debug
patching file include/linux/jiffies.h
patching file drivers/cpufreq/cpufreq.c
Hunk #2 succeeded at 1990 (offset 3 lines).
patching file drivers/cpufreq/cpufreq_ondemand.c
patching file kernel/sched/bfs.c
patching file include/uapi/linux/sched.h
patching file kernel/stop_machine.c
patching file drivers/cpufreq/cpufreq_conservative.c
patching file kernel/time/Kconfig
patching file kernel/sched/Makefile
patching file kernel/sched/bfs_sched.h
patching file kernel/sched/stats.c
patching file arch/x86/Kconfig
patching file include/linux/sched/prio.h
patching file drivers/cpufreq/intel_pstate.c
patching file kernel/sched/idle.c
patching file kernel/time/posix-cpu-timers.c

[ruben@manjaro linux-3.19.5]$ make xconfig
CHECK    qt
MOC      scripts/kconfig/qconf.moc
HOSTCXX  scripts/kconfig/qconf.o
HOSTLD   scripts/kconfig/qconf
scripts/kconfig/qconf Kconfig
```

This will bring up the configuration screen (make menuconfig) and we can just peek to see that the

BFS scheduler is installed as an option and selected.

After that we run `make, sudo make modules_install`, as we did before.

```
[ruben@manjaro src]$ ls -al /boot
total 119804
drwxr-xr-x  8 root root   4096 May  3 00:21 .
drwxr-xr-x 17 root root   4096 May  2 20:46 ..
drwxr-xr-x  2 root root   4096 Dec 31 1969 efi
drwxr-xr-x  3 root root   4096 Mar 21 01:44 EFI
drwxr-xr-x  6 root root   4096 May  1 03:41 grub
-rw-r--r--  1 root root 18575110 Apr  6 22:31 initramfs-318-x86_64-fallback.img
-rw-r--r--  1 root root  3387082 Apr  6 22:31 initramfs-318-x86_64.img
-rw-r--r--  1 root root 19104682 Mar 22 20:48 initramfs-3192-RS-x86_64-fallback.img
-rw-r--r--  1 root root  4052743 Mar 22 20:47 initramfs-3192-RS-x86_64.img
-rw-r--r--  1 root root 18519337 May  1 03:40 initramfs-319_liu_test-fallback.img
-rw-r--r--  1 root root  3436742 May  1 03:39 initramfs-319_liu_test.img
-rw-r--r--  1 root root 18648066 Apr  6 22:32 initramfs-319-x86_64-fallback.img
-rw-r--r--  1 root root  3388714 Apr  6 22:31 initramfs-319-x86_64.img
-rw-r--r--  1 root root   663040 Feb  9 10:42 intel-ucode.img
-rw-r--r--  1 root root     22 Apr  4 17:52 linux318-x86_64.kver
-rw-r--r--  1 root root     21 Mar 26 15:19 linux319-x86_64.kver
drwx-----  2 root root   16384 Mar 22 11:30 lost+found
drwxr-xr-x  2 root root   4096 Oct  6 2013 memtest86+
drwxr-xr-x  2 root root   4096 Apr  6 22:49 syslinux
-rw-r--r--  1 root root 2652731 May  3 00:21 System.map
-rw-r--r--  1 root root 2667040 May  1 03:41 System.map.bak
-rw-r--r--  1 root root 2669940 May  1 03:41 System.old
-rw-r--r--  1 root root  4067856 May  3 00:21 vmlinuz
-rw-r--r--  1 root root  4087328 Apr  4 17:52 vmlinuz-318-x86_64
-rw-r--r--  1 root root  4116496 Mar 22 20:38 vmlinuz-3192-RS-x86_64
-rw-r--r--  1 root root  4130640 May  1 03:39 vmlinuz-319_liu_test
-rw-r--r--  1 root root  4148048 Mar 26 15:19 vmlinuz-319-x86_64
-rw-r--r--  1 root root  4116496 Mar 22 20:08 vmlinuz.old
```

We notice that `make install` created a new `vmlinuz` in the `/boot` directory. We can just create a hard link from the name of the kernel that we want to the `vmlinuz`. First backup the old images and the `initramfs`

```
[ruben@manjaro src]$ cd /boot
```

```
[ruben@manjaro src]$ mv vmlinuz-319_liu_test vmlinuz-319_liu_test.old
[ruben@manjaro src]$ ln -s vmlinuz vmlinuz-319_liu_test
[manjaro boot]# mv initramfs-319_liu_test.img initramfs-319_liu_test.old.img
[manjaro boot]# mv initramfs-319_liu_test-fallback.img initramfs-319_liu_test-fallback.old.img
```

we need to create the initramfs. We are lazy, so, since we are shell jockeys, we look in the command line history to see what we ran before and just run it again. I leave this here unedited as instruction for would be commandline jockeys.

```
[ruben@manjaro boot]$ history|grep mk
 6 sudo mkinitcpio
 7 sudo mkinitcpio
 8 sudo mkinitcpio ^C
43 ALL_config="/etc/mkinitcpio.conf"
46 #default_config="/etc/mkinitcpio.conf"
49 #fallback_config="/etc/mkinitcpio.conf"
83 sudo mkinitcpio -p linux_liutest
84 sudo mkinitcpio -p linux_liutest
85 ls /etc/mkinitcpio.d/
86 sudo mkinitcpio -p linux319_liutest.present
87 sudo mkinitcpio -p linux319_liutest
89 sudo mkinitcpio -p linux319_liutest
101 sudo mkinitcpio -p linux319_liutest
105 cd /etc/mkinitcpio.d/
117 vim /etc/mkinitcpio.d/linux319_liutest.preset
118 sudo vim /etc/mkinitcpio.d/linux319_liutest.preset
122 sudo vim /etc/mkinitcpio.d/linux319_liutest.preset
123 sudo vim /etc/mkinitcpio.d/linux319_liutest.preset
125 history|grep mk
126 sudo vim /etc/mkinitcpio.d/linux319_liutest.preset
127 sudo mkinitcpio -p linux319_liutest
129 sudo mkinitcpio -p linux319_liutest
132 sudo mkinitcpio -p linux319_liutest
142 sudo mkinitcpio -p linux319_liutest
171 history|greo mk
172 history|grep mk
```

```
[ruben@manjaro boot]$ !142
sudo mkinitcpio -p linux319_liutest

[sudo] password for ruben:
==> Building image from preset: /etc/mkinitcpio.d/linux319_liutest.preset: 'default'
  -> -k /boot/vmlinuz-319_liu_test -c /etc/mkinitcpio.conf -g /boot/initramfs-319_liu_test.img
==> Starting build: 3.19.5-MANJARO
  -> Running build hook: [base]
  -> Running build hook: [udev]
  -> Running build hook: [autodetect]
  -> Running build hook: [modconf]
  -> Running build hook: [block]
  -> Running build hook: [filesystems]
  -> Running build hook: [keyboard]
  -> Running build hook: [keymap]
  -> Running build hook: [fsck]
  -> Running build hook: [usr]
  -> Running build hook: [shutdown]
==> Generating module dependencies
==> Creating gzip-compressed initcpio image: /boot/initramfs-319_liu_test.img
==> Image generation successful
==> Building image from preset: /etc/mkinitcpio.d/linux319_liutest.preset: 'fallback'
  -> -k /boot/vmlinuz-319_liu_test -c /etc/mkinitcpio.conf -g /boot/initramfs-319_liu_test-fallback.img -S autodetect
==> Starting build: 3.19.5-MANJARO
  -> Running build hook: [base]
  -> Running build hook: [udev]
  -> Running build hook: [modconf]
  -> Running build hook: [block]
==> WARNING: Possibly missing firmware for module: aic94xx
==> WARNING: Possibly missing firmware for module: wd719x
  -> Running build hook: [filesystems]
  -> Running build hook: [keyboard]
  -> Running build hook: [keymap]
  -> Running build hook: [fsck]
  -> Running build hook: [usr]
  -> Running build hook: [shutdown]
==> Generating module dependencies
==> Creating gzip-compressed initcpio image: /boot/initramfs-319_liu_test-fallback.img
==> Image generation successful

[ruben@manjaro boot]$ sudo update-grub

[sudo] password for ruben:

Generating grub configuration file ...

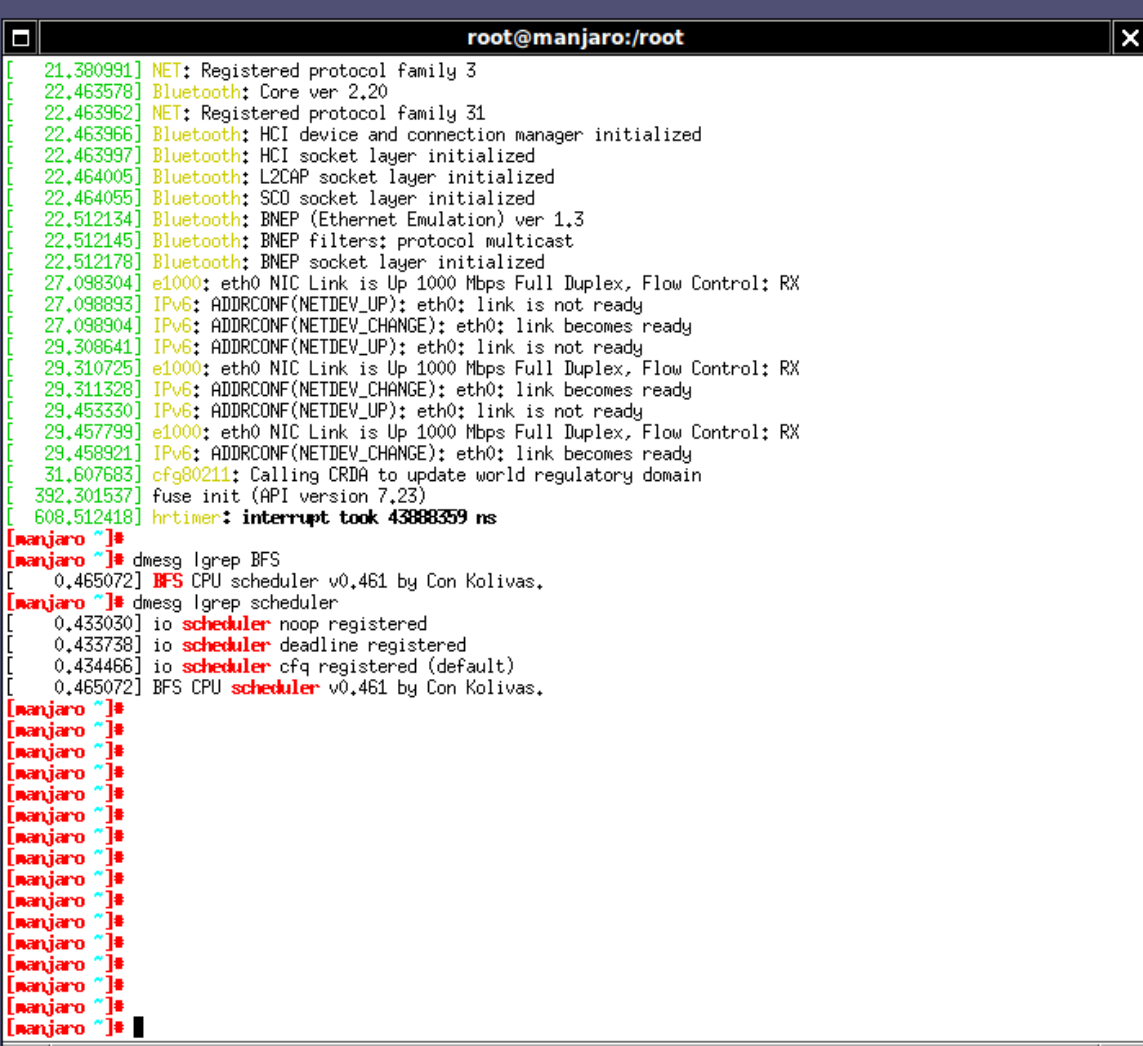
Found Intel Microcode image
```

```
Found linux image: /boot/vmlinuz-3192-RS-x86_64
Found initrd image: /boot/initramfs-3192-RS-x86_64.img
Found initrd fallback image: /boot/initramfs-3192-RS-x86_64-fallback.img
Found linux image: /boot/vmlinuz-319_liu_test
Found initrd image: /boot/initramfs-319_liu_test.img
Found initrd fallback image: /boot/initramfs-319_liu_test-fallback.img
Found linux image: /boot/vmlinuz-319_liu_test.old
Found initrd image: /boot/initramfs-319_liu_test.old.img
Found linux image: /boot/vmlinuz-319-x86_64
Found initrd image: /boot/initramfs-319-x86_64.img
Found initrd fallback image: /boot/initramfs-319-x86_64-fallback.img
Found linux image: /boot/vmlinuz-318-x86_64
Found initrd image: /boot/initramfs-318-x86_64.img
Found initrd fallback image: /boot/initramfs-318-x86_64-fallback.img
/run/lvm/lvmetad.socket: connect failed: No such file or directory
WARNING: Failed to connect to lvmetad. Falling back to internal scanning.
Found memtest86+ image: /boot/memtest86+/memtest.bin
done
[ruben@manjaro boot]$ dmesg|grep sched
[ 0.454569] io scheduler noop registered
[ 0.455219] io scheduler deadline registered
[ 0.455931] io scheduler cfq registered (default)
[ruben@manjaro boot]$
```

Notice that under the old kernel there is no CFS process scheduler list with the command: `dmesg|grep scheduler`. It is still there. It is just not listed. But when we changed the scheduler it shows in the `dmesg` command. When we reboot with the BFS scheduler, it will be reported.

At this point, we run `update-grub` and we have our new patched kernel in place. We can reboot and run it. A note to anyone doing this in a VM, the virtualbox modules need to be rebuilt to have cut and paste between the host and guest environments.

After reboot:



```
root@manjaro:/root
[ 21.380991] NET: Registered protocol family 3
[ 22.463578] Bluetooth: Core ver 2.20
[ 22.463962] NET: Registered protocol family 31
[ 22.463966] Bluetooth: HCI device and connection manager initialized
[ 22.463997] Bluetooth: HCI socket layer initialized
[ 22.464005] Bluetooth: L2CAP socket layer initialized
[ 22.464055] Bluetooth: SCO socket layer initialized
[ 22.512134] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
[ 22.512145] Bluetooth: BNEP filters: protocol multicast
[ 22.512178] Bluetooth: BNEP socket layer initialized
[ 27.098304] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[ 27.098893] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[ 27.098904] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 29.308641] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[ 29.310725] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[ 29.311328] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 29.453330] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[ 29.457799] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[ 29.458921] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 31.607683] cfg80211: Calling CRDA to update world regulatory domain
[ 392.301537] fuse init (API version 7.23)
[ 608.512418] hrtimer: interrupt took 43888359 ns
[manjaro ~]#
[manjaro ~]# dmesg |grep BFS
[   0.465072] BFS CPU scheduler v0.461 by Con Kolivas.
[manjaro ~]# dmesg |grep scheduler
[   0.433030] io scheduler noop registered
[   0.433738] io scheduler deadline registered
[   0.434466] io scheduler cfq registered (default)
[   0.465072] BFS CPU scheduler v0.461 by Con Kolivas.
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
[manjaro ~]#
```

We can see out BFS CPU Scheduler. Now lets see how it handles out stress program.

It handled stress.exe with up to a thousand threads without a problem. As before, it seems to limit out on memory above that number of threads being created.

Conclusion

The GNU/Linux scheduler is an idea starting point for learning about operating system design, coding and construction. Most of the major topics of modern computing are involved with process scheduling and the organization and source code availability of GNU/Linux systems allows for exploration of this hot area of research. With the extensive explosion of devices running GNU based systems, from android phones, real-time devices, remote virtual servers, arm based robots, in addition to the standard PC based work stations and laptops, schedulers are needed for all sorts of different conditions. Improvements and changes in the scheduler is inevitable. So learning about the GNU scheduler is not just an academic issue, but a critical area of understanding for anyone entering the computer sciences field who hopes to do important work.

This paper is only a beginning exploration of this important topic. It shows that anyone can pull down the source code, pick up a few decent text books, roll up there sleeves and learn about this core kernel feature. Most importantly, I hope that this paper fuels your curiosity and show you where to find like minded individuals in a quest for understanding and knowledge. You can pick this paper up, and use it as a road map to go the next step, and the step after. In comp sci, learning is a lifetime commitment and GNU/Linux is a lifetime partner on that adventure.

With all the noise and chatter on the internet and within the social media, the best collection of individuals for learning and finding technological information, conversation and feedback continues to be, as it was 20 years ago, on usenet. This might suprise some people, but it is 100% true. This paper would not have been possible without the contribution of regular participants and readers of comp.unix.programmer, comp.os.linux.hardware and comp.os.linux.misc . And these are not the only groups of important value. Additionally, there is a kernel developers newbies mailing list. It is available at <http://selenic.com/mailman/listinfo/kernel-mentors>. From this list I was pointed to very good text resources which greatly improved my knowledge and explained in detail kernel scheduling, as well as most of the rest of the system. See the Bibliography for a listing. After this, of course wikipedia can be helpful, when it is not confusing. And in addition to this, it needs to be mentioned that the university professors are an outstanding source of information, and comrade, that is your partners in learning. After complaining about workloads, grades, and other aspects of daily school life, I encourage students to powow with the staff. Email them at 2AM and pop into their office with

question. They are great mentors and have more to contribute than you might well believe. In terms of this paper, important contributions were made by Professor Mohamed Ghriga, who put me right onto appropriate texts which directly involved problems faced in this paper and its development.

As for myself, as an author, I know that this paper has opened more doors than it closed. It is not sufficient. There is so much more to learn, and this paper is just starting point. There should be in the future one or several more follow ups.

Bibliography

- 1: Robert Love, Linux Kernel Development, 2010
- 2: K.N. King, C Programming: A Modern Approach, 1996
- 3: Mark Allen Weiss, Structures Allgorithm Analysis in C, 1992
- 4: Cormen, Leiserson, Rivest, Stein, Introduction to Allorithms,
- 5: Greg Wilson et al, Beautiful Code, 2007
- 6: Hank Dietz: Linux Parallel Processing HOWTO, 2009, [Linux Foundation](#)
- 7: Linus Tovalds, Kernel Intallation Read Me, 2015
- 8: Linus Tovalds: CFS Scheduler - Kernel Documentation, 2015, [Linux Foundation](#)
- 9: SuSE Enterprises: Systems Analysis and tuning Guid - Completely Fair Scheduler, 2012, [SuSE](#)
- 10: Wikipedia: Completely Fair Scheduler, 2015,
http://en.wikipedia.org/wiki/Completely_Fair_Scheduler#OS_background [Wikipedia](#)
- 11: WIKPEDIA, http://en.wikipedia.org/wiki/Brain_Fuck_Scheduler, 2015
- 12: Con Kolivas: FAQs about BFS. v0.330, 2010, [Con Kolivas](#)
- 13: Con Kolivas: 4.0 BFS Patch, 2015, [Con Kolivas](#)
- 14: M. Tim Jones,: Inside the Linux 2.6 Completely Fair Scheduler, 2009,
<http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/> [IBM](#)
- 15: M. Tim Jones: Linux Scheduler simulation, 2011, <http://www.ibm.com/developerworks/library/l-linux-scheduler-simulator/> [IBM](#)
- 16: Ankita Garg: Real-Time Linux Kernel Scheduler, 2009,
<http://www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler> [The Linux Journal](#)
- 17: Ingo Molnar: Modular Scheduler Core and Completely Fair Scheduler, 2007,
<http://lwn.net/Articles/230501/> [LWN.NET](#)
- 18: Steven M. Bellovin : The Linux Scheduler, 2015,
<https://www.cs.columbia.edu/~smb/classes/s06-4118/l13.pdf> [Columbia University](#)
- 19: William Nagel: Soft Real-Time Programming with Linux, 2005, <http://www.drdoobs.com/soft-real-time-programming-with-linux/184402031> [Dr Dobbs](#)
- 20: Jaqui Lynch: Linux Performance Tuning, 2007, [IBM Systems Magazine](#)
- 21: Safir,Pitcher, Margolin, Sosman, Kylheku, guenther et al: marco typeof and 0 , 2015, [usenet comp.unix.programmerer USENET comp.unix.programmer](#)
- 22: Safir, Weikusat, Kylheku et al: Kernel and Wait, 2015, [comp.unix.programmer comp.unix.programmer](#)

Open Source Hardware: A Technical Review of the Arduino

By Fernando Gutierrez.
March 26, 2015

If you are into hacking hardware, interfacing bits of hardware to your computer, creating robots or automated systems for your car or home, then an Arduino is for you. If you are not into these things but have an interest on how things are done, then an Arduino maybe for you. If you have no interest in these things, then it is not for you at all. But with a little imagination, and a lot of work in hardware assembly and software programming, you can build something with an Arduino. But what is an Arduino?

Started in Italy at the same time the Raspberry Pi started in the UK in 2005, the Arduino system was to introduce hardware interfacing to electrical engineering students in Italian Universities. To quote Wikipedia:

Arduino started in 2005 as a project for students at the Interaction Design Institute Ivrea in Ivrea, Italy. At that time program students used a "BASIC Stamp" at a cost of \$100, considered expensive for students. Massimo Banzì, one of the founders, taught at Ivrea. The name "Arduino" comes from a bar in Ivrea, where some of the founders of the project used to meet. The bar itself was named after Arduino, Margrave of Ivrea and King of Italy from 1002 to 1014.

A hardware thesis was contributed for a wiring design by Colombian student Hernando Barragan. After the Wiring platform was complete, researchers worked to make it lighter, less expensive, and available to the open source community. The school eventually closed, but the researchers, including David Cuartielles, promoted the idea. (Arduino. In Wikipedia. Retrieved March 24, 2015, from <http://en.wikipedia.org/wiki/Arduino>)

Thus Arduino was “born.” Using the concepts of Open Source Software, Open Source Hardware was created for the Arduino, meaning anyone can copy it, make changes to it and most importantly, share the technology with others. The thing is, people in general do not have the means to create their own IC Chips, so how is this “Open Source Hardware?”

It is Open Source Hardware in that you can buy what parts you need from it to create one from their design. There are many schematics, plans, videos and various demonstrations in creating an Arduino.

All one needs to create an Arduino is an ATMEL Microcontroller chip, a couple capacitors, a timing crystal, a few resistors, a voltage regulator, LEDs, and a board to put it all on. The heart of the Arduino is the ATMEL AVR-Mega Microcontroller: ATmega8, ATmega168, ATmega328, ATmega1280, and ATmega2560. Each chip has a specific amount of RAM, ROM, Digital and Analog I/O; which can be programmed with a Software Development Kit (SDK) through the C++ language. Arduino makes this easy, programming and compiling of the C++ program and uploading it to the Arduino board through their IDE (Integrated Development Environment). It is available for Windows, Macintosh OSX and Linux.

So you have a microcontroller on a board that you can program, what can you do with it? Depending on what hardware you interface to the board - anything. The power of the Arduino is based on both the ease of hardware interfacing and software programming of the device. For example, if you want to try out 3D printing, there are many Open Source 3D Printer Kits that you can build using the Arduino system. It just requires that you assemble the unit. And many already built 3D Printer kits use the Arduino as its controller. You can also create a car/home burglar alarm, soil moisture sensor for your garden, robot kits for you and your children, and so on. It's only limits are those imposed by hardware, your imagination and how much work you put into it.

But do note, the hardware limitations can be a brick wall in where you cannot overcome; the majority of the Arudinoes out there only have 32K of RAM/ROM Space. 32K is a lot of space for small programs, but not for an OS. A Microcontroller can function without an OS, which is the beauty of the thing. All it needs is the program to run in an infinite loop unless it meets a conditional programmed end to the loop. It can be interfaced into a computer through the USB, serial, In-System Programming (ISP) Header, or GPIO port. The Atmel AVR Microcontroller on the Arduinoes is programmed with a Boot Loader, which allows the Arduino to communicate with the computer and allow program uploading to it.

The Basic Model of Arduino is the Arduino Uno R3 (Revision 3). It uses the Atmel ATmega168 with an FTDi Serial interface for USB Communications. It contains 14 digital I/O pins, which 6 of them can produce pulse-width modulated signals, and another 6 analog inputs, which can also be used as an additional 6 digital I/O pins. These pins are on a female header strip for solderless connections to the board for ease of construction and prototyping. The Microcontroller runs at 16MHz. Other Arduino boards can run as slow as 8MHz and as fast as 64MHz (200MHz for the Arduino Yun which is an ARM Processor Credit Card Size Computer like the Raspberry Pi with Built-In Wfi running Linux, with added features of an Arduino Uno tied to the computer). Board sizes range from the tiny Arduino Mini at about 3/4in X 1 1/2in with about 14 I/O pins, to the large sized Arduino Mega at about 2 3/4in X 4

1/2in with over 64 I/O pins.

Since the Arduino is Open Source Hardware, there are many companies out there selling their own Arduino Compatible devices. Names like Freeduino and Bluno are a couple of many such systems. Most Arduino Clone Systems use the Arduino ATmega chip with a FDTI Serial Adapter. But some companies, mostly Chinese Manufacturers, use the CH340 USB to Serial Adapter for USB communications. For them this might be an improvement in the USB communications of the Arduino Clone to the Computer, the host computer may not have the CH340 drivers, making the Arduino Clone look dead when it is not. The easiest solution is to download the CH340 USB Driver.

Also some Arduino Clone Manufacturers have their Arduinos run at a slower speed, many I have seen are running at 12MHz. This can cause some problems with communications with the computer you are using. This can be remedied by replacing the slower crystal with a 16MHz crystal. Sometimes an Arduino Clone could have its Bootloader missing, this can be remedied by using a second Arduino to program the Bootloader through the ISP Header of the first Arduino with the missing Boot Loader. Many Arduino kits are sold online range in price from \$3 to \$45 a piece, the lower cost options will have more issues to deal with but these issues are simple to deal with, like use of the slower crystal, programming in a bootloader and need of the CH340 driver.

An extensive list of Arduino systems and various clones can be found on Wikipedia:

http://en.wikipedia.org/wiki/List_of_Arduino_boards_and_compatible_systems

Once you get an Arduino and begin programming it, you will want to get more. With their low costs, this is possible. I would recommend getting at least one of each model: Arduino Uno, Arduino Leonardo, and an Arduino Mega. The Arduino Mini, Arduino Nano and Arduino Micro can be bought later on as optional boards though they are cheaper than the Arduino Uno. You should also get an original Arduino Unit like an Arduino Uno 3 in case you need to program a bootloader onto an Arduino clone that does not have one. Once you complete one project, you can get another Arduino for your next project.

Why get so many units? Though there is a standard of what an Arduino should be, you might want to create a hardware device that requires more than 24 I/O Ports. Only the Arduino Mega can handle up to 64 I/O ports and with 256K of RAM/ROM Space, your program can be large enough to handle them all. Another reason is when you are done with your prototype and want to make a more permanent

installation, you will need to move your project off the Arduino Uno and move it onto an Arduino Mini, Nano or Micro board by permanently soldering the wires to the needed ports. At under \$4 a board for an Arduino Mini, Nano or Micro clone board, this makes it feasible in costs and time to do! This will free up your Arduino Uno for your next project while your current project does its thing.

Again, if you tinker about with hardware and interfacing projects to your computer, an Arduino board is an excellent low cost option for you. They are capable of many functions from sensor networks, indicators, analog/digital communications, and so on. Their only limitation is what you can do with it and your skills to implement your ideas.

Links:

Arduino Website: <http://www.arduino.cc>

Arduino projects from the Arduino Website: <http://playground.arduino.cc/projects/ideas>

Various Arduino project from Make.com: <http://makezine.com/category/electronics/arduino/>

Source for Arduino and Arduino Clone boards at a low price, search on: <http://www.ebay.com>

Assorted Arduino Boards

(NOTE: Scale not to size!)

77



Arduino Uno -

“The Standard Arduino”

16 Digital I/O and 6 Analog I/O Pins
with 32K of RAM/ROM



Arduino Leonardo -

Single Chip Design with
improved USB Communications
and lower power requirements.

Otherwise same function and
size as the Uno.



Arduino Mega -

ATMega256 Chip, with 256K
RAM/ROM and 64 I/O pins.

Same width, but about twice
the length of an Uno Board



Arduino Nano -

Same function as the Uno,
but at 1/4 the board size!

Uses the same ATMega Chip
as on the Uno.



Arduino Mini -

Smaller yet same function as
the Nano, but with the USB
connector board as a separate
unit. Great for stand alone projects.



ISBN 978-1-329-11030-4 90000
9 781329 110304