Modifying a kernel for Bossa : an example with Linux kernel 2.6.32

Florian David
Pierre et Marie Curie University
4 place Jussieu, 75005 Paris, France
http://bossa.lip6.fr/

February 27, 2011

Contents

1 Introduction

Bossa is a kernel-level event-based framework that simplifies implementation of new schedulers for the Linux kernel. Bossa provides a domain-specific language that helps to implement new scheduling policies.

To support Bossa, a version of Linux must initially be modified to disable the existing scheduling support and to implement the Bossa event notifications. The specific modifications required may vary across the different Linux versions, as the Linux scheduling mechanisms evolve. This is a task for an OS expert.

The aim of this document is to describe the modifications made to version 2.6.32 of the Linux kernel to support Bossa. The strategies used may be applicable to future versions of the Linux kernel.

2 Prerequisites

2.1 Kernel configuration

The Bossa kernel is compatible with most of the available kernel modules and configuration options, but we need to disable three options:

- CONFIG_SMP: located under Processor type and features → Symmetric multiprocessing support.
- CONFIG_GROUP_SCHED: located under General setup → Group CPU scheduler.
- CONFIG_HIGH_RES_TIMERS: located under Processor type and features → High Resolution Timer Support.

2.2 Description of the Bossa runtime

We must first copy the files of the Bossa runtime into the kernel source tree. These files are as follows:

- include/linux/bossa_event_headers.h
- include/linux/bossa_events.h
- include/linux/bossa.h
- include/linux/bossa_policy_rts.h
- include/linux/bossa_rts.h
- include/linux/bossa_rts_kernel.h
- kernel/bossa/Kconfig
- kernel/bossa/sched.c
- kernel/bossa.c
- kernel/Linux.c
- kernel/prim.c
- kernel/prim1.c

2.3 Bossa kernel and Makefile configuration

The Linux kernel configuration tool needs to know where to find the file that contains the Bossa configuration options. This can be done by opening the arch/your_arch/Kconfig file and adding the following line: *source "kernel/bossa/Kconfig"*. Bossa options will now appear in the configuration menu. Note that it is better to modify the architecture specific Kconfig because the position of the call to rts_clocktick() depends on the computer architecture (cf. Section 4.4).

The Makefile must also know which files to compile when Bossa is enabled. There are only kernel/Linux.c and kernel/bossa.c to compile. The following lines must be added to kernel/Makefile in order to compile these two files.

```
1 obj-$(CONFIG_BOSSA) += bossa.o
2 obj-$(CONFIG_BOSSA) += Linux.o
```

3 Disabling the Linux scheduler

Linux's CFS (Completely Fair Scheduler) does all the scheduling management, such as choosing the next elected process, deciding whether to wake up a task preemptively, etc. Removing function calls related to the CFS is thus essential to prevent it from interfering with the behaviour of Bossa.

All of the calls to the CFS are performed by the process's scheduling class (field *sched_class* of the *task_struct* structure). Each call to a function from the *sched_class* structure has to be removed and potentially be replaced by a Bossa function. Bossa must also be the only one to ask for reschedule, so any function that tries to set the resched flag, for example, by calling functions set_tsk_need_resched(struct task_struct*) or resched_task(struct task_struct*) should also be removed.

We can remove functions by two equivalent methods: suppressing the whole definition or removing only the function body. Removing the function body is easier when the function is available in a header file and can potentially be used anywhere in the kernel. In that way, we do not need to remove every call to the function in the kernel.

The whole definition can be deleted if the use of the function is limited to the scope of a single file. In this case, we have to delete each call to this function in this file. We use this solution for clarity and to ensure that the functions are not called anymore. Preprocessing directives (*#ifdef CONFIG_BOSSA*) are used for both purpose.

All functions described in this section are located in the kernel/sched.c file.

3.1 Enqueue_task() and dequeue_task()

When a task enters in a runnable state, the scheduler calls the function enqueue_task(struct rq *, struct task_struct *, int), which puts the task into the runnable-task structure of the scheduler. The function dequeue_task(struct rq *, struct task_struct *, int) then takes the task out of the structure when the task is no longer runnable. Since Bossa decides which task should be in a runnable state and maintains its own data structures, these two functions must not be called.

```
#ifndef CONFIG_BOSSA
static void enqueue_task(struct rq *rq, struct task_struct *p, int wakeup) {
  if (wakeup)
    p->se.start_runtime = p->se.sum_exec_runtime;

  sched_info_queued(p);
  p->sched_class->enqueue_task(rq, p, wakeup);
  p->se.on_rq = 1;
}
                                                                          10
static void dequeue_task(struct rq *rq, struct task_struct *p, int sleep) {
  if (sleep) {
    if (p->se.last_wakeup) {
      update_avg(&p->se.avg_overlap,
                 p->se.sum_exec_runtime - p->se.last_wakeup);
      p->se.last_wakeup = 0;
    } else {
      update_avg(&p->se.avg_wakeup,
```

```
                sysctl_sched_wakeup_granularity);
    }
  }

  sched_info_dequeued(p);
  p->sched_class->dequeue_task(rq, p, sleep);
  p->se.on_rq = 0;
}
#endif
```

Because the functions activate_task(struct rq *, struct task_struct *, int) and deactivate_task(struct rq *, struct task_struct *, int) make an call to enqueue_task and dequeue_task; they must also be removed.

```
#ifndef CONFIG_BOSSA
/*
 * activate_task - move a task to the runqueue.
 */
static void activate_task(struct rq *rq, struct task_struct *p, int wakeup) {
        if (task_contributes_to_load(p))
                rq->nr_uninterruptible--;

        enqueue_task(rq, p, wakeup);
        inc_nr_running(rq);
}

/*
 * deactivate_task - remove a task from the runqueue.
 */
static void deactivate_task(struct rq *rq, struct task_struct *p, int sleep) {
        if (task_contributes_to_load(p))
                rq->nr_uninterruptible++;

        dequeue_task(rq, p, sleep);
        dec_nr_running(rq);
}
#endif
```

3.2 Wake_up_new_task

The function wake_up_new_task(struct task_struct *, unsigned long) is called just after a process has forked. It puts the task into the runnable-task structure (with activate_task()), notifies the scheduler that a new task has been created (with task_new() from the process's scheduling class) and checks if the newly created task should preempt the current one (with check_preempt_curr() function). All of these operations involve the Linux scheduler and they must not be called.

```
/*
 * wake_up_new_task - wake up a newly created task for the first time.
 *
 * This function will do some initial scheduler statistics housekeeping
 * that must be done for every newly created context, then puts the task
 * on the runqueue and wakes it.
 */
#ifndef CONFIG_BOSSA
void wake_up_new_task(struct task_struct *p, unsigned long clone_flags) {
```

```
  unsigned long flags;
  struct rq *rq;

  rq = task_rq_lock(p, &flags);
  BUG_ON(p->state != TASK_RUNNING);
  update_rq_clock(rq);

  if (!p->sched_class->task_new || !current->se.on_rq) {
    activate_task(rq, p, 0);
  } else {
    /*
     * Let the scheduling class do new task startup
     * management (if any):
     */
    p->sched_class->task_new(rq, p);
    inc_nr_running(rq);
  }
  trace_sched_wakeup_new(rq, p, 1);
  check_preempt_curr(rq, p, WF_FORK);
#ifdef CONFIG_SMP
  if (p->sched_class->task_wake_up)
    p->sched_class->task_wake_up(rq, p);
#endif
  task_rq_unlock(rq, &flags);
}
#endif
```

3.3 Scheduler_tick

The scheduler_tick() function is indirectly called by time interrupt handler to manage the system scheduling. This function might lead to a context switch by calling the task_tick() function from the process's scheduling class. Consequently, the function scheduler_tick() must be disabled.

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 *
 * It also gets called by the fork code, when changing the parent's
 * timeslices.
 */
void scheduler_tick(void) {
#ifndef CONFIG_BOSSA
  int cpu = smp_processor_id();
  struct rq *rq = cpu_rq(cpu);
  struct task_struct *curr = rq->curr;

  sched_clock_tick();

  spin_lock(&rq->lock);
  update_rq_clock(rq);
  update_cpu_load(rq);
  curr->sched_class->task_tick(rq, curr, 0);
  spin_unlock(&rq->lock);

  perf_event_task_tick(curr, cpu);

#ifdef CONFIG_SMP
  rq->idle_at_tick = idle_cpu(cpu);
```

```
  trigger_load_balance(rq, cpu);
#endif
#endif
}
```

3.4 Rt_mutex_setprio

The function rt_mutex_setprio(struct task_struct *, int) is used by real-time mutex code in order to implement priority inheritance. It can temporarily increase the priority of a lower priority process when that process holds a locks that is needed by the current task.

This function makes calls to functions from the scheduling class of the task (put_prev_task() and set_curr_task()) and to functions that have been previously removed (enqueue_task() and dequeue_task()). Moreover, only Bossa can manage a task's priority so this function must be disabled.

```
/*
 * rt_mutex_setprio - set the current priority of a task
 * p: task
 * prio: prio value (kernel-internal form)
 *
 * This function changes the 'effective' priority of a task. It does
 * not touch ->normal_prio like __setscheduler().
 *
 * Used by the rt_mutex code to implement priority inheritance logic.
 */
void rt_mutex_setprio(struct task_struct *p, int prio) {
#ifndef CONFIG_BOSSA
  unsigned long flags;
  int oldprio, on_rq, running;
  struct rq *rq;
  const struct sched_class *prev_class;

  BUG_ON(prio < 0 || prio > MAX_PRIO);

  rq = task_rq_lock(p, &flags);
  update_rq_clock(rq);

  oldprio = p->prio;
  prev_class = p->sched_class;
  on_rq = p->se.on_rq;
  running = task_current(rq, p);
  if (on_rq)
    dequeue_task(rq, p, 0);
  if (running)
    p->sched_class->put_prev_task(rq, p);

  if (rt_prio(prio))
    p->sched_class = &rt_sched_class;
  else
    p->sched_class = &fair_sched_class;

  p->prio = prio;

  if (running)
    p->sched_class->set_curr_task(rq);
  if (on_rq) {
```

```
    enqueue_task(rq, p, 0);

    check_class_changed(rq, p, prev_class, oldprio, running);
  }
  task_rq_unlock(rq, &flags);
#endif
}
```

3.5 __sched_setscheduler

The function __sched_setscheduler(struct task_struct *, int, struct sched_param *, bool) is used by sched_setscheduler() and sched_scheduler_nocheck() for modifying the scheduling class of a process. This function is irrelevant when using Bossa because Bossa does not use the scheduling class. Moreover, it calls many functions of the previous and new scheduling class of the process which can interfere with Bossa.

```
static int __sched_setscheduler(struct task_struct *p, int policy,
                                struct sched_param *param, bool user)
{
#ifndef CONFIG_BOSSA
  int retval, oldprio, oldpolicy = -1, on_rq, running;
  unsigned long flags;
  ...
  rt_mutex_adjust_pi(p);
#endif
    return 0;                                                          10
}
```

3.6 Normalize_task and normalize_rt_tasks

These two functions are used when the Magic SysRq key option is activated in the kernel configuration. They help in debugging the kernel in the case of a crash. Normalize_task() calls functions deactivate_task() and activate_task() which were previously removed (cf. Section 3.1) and it also sets the resched flag to the current process. The change to the resched flag is the reason why we need to remove this function. The function normalize_rt_tasks() must be removed because it calls normalize_task().

Despite the removal of these functions, the Magic SysRq key can still be used to obtain information that does not involve calling these functions. The Magic SysRq key can also be disabled completely using the Magic SysRq key option inside the kernel configuration (located under Kernel Hacking → Magic SysRq key).

```
#ifdef CONFIG_MAGIC_SYSRQ
static void normalize_task(struct rq *rq, struct task_struct *p)
{
#ifndef CONFIG_BOSSA
  int on_rq;

  update_rq_clock(rq);
  on_rq = p->se.on_rq;

  if (on_rq)                                                           10
    deactivate_task(rq, p, 0);
  __setscheduler(rq, p, SCHED_NORMAL, 0);
```

```
  if (on_rq) {
    activate_task(rq, p, 0);
    resched_task(rq->curr);
  }
#endif
}

void normalize_rt_tasks(void)
{
#ifndef CONFIG_BOSSA
  struct task_struct *g, *p;
  ...
  normalize_task(rq, p);
  ...
#endif
}
#endif
```

3.7 Pick_next_task and put_prev_task

The function pick_next_task(struct rq *) chooses the most appropriate eligible process to run. The function put_prev_task(struct rq *, struct task_struct *) deactivates the previous running task before the next election.

These two functions, used when an election is made, call functions of the scheduling class of the running process, so these functions need to be disabled when it is Bossa that decides which task to elect.

```
#ifndef CONFIG_BOSSA
static void put_prev_task(struct rq *rq, struct task_struct *p) {
  ...
  p->sched_class->put_prev_task(rq, p);
}
#endif

#ifndef CONFIG_BOSSA
static inline struct task_struct *
pick_next_task(struct rq *rq) {
  ...
  if (likely(rq->nr_running == rq->cfs.nr_running)) {
    p = fair_sched_class.pick_next_task(rq);
    if (likely(p))
      return p;
  }

  class = sched_class_highest;
  for ( ; ; ) {
    p = class->pick_next_task(rq);
    if (p)
      return p;
    /*
     * Will never be NULL as the idle class always
     * returns a non-NULL p:
     */
    class = class->next;
  }
}
```

```
#endif                                                                  30
```

4 Kernel source modification

4.1 Bossa events

The Bossa runtime system is based on nine event notifications, which has to be inserted at various places in the kernel source code. They allow Bossa to be notified when a kernel event occurs that is relevant to scheduling. Here is a list of the event notification functions:

- rts_clocktick()
- rts_create_process_wake(struct task_struct *)
- rts_create_process_sleep(struct task_struct *)
- rts_terminate_process(struct task_struct *)
- rts_unblock_preemptive(struct task_struct *)
- rts_unblock_nonpreemptive(struct task_struct *)
- rts_schedule(struct task_struct *)
- rts_set_priority(struct task_struct *, int)
- rts_yield(struct task_struct *)

Added code chunks have to be enclosed between the #ifdef CONFIG_BOSSA directive in order to enable the code only if Bossa is activated in the kernel options.

4.2 Sched.h header

In order to make Bossa functions available to the whole kernel, the following code must be added inside the include/linux/sched.h file:

```
#ifdef CONFIG_BOSSA
#ifdef EXPORTS
#include <linux/bossa_policy_rts.h>
#else
#include <linux/bossa_rts_kernel.h>
#endif
#endif
```

Each process task structure will have to contain some information about Bossa, so a data field has to be inserted inside the *task_struct* structure:

```
struct task_struct {
  ...
#ifdef CONFIG_BOSSA
  struct bossa_struct bossa;
  int bossa_data[CONFIG_BOSSA_DATA_SIZE];
#endif
};
```

4.3 Bossa_init

Bossa is initialized usinig the function bossa_init(). This function must be called inside the start_kernel() function in init/main.c file before interrupts are enabled.

```
asmlinkage void __init start_kernel(void) {
  ...
  sort_main_extable();
  trap_init();
#ifdef CONFIG_BOSSA
  bossa_init();
#endif
  mm_init();
  ...
  /* Do the rest non-__init'ed, we're now alive */                    10
  rest_init();
}
```

4.4 Rts_clocktick

The rts_clocktick() function notifies Bossa that a clock interrupt has occured. This function is inserted in the function timer_interrupt(int, void*), located in arch/cpu_arch/kernel/time.c. The linux/sched.h header must also be inserted at the beginning of this file.

For the x86 architecture, the modifications are made in the arch/x86/kernel/time.c.

```
#ifdef CONFIG_BOSSA
#include <linux/sched.h>
#endif

...

static irqreturn_t timer_interrupt(int irq, void *dev_id) {
  ...

#ifdef CONFIG_BOSSA                                                   10
  rts_clocktick();
#endif

  return IRQ_HANDLED;
}
```

4.5 Rts_create_process

Rts_create_process_wake(struct task_struct*) and rts_create_process_sleep(struct task_struct*) allow Bossa to know when a process is created. The former notifies Bossa that the created task is in the TASK_RUNNING state and the latter notifies Bossa that the task is in the TASK_STOPPED state.

The notification is added to the function do_fork() in the kernel/fork.c file:

```
long do_fork(unsigned long clone_flags, unsigned long stack_start, struct pt_regs *regs,
             unsigned long stack_size,
             int __user *parent_tidptr,
```

```
                int __user *child_tidptr) {

  ...
  if (unlikely(clone_flags & CLONE_STOPPED)) {
    /*
     * We'll start up with an immediate SIGSTOP.
     */
    sigaddset(&p->pending.signal, SIGSTOP);
    set_tsk_thread_flag(p, TIF_SIGPENDING);
#ifdef CONFIG_BOSSA
    rts_create_process_sleep(p);
#endif
    __set_task_state(p, TASK_STOPPED);
  } else {
#ifndef CONFIG_BOSSA
    wake_up_new_task(p, clone_flags);
#else
    rts_create_process_wake(p);
#endif
  }
  ...
}
```

The task is initialized inside this function so Bossa is notified that a new task has just been created and is ready to be elected. The call to wake_up_new_task() is disabled because we do not want the scheduling class to do the startup management by calling the CFS (cf. Section 3.2).

4.6 Rts_terminate_process

When a task terminates, Bossa is notified using the rts_terminate_process(struct task_struct*) function. The notification is done in the release_task() function in the kernel/exit.c file. The release_task() function is called when a task has exited and is ready to be freed.

```
void release_task(struct task_struct * p) {
  ...
  write_unlock_irq(&tasklist_lock);
  release_thread(p);
  call_rcu(&p->rcu, delayed_put_task_struct);

#ifdef CONFIG_BOSSA
  rts_terminate_process(p);
#endif

  p = leader;
  if (unlikely(zap_leader))
    goto repeat;
}
```

The call is made just before the function either ends or repeats. Just before the function repeats, the process has been freed and it is safe to notify Bossa.

4.7 Rts_unblock_preemptive and Rts_unblock_nonpreemptive

The functions rts_unblock_preemptive(struct task_struct*) and rts_unblock_nonpreemptive(struct task_struct*) inform Bossa what process is going to wake up. The former asks to reschedule after unblocking the task whereas the latter does not.

```
static int try_to_wake_up(struct task_struct *p, unsigned int state,
                          int wake_flags) {
  ...
  if (!(p->state & state))
    goto out;

#ifndef CONFIG_BOSSA
  if (p->se.on_rq)
    goto out_running;
#endif

  cpu = task_cpu(p);
  orig_cpu = cpu;

  schedstat_inc(p, se.nr_wakeups);
  if (wake_flags & WF_SYNC)
    schedstat_inc(p, se.nr_wakeups_sync);
  if (orig_cpu != cpu)
    schedstat_inc(p, se.nr_wakeups_migrate);
  if (cpu == this_cpu)
    schedstat_inc(p, se.nr_wakeups_local);
  else
    schedstat_inc(p, se.nr_wakeups_remote);
#ifndef CONFIG_BOSSA
  activate_task(rq, p, 1);
#endif
  success = 1;

  ...

 out_running:
  trace_sched_wakeup(rq, p, success);
#ifdef CONFIG_BOSSA
  if (wake_flags & WF_SYNC) {
    rts_unblock_preemptive(p);
  } else {
    rts_unblock_nonpreemptive(p);
  }
#else
  check_preempt_curr(rq, p, wake_flags);
#endif

  p->state = TASK_RUNNING;
 out:
  task_rq_unlock(rq, &flags);
  put_cpu();

  return success;
}
```

The function try_to_wake_up() (located in kernel/sched.c) is used to wake up a task. We use the flag WF_SYNC to know if the wake up should be preemptive or not. If the flag is set, the wake up is preemptive and Bossa sets the rescheduling flag of the currently running task. The function check_preempt_curr() is disabled because we don't want the CFS to interfere with Bossa as explained in Section 3.2.

4.8 Rts_set_priority

The rts_set_priority(struct task_struct*, int) function notifies Bossa that a process priority is asked to be modified, for example, with the renice command. We replace the body of the function set_user_nice(struct task_struct*, long) (located in kernel/sched.c) with a call to Bossa set priority function, rts_set_priority().

```
void set_user_nice(struct task_struct *p, long nice)
{
#ifdef CONFIG_BOSSA
  rts_set_priority(p, nice);
#else
  ...
  /* set_user_nice code */
  ...
#endif
}
```

4.9 Rts_yield

The function rts_yield(struct task_struct*) notifies Bossa that a process would like to voluntarily relinquish the processor. As done for the function rts_set_priority() (cf. Section 4.8), the body of sys_sched_yield() (located in kernel/sched.c) is replaced with a call to Bossa yield function, rts_yield(). We still keep the call to schedule() just before the function returns in order to let Bossa elect a new process.

```
/*** sys_sched_yield ***/
SYSCALL_DEFINE0(sched_yield)
{
#ifdef CONFIG_BOSSA
  rts_yield(current);
#else
  struct rq *rq = this_rq_lock();

  schedstat_inc(rq, yld_count);
  current->sched_class->yield_task(rq);

  /*
   * Since we are going to call schedule() anyway, there's
   * no need to preempt or enable interrupts:
   */
  __release(rq->lock);
  spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
  _raw_spin_unlock(&rq->lock);
  preempt_enable_no_resched();
#endif
  schedule();

  return 0;
}
```

4.10 Rts_schedule

The rts_schedule(struct task_struct*) function asks Bossa to elect a new task. In the Linux kernel, the schedule() function (located in kernel/sched.c) is called when such an election is needed. Rts_schedule() replaces the pick_next_task() function (cf. Section 3.7) in order to let Bossa choose the next process that will run on the processor.

```
asmlinkage void __sched schedule(void) {
  ...
  if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
    if (unlikely(signal_pending_state(prev->state, prev)))
      prev->state = TASK_RUNNING;
#ifndef CONFIG_BOSSA
    else
      deactivate_task(rq, prev, 1);
#endif
    switch_count = &prev->nvcsw;                                    10
  }

  pre_schedule(rq, prev);

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

#ifndef CONFIG_BOSSA
  put_prev_task(rq, prev);
  next = pick_next_task(rq);                                        20
#else
  unsigned long flags;
  spin_lock_irqsave(&bossa_scheduler_lock, flags); /* no point to this */
  next = rts_schedule(prev);
  clear_tsk_need_resched(prev);
  spin_unlock_irqrestore(&bossa_scheduler_lock, flags);
#endif

  if (likely(prev != next)) {
    ...                                                             30


}
```