Trykt af IMM, DTU

Appendix A

Project description

English title: Linux scheduling algorithems
Advisor: Jørgen Steensgaard-Madsen
Period: 1/1 - 11/10-2002
Credit: 45 ECTS points
Participant: Anders Fugmann

English project description

The goal will be pursued partly by studying the relevant literature on the subject of scheduling algorithms for multiprocessor systems generally, partly by studying the actual implementation in the Linux kernel. Based on this studies, evaluation-criteria will be defined, and by relating the actual implementation to the theoretical work some suggestions for improvements might result. In any case, it is the goal to find quantitative qualities that can be tested and measured to assess variations in algorithms.

Appendix B

Linux augmentation

This section lists the changes made to the Linux kernel in order to implement instrumentation.

arch/i386/config.in

At line 425 is inserted:

```
tristate '␣␣Scheduler␣statistics' CONFIG_DEBUG_STAT
```

Makefile

In the top-level Makefile for Linux, the versionnumber is changed by replacing line 4, with:

```
EXTRAVERSION = -stat
```

include/linux/sched.h

The structure `task_struct` has been extended, by inserting the folloing at line 413:


```
    /* time on run queue */
    unsigned long run_queue_time;

    /* Number of times a thread has been removed from the run queue */
    unsigned long preemptions;

    /* Total PE service time */
    unsigned long cpu_time;

    /* Number of times the process has been scheduled */
    unsigned long sched_count;

    /* Temporary variable holding the time of the last update of
       time values (run_queue_time and cpu_time) */
    unsigned long last_update;

    /* flag to see if the timestamps has been updated */
    unsigned long pe_add;

    /* Variables used to calculate process charictaristics */
    unsigned long intervals;
    unsigned long interval;
    unsigned long interval2;

    unsigned long added;
    unsigned long removed;
    unsigned long cpu_interval;
```

In function `del_from_runqueue`, at line 898 is inserted:

```
        p->preemptions++;

        p->run_queue_time += jiffies - p->last_update;
        p->last_update = -1;

        /* Update the CPU/(CPU+IO) stats */
        p->removed=jiffies;
```

linux/sched.c

At line 36 is inserted:

```
#include "../drivers/stat/stat.h"
```

The function `add_to_runqueue` at line 325 is replaced with:

```
static inline void add_to_runqueue(struct task_struct * p)
{
        unsigned long io,cpu,frac;
        list_add(&p->run_list, &runqueue_head);
```

```
        nr_running++;

        p->last_update = jiffies;

        /* Updated the stats, but not the firs runqueue add  */
        if (p->added < p->removed)
        {
            io=jiffies - p->removed;
            cpu=p->times.tms_stime - p->cpu_interval;
            frac=cpu*100/(cpu+io);
            p->interval+=frac;
            p->interval2+=frac*frac;
            p->intervals++;
            p->removed=jiffies;
            p->added=jiffies;
            p->cpu_interval=p->times.tms_stime;
        }
}
```

In function `schedule`, at line 697, is inserted:

```
        if (prev->pe_add != -1)
        {
            prev->cpu_time += jiffies - prev-> pe_add;
            prev->pe_add = -1;
            prev->sched_count++;
        }
        next->pe_add = jiffies;
```

drivers/stat/Makefile

```
#
# Makefile for the kernel stat device driver.
#
# Gather statistics information at each schedule,
# And at each block or nonblock
#
# Anders Fugmann (afu@fugmann.dhs.org)

# CONFIG_DEBUG_STAT can either be 'y' or undefined

O_TARGET := statistics.o
export-objs := stat.o
obj-$(CONFIG_DEBUG_STAT) += stat.o

include $(TOPDIR)/Rules.make

fastdep:
```

drivers/stat/stat.h


```
#ifndef __STAT_H__

#define __STAT_H__


#ifdef CONFIG_DEBUG_STAT

extern int stat_string(char* buffer, struct task_struct * p);
extern int stat_on(void);

#else

#define stat_string(buffer, p)  0
#define stat_on(void)           0

#endif /* CONFIG_DEBUG_STAT */


#endif /*  __STAT_H__ */
```

drivers/stat/stat.c

```
/*******************************************************************
 * Stat character driver
 * The driver creates a character device, to which the current
 * stat of all tasks is written
 *
 * Written by: Anders Fugmann
 *
 *******************************************************************/

#include <linux/kernel.h>
#include <linux/module.h>

#include <linux/proc_fs.h>
#include <linux/wait.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/kdev_t.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <linux/devfs_fs_kernel.h>
#include <linux/smp_lock.h>
#include <linux/spinlock.h>
#include <linux/ctype.h>
#include <asm/uaccess.h>
#include <linux/sched.h>

/* #include "stat.h" */

#define STAT_MINOR          1
/*==============================================================*
```

```
 * Forward declarations                                           *
 *================================================================*/

static int  __init stat_init(void);
static void __exit stat_exit(void);

static int    stat_open(struct inode *inode, struct file *file);
static int    stat_release(struct inode *inode, struct file *file);

/* File operations functions */
static loff_t stat_llseek(struct file *file, loff_t offset, int origin);
static int    stat_open(struct inode *inode, struct file *file);
static int    stat_release(struct inode *inode, struct file *file);

static ssize_t stat_read(struct file *file, char* buffer, size_t size,
                         loff_t *offset);

/* Entry end exit routines */
module_init(stat_init);
module_exit(stat_exit);

/*================================================================*
 * Variables                                                      *
 *================================================================*/
static devfs_handle_t devfs_handle;
static struct proc_dir_entry* proc_handle;

/* Major number */
static int major;

static volatile int stat_enabled;
static volatile int stat_open_lock = 0;
static volatile int stat_read_lock = 0;

/* File operations structure */
static struct file_operations stat_fops =
{
  THIS_MODULE,
  stat_llseek,
  stat_read,
  NULL, /* write nor allowed */
  NULL, /* readdir */
  NULL, /* poll */
  NULL, /* ioctl */
  NULL, /* mmap */
  stat_open,
  NULL, /* flush */
  stat_release,
  /* nothing more, fill with NULL's */
};


/*=====================================================================*
 * Functions begin                                                     *
 *=====================================================================*/

int stat_string(char* buffer, struct task_struct * p)
{
```



```
    unsigned long queue_time = p->run_queue_time;
    unsigned long cpu_time = p->cpu_time;
    if (p->last_update != -1)
        queue_time += jiffies - p->last_update;

    if (p->pe_add != -1)
        cpu_time += jiffies - p->pe_add;
    /* No divisions */
    /* unsigned long avg_wait_time = (100*wait_time)/p->times.tms_utime; */

    return
    sprintf(buffer,
            "tick:%lu␣cpu:%d␣pid:%d␣grp:%d␣policy:%lu␣"
            "nice:%lu␣rt_priority:%lu␣counter:%ld␣"
            "preemptions:%lu␣"
            "run_time:%lu␣"
            "run_queue_time:%lu␣"
            "cpu_time:%lu␣sched_count:%lu␣"
            "parent:%d␣"
            "time:%ld␣"
            "interval:%lu␣interval2:%lu␣intervals:%lu␣"
            "\n",
            jiffies, p->processor, p->pid, (int) p->pgrp, p->policy,
            p->nice, p->rt_priority, p->counter,
            p->preemptions,
            p->times.tms_utime + p->times.tms_stime,
            queue_time,
            cpu_time, p->sched_count,
            p->p_pptr->pid,
            (jiffies - p->start_time),
            p->interval, p->interval2, p->intervals);
}

int stat_on(void) {
    //return test_bit(0,&stat_open_lock);
    return stat_enabled;
}

/* procfs read/write operations */
int stat_proc_read(char *page, char **start, off_t offset, int count, int *eof, void *da
{
    if (count > 1 && offset == 0)
    {
        page[0] = '0' + stat_enabled;
        page[1] = 0;
    }

    *eof = 1;
    return 2;
}

int stat_proc_write(struct file *file, const char *buffer, unsigned long count, void *da
{
    char c;
    if (count > 0)
    {
        /* Read only one char */
        copy_from_user(&c,buffer,1);
```

```
        if (c == '0' || c == '1')
        {
            stat_enabled = c - '0';
            if (stat_enabled)
                printk(KERN_INFO "Stat:␣Statistics␣enabled.\n");
            else
                printk(KERN_INFO "Stat:␣Statistics␣disabled.\n");
        }
    }
    return 1;

}

static int __init stat_init(void)
{
    stat_enabled = 0;
    clear_bit(0, &stat_open_lock);
    clear_bit(0, &stat_read_lock);
    /* Create proc entry */
    proc_handle = create_proc_entry("sched_stat",
                                    S_IFREG | S_IRUGO | S_IWUGO,
                                    proc_root_driver);

     if (proc_handle) {
         proc_handle->read_proc=stat_proc_read;
         proc_handle->write_proc=stat_proc_write;
         printk(KERN_INFO "Stat:␣Registered␣proc␣device.\n");
     }


    major = devfs_register_chrdev(0, "stat", &stat_fops);
    printk(KERN_INFO "Stat:␣Registered␣with␣major␣number:␣%d\n", major);
    devfs_handle = devfs_mk_dir (NULL, "stat", NULL);
    devfs_register(devfs_handle, "sched",
                   DEVFS_FL_DEFAULT,
                   major, STAT_MINOR,
                   S_IFCHR | S_IRUGO | S_IWUGO, &stat_fops, NULL);
    /* Create the buffer */
    return 0;
}

static void __exit stat_exit(void)
{
    devfs_unregister(devfs_handle);
    remove_proc_entry("sched_stat", proc_root_driver);
    printk(KERN_INFO "Stat:␣Modules␣deregistered\n");
}

static int stat_open(struct inode *inode, struct file *file)
{
    /* We can only handle one minor number */
    if (MINOR(inode->i_rdev) != STAT_MINOR)
        return -ENXIO;

    /* Only one reader at a time */
    if (test_and_set_bit(0,&stat_open_lock)) return -EBUSY;
```



```
    /* All ok, return success */
    return 0;

}
static int stat_release(struct inode *inode, struct file *file)
{
    clear_bit(0, &stat_open_lock);
    return 0;
}

/*
 *  Seeking is not meaningfull for this type of device.
 */
static loff_t stat_llseek(struct file *file, loff_t offset, int origin)
{
  return -ESPIPE;
}

static ssize_t stat_read(struct file *file, char* buffer, size_t size,
                        loff_t *offset)
{
    int data_copied;
    char* local_buffer;
    int data_size;
    struct task_struct *p;

    if(test_and_set_bit(0, &stat_read_lock)) {
        //return -EBUSY;
    }
    local_buffer = kmalloc(size + 1024, GFP_KERNEL);
    if (!local_buffer) return -EAGAIN;

    /* Grab the run_queue_lock, disabling IRQ on the local
       CPU to avoid beeing interrupted in this process */
    /* IRQ's must be anabled at this time, so this is not dangerous */
    local_irq_disable();
    read_lock(&tasklist_lock);

    data_copied = 0;
    /* Dont go over the threshold */
    data_size = 0;
    for_each_task(p) {
        if (data_copied >= size)
            break;
        data_size = stat_string(&local_buffer[data_copied],p);
        data_copied += data_size;
        if (data_copied > size) {
            data_copied -= data_size;
            break;
        }
    }
    read_unlock(&tasklist_lock);
    local_irq_enable();
    /* Subtract offset, and add one for ending '0' */
    //data_copied++;

    /* Only copy data if there is data */
    if (data_copied > 0)
```

```
    {
        if(copy_to_user(buffer, local_buffer, data_copied) > 0)
            data_copied = -EFAULT;
    }
    else
        data_copied = 0;
    /* Clear the bit. */
    kfree(local_buffer);

    clear_bit(0, &stat_read_lock);
    return data_copied;

}

MODULE_LICENSE("GPL");
```

Appendix C

Simulator

C.1 Task description

Simulator control file for kernel compilation tests.

```
#define SCHED_OTHER             0
#define SCHED_FIFO              1
#define SCHED_RR                2

Globals: hz=100, cpus=2, io_resources=3, tasks=J, ticks=14000

# I/O-bound
process {
        pid:1, grp:1, nice:0, policy: 0, run: 0
        CPU Task:
        Fork Task: forkprob=(J,0) durration=(10,15) type='IO␣Task:␣cpu=(1,3),␣io=(50,100)'
}

process {
        pid:2, grp:1, nice:0, policy: 0, run: 0
        CPU Task:
        Fork Task: forkprob=(J,0) durration=(40,80) type='IO␣Task:␣cpu=(1,2),␣io=(5,10)'
}

process {
        pid:3, grp:3, nice:0, policy: 0, run: 0
        CPU Task:
        Fork Task: forkprob=(J,0) durration=(120,240) type='IO␣Task:␣cpu=(10,10),␣io=(5,0)'
}
```

Simulator control file for static tests.


```
#define SCHED_OTHER             0
#define SCHED_FIFO              1
#define SCHED_RR                2

Globals: hz=100, cpus=2, io_resources=5, tasks=5, ticks=10000

process {
        pid:1, grp:1, nice:0, policy: 0, run: 0
        CPU Task:
}

process {
        pid:2, grp:1, nice:0, policy: 0, run: 1
        CPU Task:
        IO Task: cpu=(80,0), io=(20,0)
}

process {
        pid:3, grp:1, nice:0, policy: 0, run: 1
        CPU Task:
        IO Task: cpu=(60,0), io=(40,0)
}

process {
        pid:4, grp:1, nice:0, policy: 0, run: 1
        CPU Task:
        IO Task: cpu=(40,0), io=(60,0)
}

process {
        pid:5, grp:1, nice:0, policy: 0, run: 1
        CPU Task:
        IO Task: cpu=(20,0), io=(80,0)
}
```

C.2 Source

Makefile

```
##########################################################################
# Final Thesis
#
# Makefile for simulator
# Source   :  $RCSfile: Makefile,v $
#
# To change the scheduling algorithem used,
# modify the SCHEDULER variable below.
#
# Anders Fugmann.
#
#########################################################################

CFLAGS = -O2 -Wall -g -march=athlon -mcpu=athlon

TARGET = simulate

SCHED_LINUX = sched
SCHED_TWO_LEVEL = sched-first_level sched-second_level
SCHED_MULTI_QUEUE = sched-multi_queue
SCHED_LOCAL_QUEUE = sched-local_queue
SCHED_ROUND_ROBIN = sched-round_robin

# This variable holds the names of the C++ files to be linked.
# Use this to change the scheduler used.
SCHEDULER = $(SCHED_LINUX)

FILES = main processes globals task \
        iotask cputask periodictask killtask forktask iosched \
        global_sched $(SCHEDULER)

SRCS = $(addsuffix .cc, $(FILES)) $(addsuffix .cc, $(SCHEDULER))

OBJS = $(addsuffix .o, $(FILES))

TASKS=tasks

.PHONY: all dep clean

all: $(TARGET)

$(TARGET): $(OBJS) Makefile
        g++ $(CFLAGS) -o $(TARGET) $(OBJS)

.cc.o:
        g++ $(CFLAGS) -c -o $@ $<
.c.o:
        gcc $(CFLAGS) -c -o $@ $<

dep:
        gccmakedep $(SRCS)
clean:
        rm -f *o *~ $(TARGET) data* *.ps core Makefile.bak \
        *ps *fig
```

cputask.cc

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/times.h>
#include <time.h>

#include "cputask.h"
#include "task.h"
#include "sched_interface.h"

CpuTask::CpuTask()
{
    //initialisation
    prev_cpu = -1;
    cpu_switches = 0;
}

TaskProp* CpuTask::read(char* line)
{
    if (strcmp(line, "CPU␣Task:") == 0)
    {
        return new CpuTask();
    }
    else
        return NULL;
}


void CpuTask::statechange(enum task_state state, Task *task)
{

    /* Remember what cpu we was running on. This will have some
       influence on the run_time (cold cache etc.) */

    /* We do not react on this. */
    /* A cpu task just wants to know how muck time it gets. */

    /* This should be called whenever state changes from running to active */
}

int CpuTask::tick(Task *task)
{
    /* Check if the task wants to wait on some IO, or other stuff */
    if ((task->state == TASK_RUNNING) && (task->processor != prev_cpu))
    {
        cpu_switches++;
        prev_cpu = task->processor;
    }
    return 0;
}

void CpuTask::print(int tick)
{
    printf("cpu_switches:%d␣", cpu_switches);
}

/* start a process */
```



```
void CpuTask::run()
{
    int i = 3;
    printf ("Pid:␣%d\t(CpuTask)\n", getpid());
    fflush(stdout);
    /* Just do some CPU work repeatably */
    while (1)
        i = i * i;
}
```

cputask.h

```
#ifndef __CPUTASK_H__
#define __CPUTASK_H__

#include "task.h"

class CpuTask : public TaskProp
{
 public:
    int run_time;
    int cpu_switches;
    int prev_cpu;
    int optim_time;
    int wait_time;
    int run_queue_time;

    void statechange(enum task_state state, Task *task);
    int tick(Task *task);
    void print(int tick);
    void run();

    static TaskProp* read(char* line);
    CpuTask();
};

#endif /* __CPUTASK_H__ */
```

forktask.cc

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/times.h>
#include <time.h>
#include <math.h>

#include "forktask.h"
#include "killtask.h"
#include "task.h"
#include "processes.h"
```

```
#include "global_sched.h"
#include "globals.h"

ForkTask::ForkTask(struct prop_struct *interval,
                   struct prop_struct *durration,
                   char *type)
{
    this->interval=interval;
    this->durration=durration;
    this->type=type;
    this->next_change=calc_prop(interval);

    forked_tasks = new list<Task*>;
    /* Ignore this pid as a live one */
    nr_tasks++;
}

TaskProp* ForkTask::read(char* line)
{
    struct prop_struct *interval = new prop_struct();
    struct prop_struct *durration = new prop_struct();


    char* type = new char[256];
    if (sscanf(line, "Fork␣Task:␣forkprob=(%d,%d)␣durration=(%d,%d)␣type=",
               &interval->mean, &interval->variance,
               &durration->mean, &durration->variance) == 4)
    {
        /* Read the type. */
        int start = strcspn(line, "'") + 1;
        int length = strcspn(&line[start], "'");
        strncpy(type, &line[start], length);
        return new ForkTask(interval, durration, type);
    }
    delete interval;
    delete durration;
    return NULL;
}


void ForkTask::statechange(enum task_state state, Task *task)
{
}

int ForkTask::tick(Task *task)
{
    int tasks = 0;
    /* Make sure that we have the expected number of tasks */
    list<Task*>::iterator iter;
    for (iter=forked_tasks->begin();iter != forked_tasks->end(); iter++)
        if ((*iter)->state != TASK_STOPPED)
            tasks++;
    /*else
      forked_tasks->remove(*iter); */

    while (next_change-tasks++ > 0)
    {
        /* Fork the process. Remember to add a terminating task */
```



```
        CpuTask *ct = new CpuTask();
        KillTask *kt = new KillTask(calc_prop(durration));
        Task* t = new Task(task->pid*1000+1+forks, task->grp,
                           task->nice, task->policy, 0);
        if (strlen(type) > 0)
        {
            TaskProp *tp = get_task_prop(type);
            if (tp == NULL)
                fprintf(stderr, "Error:␣Could␣not␣create␣Forked␣Task:␣'%s'\n"
                       , type);
            else
                t->add_task_prop(tp);

        }
        t->add_task_prop(ct);
        t->add_task_prop(kt);
        prop_list.push_back(ct);
        /* Let the simulator and system know of this process */
        global_add_task(t);
        forked_tasks->push_front(t);
        forks++;
    }
    next_change=calc_prop(this->interval);
    /* Suspend if running */
    if (task->state == TASK_RUNNING)
    {
        task->state = TASK_SUSPENDED;
        return 1 << task->processor;
    }
    return 0;
}

void ForkTask::print(int tick)
{
    float run_time=0;
    float efficiency=0;
    /* Go through all Cpu properties */
    CpuTask* ct;
    list<CpuTask*>::iterator iter;
    for (iter = prop_list.begin(); iter != prop_list.end(); iter++)
    {
        ct = (*iter);
        run_time += ct->run_time;
        efficiency += ct->run_time * 1.0 / ct->run_queue_time;
    }
    printf("fork_forked:%d␣fork_runtime:%f,␣fork_efficiency:%f␣",
           forks, run_time/forks, efficiency/forks);
}

/* start a process */
void ForkTask::run()
{
    /* This is gonna be hard */
}

int ForkTask::calc_prop(struct prop_struct *prop)
{
    if (prop->mean == -1) return INT_MAX; /* Never */
```

```
    float length = prop->mean;
    length += (2.0*random()/RAND_MAX-1.0) * prop->variance/2.0;
    return (int) round(length);
}

ForkTask::~ForkTask()
{
    /* Delete all tasks */

}
```

forktask.h

```
#ifndef __FORKTASK_H__
#define __FORKTASK_H__

#include "task.h"
#include "cputask.h"
struct prop_struct
{
    int mean;
    int variance;
};


class ForkTask : public TaskProp
{
 public:
    /* State attributes */
    struct prop_struct *interval;
    struct prop_struct *durration;
    char *type;
    int next_change;

    /* Statictics attributes */
    int forks;
    int terminations;
    list<CpuTask*> prop_list;
    list<Task*> *forked_tasks;

    /* Functions */
    ForkTask(struct prop_struct *interval,
             struct prop_struct *durration,
             char *type);
    virtual ~ForkTask();

    void statechange(enum task_state state, Task *task);
    int tick(Task *task);
    void print(int tick);
    void run();

    static TaskProp* read(char* line);
    int calc_prop(struct prop_struct *prop);
};
```



```
#endif /* __FORKTASK_H__ */
```

global_sched.cc

```
#include "global_sched.h"
#include "sched_interface.h"
#include "task.h"
#include "globals.h"

/* List of task sources */
list<struct task_source *> sources;

void global_add_task(Task* p)
{
    int id = p->grp;
    struct task_source *ts = NULL;
    /* Find the task source */
    list<struct task_source *>::iterator i;

    for (i=sources.begin();i != sources.end(); i++)
    {
        if ((*i)->id == id)
        {
            ts = *i;
            break;
        }
    }
    /* If none was found, create a new entry */
    if (i == sources.end())
    {
        ts = new struct task_source;
        ts->task_list = new list<Task*>;
        ts->id = id;
        sources.push_back(ts);
    }
    ts->task_list->push_back(p);
}

/* Make sure that we have excatly <tasks> running tasks */
/* If there is less, only one task is added */
void global_sched(int nr_tasks)
{
    Task* p;
    list<Task*>::iterator it;
    /* First find out if any new tasks are needed */
    for (it = tasks.begin();it != tasks.end(); it++)
        if ((*it)->state != TASK_STOPPED)
            nr_tasks--;

    //printf("### Need to add %d tasks\n", nr_tasks);

    /* Still tasks to add? */
    static list<struct task_source *>::iterator iter;
    if (iter == NULL)
        iter = sources.begin();
```

```
    list<Task*> *selected = NULL;
    while (nr_tasks-- > 0)
    {
        /* This implements a RR.
           To store where we are, a static variable is used */

        for(unsigned int i=0; i < sources.size(); i++)
        {
            iter++;

            if (iter == sources.end())
            {
                iter = sources.begin();
            }

            if ((*iter)->task_list->size() > 0)
            {
                selected = (*iter)->task_list;
                break;
            }
        }

        /* If no lists was found, stop processing */
        if (selected == NULL)
            return;

        p = selected->front();
        p->state = TASK_RUNNING;
        tasks.push_front(p);
        wake_up_process(p);

        selected->remove(p);
        selected = NULL;
    }
}
```

global_sched.h

```
#ifndef __GLOBAL_SCHED_H__
#define __GLOBAL_SCHED_H__


#include "task.h"
#include "main.h"

extern list<struct task_source *> sources;

struct task_source {
    int id;
    list<Task*> *task_list;
};

void global_add_task(Task* p);
void global_remove_task(Task* p, int id);
```



```
void global_sched(int tasks);

#endif /* __GLOBAL_SCHED_H__  */
```

globals.cc

```
#include "globals.h"
#include "task.h"

/* Number of PE's in the system */
int smp_num_cpus;

/* Number of ticks to simulate */
int ticks;

/* Number of IO resources */
int io_resources;

/* System freq */
int HZ;

/* Total number of tasks in the system */
int nr_tasks;

/* Every task that has ever exsisted in the simulation */
list<Task*> tasks;

/* Jiffies counter */
cycles_t jiffies;
```

globals.h

```
/* Global defines for the program */
#ifndef __GLOBALS_H__
#define __GLOBALS_H__

#include "task.h"
#define cycles_t unsigned int

extern int ticks;
extern int smp_num_cpus;
extern int io_resources;

extern int HZ;
extern int nr_tasks;

extern list<Task*> tasks;

extern cycles_t jiffies;

#endif /* __GLOBALS_H__ */
```

iosched.cc

```
#include <list>
#include <vector>
#include <algorithm>
#include "task.h"
#include "globals.h"
#include "iosched.h"


/* The io queue */
static list<Task*> io_queue;
/* List of tasks receiving IO */
static list<Task*> active;

/* Return true, if the thread is not already assigned to an IO resource */
int can_ioschedule(Task* task)
{
    return (!task_has_io(task));
}

/* Return true, if the process is retriving IO */
int task_has_io(Task* task)
{
    return (find(active.begin(), active.end(), task) != active.end());
}

/* Return true, if the task is waiting for IO */
int task_on_ioqueue(Task* task)
{
    return (find(io_queue.begin(), io_queue.end(), task) != io_queue.end());
}

/* Add a task to the IO-queue */
void add_task_ioqueue(Task* task)
{
    /* Add to tail */
    io_queue.push_back(task);
}

/* Remove a task to the IO-queue */
void remove_task_ioqueue(Task* task)
{
    io_queue.remove(task);
}

/* Schedule tasks waiting for IO */
/* Should be called for each tick */

void iosched(void)
{
    /* Remove all tasks from the active queue, that no longer
     * wants IO.
     * Also count the number of tasks receiving IO */

    int active_tasks = 0;
    list<Task*>::iterator iter;
    for (iter = active.begin(); iter != active.end(); iter++)
```



```
    {
        if ((*iter)->state == TASK_SUSPENDED)
            active_tasks++;
        else
        {
            //printf("### Removing pid from IO: %d\n",(*iter)->pid);
            active.remove(*iter);
            /* Remove also from the IO wait queue */
            io_queue.remove(*iter);
            iter--;
        }
    }

    /* Add upto io_resources to the list */
    while (active_tasks < io_resources)
    {
        for (iter = io_queue.begin(); iter != io_queue.end(); iter++)
        {
            /* Only schedule for IO, if it does not retrieve IO */
            if (can_ioschedule(*iter))
            {
                active.push_back(*iter);
                active_tasks++;
                break;
            }
        }
        if (iter == io_queue.end())
            break;
    }
}
```

iosched.h

```
#ifndef __IOSCHED_H__
#define __IOSCHED_H__

#include "task.h"

/* Return true, if the process is retriving IO */
int task_has_io(Task* task);

/* Return true, if the task is waiting for IO */
int task_on_ioqueue(Task* task);

/* Add a task to the IO-queue */
void add_task_ioqueue(Task* task);

/* Remove a task to the IO-queue */
void remove_task_ioqueue(Task* task);

/* Schedule tasks waiting for IO */
void iosched(void);

#endif /* __IOSCHED_H__ */
```

iotask.cc

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/times.h>
#include <time.h>
#include <stdio.h>
#include <math.h>

#include "iotask.h"
#include "task.h"
#include "sched_interface.h"
#include "globals.h"
#include "iosched.h"

IoTask::IoTask(struct prob_struct *cpu, struct prob_struct *io)
{
    prob_cpu = cpu;
    prob_io = io;

    cpu_length = prob_calc_length(prob_cpu);
    io_length =  prob_calc_length(prob_io);

    last_cpu = 0;
    last_io = 0;
}


IoTask::~IoTask()
{
    free(prob_cpu);
    free(prob_io);
}

/* Construct a ioprob from a string */
TaskProp* IoTask::read(char* line)
{
    prob_struct *cpu, *io;
    cpu = (prob_struct*) malloc(sizeof(prob_struct));
    io = (prob_struct*) malloc(sizeof(prob_struct));

    if (sscanf(line, "IO␣Task:␣cpu=(%d,%d),␣io=(%d,%d)",
               &cpu->mean, &cpu->variance, &io->mean, &io->variance) == 4)
        return new IoTask(cpu, io);
    else
    {
        free(cpu);
        free(io);
        return NULL;
    }

}

void IoTask::statechange(enum task_state state, Task *task)
{
}

int IoTask::tick(Task *task)
```



```
{
    /* Check if the task wants to wait on some IO, or other stuff */
    /* we use the statistics counter from task */
    switch (task->state)
    {
        case TASK_RUNNING:
            if (task->cpu_time - last_cpu >= cpu_length)
            {
                io_length = prob_calc_length(prob_io);
                task->state = TASK_SUSPENDED;
                /* Add to io_scheduler */
                add_task_ioqueue(task);
                last_io = task->io_time;
                return 1 << task->processor;
            }
            break;

        case TASK_SUSPENDED:
            if (task->io_time - last_io >= io_length)
            {
                cpu_length = prob_calc_length(prob_cpu);
                task->state = TASK_RUNNING;
                /* Remove from the io_scheduler */
                remove_task_ioqueue(task);
                last_cpu = task->cpu_time;
                wake_up_process(task);
                return 1 << task->processor;
            }
            break;

        default:
            break;
    }
    return 0;
}

void IoTask::print(int tick)
{
}

/* Calcualte the length of a event */
int IoTask::prob_calc_length(struct prob_struct *prob)
{
    if (prob->mean == -1) return INT_MAX;
    float length = prob->mean;
    length += (2.0*random()/RAND_MAX-1.0) * prob->variance/2.0;
    return lroundf(length);
}

/* start a process */
void IoTask::run()
{
    struct tms buf, nbuf;
    clock_t time, ntime;
    struct timespec req, rem;
    int cpu_length, io_length;
```

```
    printf ("Pid:␣%d\t(IoTask)␣(%d,␣%d)\n", getpid(), prob_cpu->mean, prob_io
         ->mean);
    fflush(stdout);

    while (1)
    {
        cpu_length = prob_calc_length(prob_cpu);
        /* First get the req cpu_time */
        time = times(&buf);
        ntime = times(&nbuf);

        while (nbuf.tms_utime - buf.tms_utime < cpu_length)
        {
            cputime((cpu_length - (nbuf.tms_utime - buf.tms_utime))
                  * 10000000);
            ntime = times(&nbuf);
        }
        ntime = times(&nbuf);
        /* Ok now sleep the io time */
        io_length =  prob_calc_length(prob_io);

        req.tv_sec = io_length/HZ;
        req.tv_nsec = (io_length%HZ) * 1000*1000*(1000/HZ);
        nanosleep(&req, &rem);
    }

}
```

iotask.h

```
#ifndef __IOTASK_H__
#define __IOTASK_H__

#include "task.h"

/* Struct to define io proberbility */
struct prob_struct {
    int mean; /* Avarage length in ticks. */
    int variance; /* mean_time + [-1;1]*variance/2.0 */
};

class IoTask : public TaskProp
{
 public:
    /* Prob for io operations. */
    struct prob_struct *prob_cpu;
    struct prob_struct *prob_io;

    /* Internal variables */
    int cpu_length;
    int io_length;
    int last_cpu;
    int last_io;
```



```
    void statechange(enum task_state state, Task *task);
    int tick(Task *task);
    void print(int tick);
    int prob_calc_length(struct prob_struct *prob);
    void run();

    static TaskProp* read(char *line);
    IoTask(struct prob_struct *cpu, struct prob_struct *io);
    virtual ~IoTask();
};

#endif /* __IOTASK_H__ */
```

killtask.cc

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/times.h>
#include <time.h>

#include "killtask.h"
#include "task.h"
#include "sched_interface.h"

KillTask::KillTask(int lifetime)
{
    this->lifetime = lifetime;
}

TaskProp* KillTask::read(char* line)
{
    int lifetime;

    if (sscanf(line, "Kill␣Task:␣lifetime=%d", &lifetime) == 1)
    {
        return new KillTask(lifetime);
    }
    return NULL;
}


void KillTask::statechange(enum task_state state, Task *task)
{
    /* Nothing to do */
}

int KillTask::tick(Task *task)
{
    /* Do only if we are running */
    if (task_has_cpu(task)) lifetime--;
    if (lifetime <= 0)
    {
        /* printf("### Stopping task: %d\n", task->pid); */
        task->state = TASK_STOPPED;
        return 1;
```

```
    }
    return 0;
}
void KillTask::print(int tick)
{
    /* No extra output */
}

/* start a process */
void KillTask::run()
{
    /* Not implemented */
}
```

killtask.h

```
#ifndef __KILLTASK_H__
#define __KILLTASK_H__

#include "task.h"

class KillTask : public TaskProp
{
 public:
    /* State attributes */
    int lifetime;

    KillTask(int lifetime);

    void statechange(enum task_state state, Task *task);
    int tick(Task *task);
    void print(int tick);
    void run();

    static TaskProp* read(char* line);
};

#endif /* __KILLTASK_H__ */
```

main.cc

```
#include <list>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include "task.h"
#include "sched_interface.h"
#include "simulator_interface.h"
#include "iosched.h"
#include "globals.h"
#include "processes.h"
```



```
#include "global_sched.h"
#include "main.h"

using namespace std;

int current_processor;

int smp_processor_id()
{
    return current_processor;
}

cycles_t get_cucles()
{
    return 0;
}

void smp_send_reschedule(int cpu)
{
    /* Normally we need to send a request to the other CPU */
    int old_cpu = current_processor;
    current_processor=cpu;
    schedule();
    current_processor=old_cpu;
}


void timer_tick()
{
    jiffies++;
    for (int cpu=0;cpu<smp_num_cpus;cpu++)
    {
        current_processor=cpu;
        do_timer();
        if (get_current()->need_resched)
            schedule();
    }
    iosched();
}

/* Print out all statistics */
void statistics(list<Task*> *task_list, int tick)
{
    list<Task*>::iterator iter;
    for (iter = task_list->begin();iter != task_list->end(); iter++)
    {
        (*iter)->print(tick);
    }
}

void tick(int ticks, list<Task*> *task_list, int print)
{

    /* Send this many ticks */
    for (int ttick=1;ttick<=ticks;ttick++)
    {
        /* Make sure that the tasks are there */
        global_sched(nr_tasks);
```

```
        timer_tick();

        /* Send a tick to all tasks */
        list<Task*>::iterator iter;
        for (iter = task_list->begin();iter != task_list->end(); iter++)
        {
            /* Call schedule on the current cpu? */
            /* Dont touch stopped_tasks */
            if ((*iter)->state == TASK_STOPPED) continue;

            unsigned int sched_cpus = (*iter)->tick();

            if (sched_cpus)
            {
                /* Schedule on selected  cpu's */
                for (int i=0;i<smp_num_cpus;i++)
                    if (sched_cpus | (1 << i))
                    {
                        current_processor = i;
                        schedule();
                        iosched();
                    }
            }
        }
        /* Glocal scheduler */
        //global_sched(nr_tasks);
        if (print)
            statistics(task_list, ttick);
    }
}

void run_tasks(list<Task*> l)
{
    list<Task*>::iterator iter;
    printf("Starting␣jobs.\n");
    for (iter = l.begin();iter != l.end(); iter++)
    {
        (*iter)->run();
    }
    fflush(stdout);
    wait(NULL);
}

int main(int argc, char *argv[])
{
    int cpu=0;
    int last = 0;
    list<Task*> *task_list;
    if (argc == 1)
    {
        printf("Usage:␣%s␣<task_file>␣[run]␣[last]\n",argv[0]);
        printf("␣␣task_file:␣A␣file␣containing␣the␣definition␣of␣"
               "tasks␣to␣simulate.\n");
        printf("␣␣run:␣␣␣␣␣␣Given␣this␣keyword,␣the␣tasks␣will␣"
               "be␣executed\n");
        printf("␣␣last:␣␣␣␣␣Only␣print␣out␣final␣stats\n");
        return 1;
    }
```



```
    task_list = read_tasks(argv[1]);

    /* Should the tasks be started on the OS? */

    if (argc >= 3 && !strcmp("run", argv[2]))
    {
        run_tasks(*task_list);
        return 0;
    }

    if (argc >= 3 && !strcmp("last", argv[2]))
        last = 1;

    /* Begin schedule all tasks */
    list<Task*>::iterator iter;

    for (iter = task_list->begin();iter != task_list->end(); iter++)
    {
        if ((*iter)->pid >= 0)
            global_add_task(*iter);
        else /* Idle_task */
        {
            /* No stats can be retrieved for those */
            current_processor = cpu++;
            sched_init(*iter);
            tasks.push_front(*iter);
            nr_tasks++;

        }
    }
    /* Lets start the clock */
    tick(ticks, &tasks, !last);

    if (last)
    {
        statistics(&tasks, ticks);
    }
}
```

main.h

```
#ifndef __MAIN_H_
#define __MAIN_H_

#endif /* __MAIN_H_ */
```

periodictask.cc

```
#include <sys/types.h>
#include <unistd.h>
```

```
#include <sys/times.h>
#include <time.h>

#include "periodictask.h"
#include "task.h"
#include "sched_interface.h"
#include "globals.h"
#include <math.h>


PeriodicTask::PeriodicTask(int period, int cpu_time)
{
    this->period = period;
    this->cpu_time = cpu_time;

    /* Internal variables */
    period_left=0, cpu_left=cpu_time, cpu_overdue=0, max_cpu_overdue=0;
}

/* Construct from a string */
TaskProp* PeriodicTask::read(char* line)
{
    int period, cpu;

    if (sscanf(line, "Periodic␣Task:␣period=%d,␣cpu=%d", &period, &cpu) == 2)
        return new PeriodicTask(period, cpu);
    else
        return NULL;
}

void PeriodicTask::statechange(enum task_state state, Task *task)
{
}

int PeriodicTask::tick(Task *task)
{
    if (task->state == TASK_STOPPED) return 0;
    --period_left;

    if (task_has_cpu(task) && --cpu_left <= 0) /* Implies task->state !=
         TASK_STOPPED */
    {
        /* Now sleep */
        task->state = TASK_SUSPENDED;
        return 1 << task->processor;
    }

    /* New period */
    if (period_left <= 0)
    {
        /* Remember cpu overdue */
        cpu_overdue += cpu_left;
        if (cpu_left > max_cpu_overdue)
            max_cpu_overdue = cpu_left;

        cpu_left = cpu_time;
        period_left = period;
```



```
        task->state = TASK_RUNNING;
        wake_up_process(task);
    }

    return 0;
}

void PeriodicTask::print(int tick)
{
    printf("period_left:%d␣per_cpu_left:%d␣per_cpu_overdue:%d␣
        per_max_cpu_overdue:%d␣",
          period_left, cpu_left, cpu_overdue, max_cpu_overdue);
}

void PeriodicTask::run()
{
    struct tms buf, nbuf;
    clock_t time, ntime;
    struct timespec req, rem;
    long sleep_time;

    printf ("Pid:␣%d\t(PeriodicTask)\n", getpid());
    fflush(stdout);

    while (1)
    {
        /* First get the req cpu_time */
        time = times(&buf);
        ntime = times(&nbuf);

        while (nbuf.tms_utime - buf.tms_utime < cpu_time )
        {
            cputime((cpu_time - (nbuf.tms_utime - buf.tms_utime)) * 10000000)
                ;
            ntime = times(&nbuf);
        }
        ntime = times(&nbuf);
        /* Ok now sleep untill the period is over */
        sleep_time = period - (ntime - time);
        req.tv_sec = sleep_time/HZ;
        req.tv_nsec = (sleep_time%HZ)*1000000000/HZ;
        nanosleep(&req, &rem);
    }

}
```

periodictask.h

```
#ifndef __PERIODICTASK_H__
#define ___PERIODICTASK_H__

#include "task.h"

class PeriodicTask : public TaskProp
{
```

```
 public:
    int cpu_time;
    int period;

    /* Internal variables */
    int period_left;
    int cpu_left;

    /* Statistics variables */
    int max_cpu_overdue;
    int cpu_overdue;

    void statechange(enum task_state state, Task *task);
    int tick(Task *task);
    void print(int tick);
    void run();


    static TaskProp* read(char *line);
    PeriodicTask(int period, int cpu_time);
};

#endif /* ___PERIODICTASK_H__ */
```

processes.cc

```
#include <list>
#include <stdio.h>
#include "globals.h"
#include "task.h"
#include "iotask.h"
#include "cputask.h"
#include "periodictask.h"
#include "processes.h"
#include "killtask.h"
#include "forktask.h"

using namespace std;

/* Read all processes from a file
 * The structure if the file is defined by each subclass
 * A line is scanned, and given to all known modules, and if one
 * recognices it, it returns a class to be added to the property list
 */

char* read_line(char* buf, int len, FILE* f)
{
    do {
        if (!fgets(buf, len, f))
            return NULL;
    } while (buf[0] == '#');

    /* Remove all tabs and spaces */
    int i=0;
```



```
    while (buf[i]==' ' || buf[i]=='\t') i++;
    if (buf[strlen(buf)-1] == '\n')
        buf[strlen(buf)-1] = 0;
    return &buf[i];
}

list<Task*> *read_tasks(char* file_name)
{
    int ln = 0;
    list<Task*> *l = new list<Task*>;
    Task *t;
    char buf[255];
    char* line;


    /* Read in the globals */
    FILE* file;
    if (!(file = fopen(file_name, "r")))
    {
        fprintf(stderr,"Fatal: Unable to open file: %s\n", file_name);
        exit(-1);
    }

    while ((line = read_line(buf, 255, file)))
    {
        if (line[0] != 0) break;
    }

    if (sscanf(line, "Globals: hz=%d, cpus=%d, io_resources=%d, "
                     "tasks=%d, ticks=%d",
                     &HZ, &smp_num_cpus, &io_resources, &nr_tasks, &ticks) != 5)
    {
        fprintf(stderr, "Fatal: Globals must be defined\n");
        exit(-1);
    }

    /* Create idle_tasks */
    for (int i=1; i <= smp_num_cpus;i++)
    {
        t = new Task(-i,0,0,0,-1);
        t->add_task_prop(new CpuTask());
        (*l).push_front(t);
    }


    /* For each line */
    while ((line = read_line(buf, 255, file)))
    {
        ln++;

        /* Read "process {" */
        if (strcmp("process {", line) == 0)
        {
            /* Found a process entry
             * Start processing */
            if (!(line = read_line(buf, 255, file)))
                printf("### Missing process parameters on line: %d\n", ln);
            ln++;
```

```
            if ((t = Task::read(line)))
            {
                for (;;)
                {
                    TaskProp* tp;
                    if (!(line = read_line(buf, 255, file)))
                    {
                        printf("###␣Missing␣'}'␣on␣line:␣%d\n", ln);
                        break;
                    }
                    ln++;
                    if (line[0] == '}') break;

                    /* Let all properties read the line */
                    tp = get_task_prop(line);
                    if (!tp) {
                        printf("###␣Unknow␣property␣'%s'␣on␣line:␣%d\n", line
                             , ln);
                        continue;
                    }
                    t->add_task_prop(tp);
                }
            }
            else
                printf("###␣Could␣not␣read␣task␣definitions\n");
            (*l).push_front(t);
        } else
            if (line[0] != 0) printf("###␣Unknown␣line(%d):␣'%s'\n", ln, line
                 );
    }
    fclose(file);

    printf("###␣Found␣and␣scanned␣%d␣tasks\n", (*l).size()-smp_num_cpus);

    return l;
}

TaskProp* get_task_prop(char* line) {
    TaskProp* tp;
    if ((tp = IoTask::read(line))) return tp;
    if ((tp = CpuTask::read(line))) return tp;
    if ((tp = PeriodicTask::read(line))) return tp;
    if ((tp = KillTask::read(line))) return tp;
    if ((tp = ForkTask::read(line))) return tp;
    fprintf(stderr, "Error:␣Cannot␣create␣property:␣%s\n", line);
    return NULL;
}
```

processes.h

```
#ifndef __PROCESSES_H__
#define __PROCESSES_H__

#include <list>
```



```
using namespace std;

list<Task*> *read_tasks(char* file_name);
TaskProp* get_task_prop(char *line);
#endif /* __PROCESSES_H__ */
```

sched-first_level.cc

```
#include <list>
#include <vector>
#include <math.h>
#include <algorithm>
#include "sched-first_level.h"
#include "task.h"
#include "globals.h"
#include "sched_interface.h"
#include "simulator_interface.h"
#include "sched.h"

/* List of task in the 2. level scheduler */
static list<Task*> *expired;
static list<Task*> *ready;


/* Utilization counters */
/* We have: IO Length, CPU Length. CPU = CPU/CPU+IO IO=IO/IO+CPU */

static int u_cpu;
static int u_io;
static int last_swap;

void add_utilization(Task *t)
{
    if (t->util_add==1) return;
    if (t->cpu_length > 0 || t->io_length >= 0)
    {
        u_cpu += 100*t->cpu_length/(t->cpu_length+t->io_length);
        u_io += 100*t->io_length/(t->cpu_length+t->io_length);
    }
    t->util_add = 1;
}

void sub_utilization(Task *t)
{
    if (t->util_add==0) return;
    if (t->cpu_length > 0 || t->io_length >= 0)
    {
        u_cpu -= 100*t->cpu_length/(t->cpu_length+t->io_length);
        u_io -= 100*t->io_length/(t->cpu_length+t->io_length);
    }
    t->util_add = 0;
}


void update_task_io(Task* t)
```

```
{
    if (t->io_start > -1)
        t->io_length += jiffies - t->io_start;

    /* And CPU starts. */
    t->cpu_start = jiffies;
}

void update_task_cpu(Task* t)
{
    if (t->cpu_start > -1)
        t->cpu_length += jiffies - t->cpu_start;

    /* And IO starts */
    t->io_start = jiffies;
}


Task* get_task()
{
    list<Task*>::iterator iter;
    int u_sys, u_best;
    Task* next = NULL;
    /* Choose a new task */
    u_sys=100*u_cpu/(u_cpu+u_io+1);  /* Range [0..100] */
    u_best=101;
    int u_task;

    /* Find a task in the ready queue,
       with stats as close as possible to uSYS.
       By tests above the queue cannot be empty. */

    for (iter = ready->begin();iter != ready->end(); iter++)
    {
        if ((*iter)->cpu_length > 0)
            u_task = 100*(*iter)->cpu_length/
                ((*iter)->cpu_length+(*iter)->io_length);
        else
            u_task=START_UTIL;

        if (abs(u_task-u_sys) < u_best)
        {
            next = *iter;
            u_best = abs(u_task-u_sys);
        }
    }
    return next;
}

void swap_queues()
{
    list<Task*> *tmp = ready;
    ready = expired;
    expired = tmp;
    last_swap=jiffies;
}

void high_level_stats()
```



```
{
    printf("###␣Tasks␣on␣ready␣queue:␣");
    list<Task*>::iterator iter;
    for (iter = ready->begin(); iter != ready->end(); iter++)
        printf("%d␣", (*iter)->pid);
    printf("\n");

    printf("###␣Tasks␣on␣expired␣queue:␣");
    for (iter = expired->begin(); iter != expired->end(); iter++)
        printf("%d␣", (*iter)->pid);
    printf("\n");

    printf("###␣Utilization.␣cpu:%d,␣io:%d\n",
           u_cpu, u_io);

}

/* Schedule processes to the low level scheduler,
   until the threasholds are up */
void fill()
{
    Task* next;
    while ((u_cpu < IO_THRESHOLD) || (u_cpu < CPU_THRESHOLD) ||
           (run_queue.size()==0) )
    {
        if (ready->size() == 0)
        {
            if (expired->size() > 0)
                swap_queues();
            else
            {
                /* Break, since there are no process to schedule */
                break;
            }
        }
        /* Select a new task */
        next = get_task();
        if (next == NULL) break;
        next->sched_stamp=jiffies;
        next->cpu_start = jiffies;
        ready->remove(next);
        add_utilization(next);
        sc_add_to_runqueue(next);
    }
    //high_level_stats();
}

/* This function is called whenever a process enters the runqueue,
   and is the base for all desicions. This is where processes
   are trapped by the first level scheduler */

void add_to_runqueue(Task* t)
{
    if (task_on_runqueue(t)) return;

    if (t->io_length < 1)
    {
        /* New process */
```

```
        t->io_start=jiffies;
        t->io_length=1;
        t->cpu_length=1;
        expired->push_back(t);
        fill();
        return;
    }

    t->cpu_start=jiffies;

    /* Has the task expired? */
    if (jiffies - t->sched_stamp > TASK_THRESHOLD)
    {
        sub_utilization(t);
        expired->push_back(t);
        fill();
        return;
    }

    /* Update global utilization counters */
    sub_utilization(t);
    update_task_io(t);
    add_utilization(t);
    sc_add_to_runqueue(t);
}


/* Remove a tasks from the rq.
   The function updates the task stats,
   and if the task has been terminated, fill is called */

void del_from_runqueue(Task* t)
{
    sc_del_from_runqueue(t);

    if (t->state == TASK_STOPPED)
    {
        //printf("### Task is stopped: %d\n", t->pid);
        sub_utilization(t);
        fill();
        return;
    }
    /* Recalcualte task info */
    sub_utilization(t);
    update_task_cpu(t);
    add_utilization(t);
    t->io_start=jiffies;
}

/* For each tick, this functions is called.
   It periodically tests for expired tasks, and moves them to the
   expired queue. It then add new tasks to the queue from the ready task
   list */

void high_level_tick()
{
    Task* t;
    list<Task*>::iterator iter;
```



```
    /* If the period has not yet expired */
    if (jiffies - last_swap < HIGH_LEVEL_PERIOD)
        return;

    last_swap = jiffies;

    for (iter = run_queue.begin();iter != run_queue.end(); iter++)
    {
        t = *iter;
        if ((jiffies - t->sched_stamp) > TASK_THRESHOLD)
        {
            /* Remove the process from the rq */
            iter=run_queue.erase(iter);

            /* Preempt the process, if needed */
            if (task_has_cpu(t))
                smp_send_reschedule(t->processor);

            sub_utilization(t);

            update_task_cpu(t);
            expired->push_back(t);
        }
    }
    fill();
}

void high_level_init()
{
    expired = new list<Task*>;
    ready = new list<Task*>;
    u_cpu=0;
    u_io=0;
    last_swap = 0;
}
```

sched-first_level.h

```
#ifndef __TWO_LEVEL_H__
#define __TWO_LEVEL_H__

#include "task.h"

#define IO_THRESHOLD 200*smp_num_cpus
#define CPU_THRESHOLD 200*smp_num_cpus
#define HIGH_LEVEL_PERIOD 200
#define TASK_THRESHOLD 400
#define START_UTIL 100

/* From low_level_sched */
extern list<Task*> run_queue;

void update_task_io(Task* t);
void update_task_cpu(Task* t);
```

```
Task* get_task();
void swap_queues();
void fill();
void add_to_runqueue(Task* t);
void del_from_runqueue(Task* t);
void high_level_tick();
void high_level_init();

void sc_add_to_runqueue(Task* t);
void sc_del_from_runqueue(Task* t);

#endif /* __TWO_LEVEL_H__ */
```

sched-local_queue.cc

```
#include <list>
#include <vector>
#include <math.h>
#include <algorithm>
#include "simulator_interface.h"
#include "sched_interface.h"
#include "sched.h"
#include "globals.h"

using namespace std;

/* A list of running programs on the cpu's */
Task ** current;
Task ** idle_tasks;
/* Per CPU data */
struct schedule_data *cpu_schedule_data;

/* The run queues */
list<Task*> *run_queues;
/* Number of running processes */
int nr_running = 0;

#define PROC_CHANGE_PENALTY 0
#define UPPER_THRESHOLD 100+100/smp_num_cpus
#define LOWER_THRESHOLD (100-100/smp_num_cpus)

#define unlikely(x) x
/* Actually this is the original define */
#define prepare_to_switch()   do { } while(0)

/* Assembler macro - This sets up the new task.
 * This is not needed in this simulator */
#define switch_to(prev,next,last) do { } while(0)

/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
```



```
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x)   ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x)   ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x)   (x)
#elif HZ < 1600
#define TICK_SCALE(x)   ((x) << 1)
#else
#define TICK_SCALE(x)   ((x) << 2)
#endif

#define NICE_TO_TICKS(nice)     (TICK_SCALE(20-(nice))+1)

/* Move a task from one run_queue to another.
   Make sure that the task is not in the running state */
void move_task(int from_rq, int to_rq)
{
    /* Select a task to be moved */
    list<Task*>::iterator iter;
    Task *p = NULL;
    for (iter = run_queues[from_rq].begin(); iter != run_queues[from_rq].end
         (); iter++)
    {
        if (can_schedule((*iter), to_rq))
        {
            p = *iter;
            break;
        }
    }
    if (p == NULL)
        return;


    /* Move the selected task */
    run_queues[from_rq].remove(p);
    p->run_queue = to_rq;
    run_queues[to_rq].push_back(p);
}


/* This function is used to load balance the run_queues */
/* The function only balance the current run_queue against others. */

void balance()
{

    /* printf("### Balance\n"); */
    int this_rq = smp_processor_id(), rq;
    int imbalance = 0;
    int i;

    int local = run_queues[this_rq].size()*100;
```

```
    int avg = nr_running*100/smp_num_cpus;

    /* Test is an imbalance may exist */
    /* if (abs(avg-local) <= 1)
        return;
    */
    /* imbalance = (local - avg) * 100 / (local + avg); */
    imbalance = local-avg;

    if (abs(imbalance) < UPPER_THRESHOLD)
        return;

    /* Need to balance */
    /* Do this while above the LOWER_THRESHOLD */
    while (abs(imbalance) > LOWER_THRESHOLD)
    {
        /* Need to steal processes from other RQ's? */
        if (imbalance < 0)
        {
            rq = this_rq;
            for (i=0;i<smp_num_cpus;i++)
            {
                if (run_queues[i].size() >  run_queues[rq].size())
                    rq = i;
            }

            /* rq points to the longest queue */

            /* Steal the task in the back of the queue */
            move_task(rq, this_rq);

        }

        if (imbalance > 0)
        {
            rq = this_rq;
            for (i=0;i<smp_num_cpus;i++)
            {
                /* printf("### Queue: %d, Length: %d\n",
                   i,run_queues[i].size()); */
                if (run_queues[i].size() < run_queues[rq].size())
                    rq = i;
            }

            /* rq points to the shortest queue */

            /* Push a task to the shortest rq */
            move_task(this_rq, rq);
        }

        /* Recalculate imbalance */
        local = run_queues[this_rq].size()*100;

        /* if (abs(avg-local) <= 1)
            return;
            imbalance = (local - avg) * 100 / (local + avg); */
        imbalance = local-avg;
    }
```



```
}


int preemption_goodness(Task *prev, Task *p, int cpu)
{
    return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->
         active_mm);
}

void reschedule_idle(Task *p)
{
        int this_cpu = smp_processor_id();
        Task *tsk, *target_tsk;
        int cpu, best_cpu, i, max_prio;
        cycles_t oldest_idle;

        /*
         * shortcut if the woken up task's last CPU is
         * idle now.
         */
        best_cpu = p->processor;
        if (can_schedule(p, best_cpu)) {
                tsk = idle_task(best_cpu);
                if (cpu_curr(best_cpu) == tsk) {
                        int need_resched;
send_now_idle:
                        /*
                         * If need_resched == -1 then we can skip sending
                         * the IPI altogether, tsk->need_resched is
                         * actively watched by the idle thread.
                         */
                        need_resched = tsk->need_resched;
                        tsk->need_resched = 1;
                        if ((best_cpu != this_cpu) && !need_resched)
                                smp_send_reschedule(best_cpu);
                        return;
                }
        }

        /*
         * We know that the preferred CPU has a cache-affine current
         * process, lets try to find a new idle CPU for the woken-up
         * process. Select the least recently active idle CPU. (that
         * one will have the least active cache context.) Also find
         * the executing process which has the least priority.
         */
        oldest_idle = (cycles_t) -1;
        target_tsk = NULL;
        max_prio = 0;

        for (i = 0; i < smp_num_cpus; i++) {
                cpu = cpu_logical_map(i);
                if (!can_schedule(p, cpu))
                        continue;
                tsk = cpu_curr(cpu);
                /*
                 * We use the first available idle CPU. This creates
                 * a priority list between idle CPUs, but this is not
```

```
                 * a problem.
                 */
                if (tsk == idle_task(cpu)) {
/* Added in 2.4.17
#if defined(__i386__) && defined(CONFIG_SMP)
                        *//*
                         * Check if two siblings are idle in the same
                         * physical package. Use them if found.
                         *//*
                         if (smp_num_siblings == 2) {
                                if (cpu_curr(cpu_sibling_map[cpu]) ==
                                    idle_task(cpu_sibling_map[cpu])) {
                                        oldest_idle = last_schedule(cpu);
                                        target_tsk = tsk;
                                        break;
                                }

                        }
#endif
*/
                        if (last_schedule(cpu) < oldest_idle) {
                                oldest_idle = last_schedule(cpu);
                                target_tsk = tsk;
                        }
                } else {
                        if (oldest_idle == -1ULL) {
                                int prio = preemption_goodness(tsk, p, cpu);

                                if (prio > max_prio) {
                                        max_prio = prio;
                                        target_tsk = tsk;
                                }
                        }
                }
        }
        tsk = target_tsk;
        if (tsk) {
                if (oldest_idle != -1ULL) {
                        best_cpu = tsk->processor;
                        goto send_now_idle;
                }
                tsk->need_resched = 1;
                if (tsk->processor != this_cpu)
                        smp_send_reschedule(tsk->processor);
        }
        return;
}

/*
 * Wake up a process. Put it on the run-queue if it's not
 * already there.  The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you're allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
static inline int try_to_wake_up(Task *p, int synchronous)
{
```



```
    //unsigned long flags;
        int success = 0;

        /*
         * We want the common case fall through straight, thus the goto.
         */
        /* spin_lock_irqsave(&runqueue_lock, flags); */
        p->state = TASK_RUNNING;
        if (task_on_runqueue(p))
                goto out;
        add_to_runqueue(p);
        if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
                reschedule_idle(p);
        success = 1;
out:
        /* spin_unlock_irqrestore(&runqueue_lock, flags); */
        return success;
}

int wake_up_process(Task * p)
{
        return try_to_wake_up(p, 0);
}


/* Calculate the goodness of the process */
int goodness(Task* p, int this_cpu, int this_mm)
{
    /* This is soly based on quantum and nice-level */
    int weight = -1;
    if (p->policy & SCHED_YIELD)
        return weight;

    if (p->policy == SCHED_OTHER) {
        /*
         * Give the process a first-approximation goodness value
         * according to the number of clock-ticks it has left.
         *
         * Don't do any other calculations if the time slice is
         * over..
         */
        weight = p->counter;
        if (!weight)
            return weight;

        if (p->processor == this_cpu)
            weight += PROC_CHANGE_PENALTY;

        /* .. and a slight advantage to the current MM */


        if (p->mm == this_mm || !p->mm)
            weight += 1;


        weight += 20 - p->nice;

        return weight;
```

```
    }

    /*
     * Realtime process, select the first one on the
     * runqueue (taking priorities within processes
     * into account).
     */
    weight = 1000 + p->rt_priority;
    return weight;
}

void __schedule_tail(Task* prev)
{
    task_release_cpu(prev);
}



void schedule_tail(Task* prev)
{
    __schedule_tail(prev);
}



void schedule()
{
    int this_cpu = smp_processor_id();
    list<Task*> run_queue = run_queues[this_cpu];
    struct schedule_data * sched_data;
    Task *prev, *next;
    int c;

    /*printf("### Sched - Queue %d, Size. %d Total: %d\n",
      this_cpu, run_queue.size(), nr_running);*/

need_resched_back:

    prev = get_current();
    next = idle_task(this_cpu);


    /* This is not modeled in the simulator
    if (unlikely(in_interrupt())) {
        printk("Scheduling in interrupt\n");
        BUG();
    }
    */

    /* release_kernel_lock(prev, this_cpu); */

    /*
     * 'sched_data' is protected by the fact that we can run
     * only one process per CPU.
     */
    sched_data = &cpu_schedule_data[this_cpu];

    /* spin_lock_irq(&runqueue_lock); */
```



```
    /* move an exhausted RR process to be last.. */
    if (unlikely(prev->policy == SCHED_RR))
        if (!prev->counter) {
            prev->counter = NICE_TO_TICKS(prev->nice);
            move_last_runqueue(prev);
        }
    /*
    switch (prev->state) {
        case TASK_INTERRUPTIBLE:
            if (signal_pending(prev)) {
                prev->state = TASK_RUNNING;
                break;
            }
        default:
             del_from_runqueue(prev);
        case TASK_RUNNING:;
    }
    */
    /* Since tasks themself set the state running, no need to check for
     * a pending signal. All in all this boild down to the following: */
    if (prev->state != TASK_RUNNING)
        del_from_runqueue(prev);

    prev->need_resched = 0;

    /*
     * this is the scheduler proper:
     */

repeat_schedule:
    /*
     * Default process to select..
     */
    c = -1000;
    list<Task*>::iterator iter;

    for (iter = run_queue.begin(); iter != run_queue.end(); iter++)
    {
        if (can_schedule((*iter), this_cpu)) {
            int weight = goodness((*iter), this_cpu, prev->active_mm);
            if (weight > c)
                c = weight, next = (*iter);
        }
    }

    /* Do we need to re-calculate counters? */
    if (!c) {
        /* Recalculate all counter on all tasks. */
        list<Task*>::iterator p;
        /* spin_unlock_irq(&runqueue_lock);
         * read_lock(&tasklist_lock);
         */
        for_all_tasks(p)
            if ((*p)->run_queue == this_cpu)
                (*p)->counter = ((*p)->counter >> 1) +
                    NICE_TO_TICKS((*p)->nice);
```

```
        /*
         * read_unlock(&tasklist_lock);
         * spin_lock_irq(&runqueue_lock);
         */
        goto repeat_schedule;
    }

    /*
     * from this point on nothing can prevent us from
     * switching to the next task, save this fact in
     * sched_data.
     */

    /* Add the task to the running list */
    /* Mark which cpu the task is running on */

    /* Sched data is locale for the cpu itself */

    //sched_data->curr = next;
    schedule_tail(prev);
    task_set_cpu(next, this_cpu);
    /* spin_unlock_irq(&runqueue_lock); */

    if (unlikely(prev == next)) {
        /* We won't go through the normal tail, so do this by hand */
        prev->policy &= ~SCHED_YIELD;
        goto same_process;
    }

/*#ifdef CONFIG_SMP*/
    /*
     * maintain the per-process 'last schedule' value.
     * (this has to be recalculated even if we reschedule to
     * the same process) Currently this is only used on SMP,
     * and it's approximate, so we do not have to maintain
     * it while holding the runqueue spinlock.
     */

//    sched_data->last_schedule = get_cycles();

    /*
     * We drop the scheduler lock early (it's a global spinlock),
     * thus we have to lock the previous process from getting
     * rescheduled during switch_to().
     */

/*#endif*/ /* CONFIG_SMP */

    /* kstat.context_swtch++; */
    /*
     * there are 3 processes which are affected by a context switch:
     *
     * prev == .... ==> (last => next)
     *
     * It's the 'much more previous' 'prev' that is on next's stack,
     * but prev is set to (the just run) 'last' process by switch_to().
     * This might sound slightly confusing but makes tons of sense.
     */
```



```
    prepare_to_switch();
    /*
    {
        struct mm_struct *mm = next->mm;
        struct mm_struct *oldmm = prev->active_mm;
        if (!mm) {
            if (next->active_mm) BUG();
            next->active_mm = oldmm;
            atomic_inc(&oldmm->mm_count);
            enter_lazy_tlb(oldmm, next, this_cpu);
        } else {
            if (next->active_mm != mm) BUG();
            switch_mm(oldmm, mm, next, this_cpu);
        }

        if (!prev->mm) {
            prev->active_mm = NULL;
            mmdrop(oldmm);
        }
    }
    */

    /*
     * This just switches the register state and the
     * stack.
     */
    switch_to(prev, next, prev);
    __schedule_tail(prev);

 same_process:

    /* reacquire_kernel_lock(current); */
    if (get_current()->need_resched)
        goto need_resched_back;
    return;
}


/* Called for each timer interrupt on each processor. */
void do_timer()
{
    int cpu=smp_processor_id();
    /* A tick actually shouldent be cpu specific. */
    /* For now we work out the single cpu problem. */
    Task *p = get_current();
    if (p == idle_task(cpu))
    {
        p->need_resched = 1;
    }

    if (--p->counter <= 0) {
        p->counter = 0;
        p->need_resched = 1;
    }
}


/* Return the idle task for the spec. cpu */
```

```
Task* idle_task(int cpu)
{
    if (!idle_tasks[cpu])
        printf("## No idle task for PE:%d\n", cpu);
    return idle_tasks[cpu];
}

/* Return the current task on this cpu */
Task* get_current()
{
    return current[smp_processor_id()];
}


/* Remove a task from the run_queue */
void del_from_runqueue(Task *task)
{
    /* printf("### Delete from rq: %d\n", task->pid); */
    int i = run_queues[task->run_queue].size();
    run_queues[task->run_queue].remove(task);
    nr_running-= i-run_queues[task->run_queue].size();

    /* Balance if nessesary */
    balance();
}

/* Set the mask for the task */
void task_set_cpu(Task *task, int cpu)
{
    task->set_state(TASK_RUNNING);
    current[cpu] = task;
    task->processor = cpu;
    task->cpus_runnable = 1UL << cpu;
}

/* Change to mask to have no cpu */
void task_release_cpu(Task *task)
{
    task->cpus_runnable = ~0UL;
}

/* Examine if the task is running on a cpu */
int task_has_cpu(Task *task)
{
    return (task->cpus_runnable != ~0UL);
}

/* Move a task to the last of the runqueue */
void move_last_runqueue(Task *task)
{
    int i = run_queues[task->run_queue].size();
    run_queues[task->run_queue].remove(task);
    run_queues[task->run_queue].push_back(task);
    nr_running += run_queues[task->run_queue].size()-i;
}

/* Move a task to the front of a runqueue */
void move_first_runqueue(Task *task)
```



```
{
    int i = run_queues[task->run_queue].size();
    run_queues[task->run_queue].remove(task);
    run_queues[task->run_queue].push_front(task);
    nr_running += run_queues[task->run_queue].size()-i;
}

/* Add a task to the runqueue */
void add_to_runqueue(Task *task)
{
    /* Add to the previous runqueue.
       If this is negative, then add to the runqueue
       for this processor */
    if (task->run_queue < 0)
        task->run_queue = smp_processor_id();

    /* printf("### Adding task %d to rq:%d\n", task->pid, task->run_queue);
        */

    run_queues[task->run_queue].push_front(task);
    nr_running++;

    /* Balance if nessesary */
    balance();
}

/* Examine if a task can run on the spec. CPU */
int can_schedule(Task *p, int cpu)
{
    return (p->cpus_runnable & p->cpus_allowed & (1 << cpu));
}

Task* cpu_curr(int cpu)
{
    return current[cpu];
}

/*
 * On x86 all CPUs are mapped 1:1 to the APIC space.
 * This simplifies scheduling and IPI sending and
 * compresses data structures.
 */
int cpu_logical_map(int cpu)
{
    return cpu;
}

int cpu_number_map(int cpu)
{
    return cpu;
}

/* Check if the task is on any runqueue */
int task_on_runqueue(Task *p)
{
    /* Go through the runqueue */
    if (p->run_queue < 0) return 0;
```

```
    /* In linux, the thread would know itself through p->run_list.next */
    return (find(run_queues[p->run_queue].begin(),
                 run_queues[p->run_queue].end(), p) != run_queues[p->
                     run_queue].end());
}

/* Retrive number of ticks since last schedule in this cpu */
/* This is updated for every tick */
cycles_t last_schedule(int cpu)
{
    return cpu_schedule_data[cpu].last_schedule;
}

/* Get System Ticks */
cycles_t get_cycles()
{
    return jiffies;
}

void sched_init(Task *idle_task)
{
    int cpu = smp_processor_id();
    if (!cpu)
    {
        current = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
        idle_tasks = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
        cpu_schedule_data = (struct schedule_data*)
            malloc(sizeof(struct schedule_data)*smp_num_cpus);
        run_queues = new list<Task*>[smp_num_cpus];

    }

    idle_tasks[cpu] = idle_task;
    /* set the idle task as the active task the CPU */
    task_set_cpu(idle_task, cpu);
}
```

sched-multi_queue.cc

```
#include <list>
#include <vector>
#include <math.h>
#include <algorithm>
#include "sched.h"
#include "simulator_interface.h"
#include "sched_interface.h"
#include "globals.h"

using namespace std;

/* A list of running programs on the cpu's */
Task ** current;
Task ** idle_tasks;
/* Per CPU data */
struct schedule_data *cpu_schedule_data;
```



```
#define PROC_CHANGE_PENALTY 15

/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x)   ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x)   ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x)   (x)
#elif HZ < 1600
#define TICK_SCALE(x)   ((x) << 1)
#else
#define TICK_SCALE(x)   ((x) << 2)
#endif

#define NICE_TO_TICKS(nice)     (TICK_SCALE(20-(nice))+1)

struct queue {
    int counter;          /* The total quantum left for the queue */
    int number;
    list<Task*> active;   /* List of active threads in this cycle */
};

/* Number of queues in the system */
#define NR_QUEUES 4

/* Queue 0, CPU-bound
   Queue 3, Interactive.
   Fairness are still in place.
   Interactive tasks are run for shorter periods of time, more
   frequently */

/* Allocate the queues */
static struct queue run_queues[NR_QUEUES];


#define unlikely(x) x
/* Actually this is the original define */
#define prepare_to_switch()   do { } while(0)

/* Assembler macro - This sets up the new task.
 * This is not needed in this simulator */
#define switch_to(prev,next,last) do { } while(0)
void scheduling_functions_begin() {}

void reschedule_idle(Task *p)
{
```

```
        int this_cpu = smp_processor_id();
        Task *tsk, *target_tsk;
        int cpu, best_cpu, i, max_prio;
        cycles_t oldest_idle;

        /*
         * shortcut if the woken up task's last CPU is
         * idle now.
         */
        best_cpu = p->processor;
        if (can_schedule(p, best_cpu)) {
                tsk = idle_task(best_cpu);
                if (cpu_curr(best_cpu) == tsk) {
                        int need_resched;
send_now_idle:
                        /*
                         * If need_resched == -1 then we can skip sending
                         * the IPI altogether, tsk->need_resched is
                         * actively watched by the idle thread.
                         */
                        need_resched = tsk->need_resched;
                        tsk->need_resched = 1;
                        if ((best_cpu != this_cpu) && !need_resched)
                                smp_send_reschedule(best_cpu);
                        return;
                }
        }

        /*
         * We know that the preferred CPU has a cache-affine current
         * process, lets try to find a new idle CPU for the woken-up
         * process. Select the least recently active idle CPU. (that
         * one will have the least active cache context.) Also find
         * the executing process which has the least priority.
         */
        oldest_idle = (cycles_t) -1;
        target_tsk = NULL;
        max_prio = 0;

        for (i = 0; i < smp_num_cpus; i++) {
                cpu = cpu_logical_map(i);
                if (!can_schedule(p, cpu))
                        continue;
                tsk = cpu_curr(cpu);
                /*
                 * We use the first available idle CPU. This creates
                 * a priority list between idle CPUs, but this is not
                 * a problem.
                 */
                if (tsk == idle_task(cpu)) {
/* Added in 2.4.17
#if defined(__i386__) && defined(CONFIG_SMP)
                      *//*
                         * Check if two siblings are idle in the same
                         * physical package. Use them if found.
                         *//*
                         if (smp_num_siblings == 2) {
                                if (cpu_curr(cpu_sibling_map[cpu]) ==
```



```
                                    idle_task(cpu_sibling_map[cpu])) {
                                        oldest_idle = last_schedule(cpu);
                                        target_tsk = tsk;
                                        break;
                                }

                        }
#endif
*/
                        if (last_schedule(cpu) < oldest_idle) {
                                oldest_idle = last_schedule(cpu);
                                target_tsk = tsk;
                        }
                }
        }
        tsk = target_tsk;
        if (tsk) {
                if (oldest_idle != -1ULL) {
                        best_cpu = tsk->processor;
                        goto send_now_idle;
                }
                tsk->need_resched = 1;
                if (tsk->processor != this_cpu)
                        smp_send_reschedule(tsk->processor);
        }
        return;
}

/*
 * Wake up a process. Put it on the run-queue if it's not
 * already there.  The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you're allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
static inline int try_to_wake_up(Task *p, int synchronous)
{
    //unsigned long flags;
        int success = 0;

        /*
         * We want the common case fall through straight, thus the goto.
         */
        /* spin_lock_irqsave(&runqueue_lock, flags); */
        p->state = TASK_RUNNING;
        if (task_on_runqueue(p))
                goto out;
        add_to_runqueue(p);
        if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
                reschedule_idle(p);
        success = 1;
out:
        /* spin_unlock_irqrestore(&runqueue_lock, flags); */
        return success;
}

int wake_up_process(Task * p)
```

```
{
        return try_to_wake_up(p, 0);
}


void __schedule_tail(Task* prev)
{
    task_release_cpu(prev);
}



void schedule_tail(Task* prev)
{
    __schedule_tail(prev);
}

/* We dont know anything about SCHED_RR or SCHED_FIFO.
   We could just implement these in a single queue, that
   must be empty before scheduling SCHED_OTHER */
void schedule()
{
    int this_cpu = smp_processor_id();
    list<Task*>::iterator p;

    struct schedule_data * sched_data;
    Task *prev, *next;
    int c;

    struct queue* active;
    int weight, new_weight;

    prev = get_current();
    next = idle_task(this_cpu);

    printf("###␣Schedule␣on␣PE:␣%d\n", this_cpu);

    /* Write the state of all tasks on one line */
    printf("###␣States:␣");
    for_all_tasks(p) {
        printf("(%d,%d)␣", (*p)->run_queue, (*p)->state);
    }
    printf("\n");

    if (prev != idle_task(this_cpu)) {
        //task_release_cpu(prev);
        if (prev->state != TASK_RUNNING)
            del_from_runqueue(prev);
        else {
            if (prev->counter==0) {
                /* Has the thread used all of its timeslice? */
                if ((!prev->moved) && (prev->run_queue > 0)) {
                    printf("###␣Less␣interactive:␣%d\n", prev->pid);
                    del_from_runqueue(prev);
                    prev->moved++;
                    add_to_runqueue_tail(prev);
                }
            }
```



```
            /*
            if (prev->quantum > 0)
            {
                move_first_runqueue(prev);
            }
            */
        }
    }

need_resched_back:

    weight=-100000;
    active=NULL;
    int t=-100000;
    /* Find the best task to run, if two queues have the same weight,
     * Select the late */
    for (c=0;c<NR_QUEUES;c++)
    {
        printf("###␣Queue␣%d:␣(%d,%d)\n",
               c,run_queues[c].number,run_queues[c].counter);
        if (run_queues[c].number && can_schedule(*run_queues[c].active.begin
             (),this_cpu)) {
            new_weight=run_queues[c].counter/run_queues[c].number;
            if (new_weight >= weight)
            {
                weight = new_weight;
                active = &run_queues[c];
                t=c;
            }
        }
    }

    /* Recalculate counters?
     * Counter are recalculated, if there was a runqueue, but counter <= 0 */
    if (active && active->counter <= 0)
    {

        printf("###␣Recalculating\n");
        for (c=1;c<NR_QUEUES;c++)
            run_queues[c].counter=0;

        for_all_tasks(p) {
            (*p)->counter = ((*p)->counter >> 2) + NICE_TO_TICKS((*p)->nice);
            if ((*p)->state == TASK_RUNNING)
            {
                run_queues[(*p)->run_queue].counter += (*p)->counter;
                (*p)->moved=0;
            }

        }
        goto need_resched_back;
    }


    /* Ok we got the queue, from which we should schedule */
    /* Schedule in a round robin fashion */

    /* A process assigned to another PE cannot be scheduled */
```

```
    if (active) {
        next = *active->active.begin();
        move_last_runqueue(next);
        next->quantum=(NICE_TO_TICKS(next->nice)>>next->run_queue);
    } else
        printf("###␣Choosing␣idle␣process\n");

    /* All process know their quantum.
       A extra quantum is added. A thread is only run for this quantum.
       The temporary quantum depends on the queue the thread is in. */

    /* release_kernel_lock(prev, this_cpu); */

    sched_data = &cpu_schedule_data[this_cpu];

    /* spin_lock_irq(&runqueue_lock); */

    prev->need_resched = 0;

    /*
     * this is the scheduler proper:
     */

    schedule_tail(prev);
    task_set_cpu(next, this_cpu);
    /* spin_unlock_irq(&runqueue_lock); */

    prepare_to_switch();

    /*
     * This just switches the register state and the
     * stack.
     */
    printf("###␣Prev:␣%d,␣Next:␣%d\n", prev->pid, next->pid);
    switch_to(prev, next, prev);

}

/* Called for each timer interrupt on each processor. */
void do_timer()
{
    /* A tick actually shouldent be cpu specific. */
    /* For now we work out the single cpu problem. */
    int cpu = smp_processor_id();
    Task *p = get_current();
    if (p == idle_task(cpu))
    {
        p->need_resched = 1;
    }

    run_queues[p->run_queue].counter--;
    --p->counter;
    --p->quantum;
    if (p->quantum <= 0) {
        p->need_resched = 1;
        p->quantum = 0;
    }
```



```
    if (p->counter <= 0) {
        p->need_resched = 1;
        p->counter = 0;
        p->quantum = 0;
    }
}

void scheduling_functions_end() {}

/* Return the idle task for the spec. cpu */
Task* idle_task(int cpu)
{
    return idle_tasks[cpu];
}

/* Return the current task on this cpu */
Task* get_current()
{
    return current[smp_processor_id()];
}


/* Remove a task from the run_queue */
void del_from_runqueue(Task *p)
{
    if (!task_on_runqueue(p)) return;
    printf("###␣Del␣from␣runqueue:␣%d,%d\n", p->pid,p->run_queue);
    if (!run_queues[p->run_queue].number)
    {
        printf("###␣Bug␣in␣del.␣No␣more␣processes␣on␣queue\n");
        //run_queues[p->run_queue].number++;
    }

    run_queues[p->run_queue].active.remove(p);
    run_queues[p->run_queue].number--;
    run_queues[p->run_queue].counter-=p->counter;

    /* Find out, if the task should be moved to another queue */
    /* If the task has used less than half of the temporary quanta */
    /* Do not move, if it has been moved before within this scheduling cycle
       */
    /* thread suspended. */
    if ((!p->moved) &&
        (p->run_queue < NR_QUEUES-1) &&
        (p->quantum  < (NICE_TO_TICKS(p->nice) >> (p->run_queue+2)))) {
        printf("###␣More␣interactive:␣%d\n", p->pid);
        p->run_queue++;
    }
}

/* Set the mask for the task */
void task_set_cpu(Task *task, int cpu)
{
    task->set_state(TASK_RUNNING);
    current[cpu] = task;
    task->processor = cpu;
    task->cpus_runnable = 1UL << cpu;
}
```

```
/* Change to mask to have no cpu */
void task_release_cpu(Task *task)
{
    task->cpus_runnable = ~0UL;
}

/* Examine if the task is running on a cpu */
int task_has_cpu(Task *task)
{
    return (task->cpus_runnable != ~0UL);
}

/* Move a task to the last of the runqueue */
void move_last_runqueue(Task *p)
{
    run_queues[p->run_queue].active.remove(p);
    run_queues[p->run_queue].active.push_back(p);
}

/* Move a task to the front of a runqueue */
void move_first_runqueue(Task *p)
{
    run_queues[p->run_queue].active.remove(p);
    run_queues[p->run_queue].active.push_front(p);
}

/* Add a task to the runqueue */
void add_to_runqueue(Task *p)
{
    if (task_on_runqueue(p)) return;
    printf("###␣Add␣to␣runqueue:␣%d\n", p->pid);

    run_queues[p->run_queue].active.push_front(p);
    /* Add the rest of the quantum to the run_queue */
    run_queues[p->run_queue].counter += p->counter;
    run_queues[p->run_queue].number++;
}
void add_to_runqueue_tail(Task *p)
{
    if (task_on_runqueue(p)) return;
    printf("###␣Add␣to␣runqueue:␣%d\n", p->pid);

    run_queues[p->run_queue].active.push_back(p);
    /* Add the rest of the quantum to the run_queue */
    run_queues[p->run_queue].counter += p->counter;
    run_queues[p->run_queue].number++;
}

/* Examine if a task can run on the spec. CPU */
int can_schedule(Task *p, int cpu)
{
    return (p->cpus_runnable & p->cpus_allowed & (1 << cpu));
}

Task* cpu_curr(int cpu)
{
    return current[cpu];
```



```
}

/*
 * On x86 all CPUs are mapped 1:1 to the APIC space.
 * This simplifies scheduling and IPI sending and
 * compresses data structures.
 */
int cpu_logical_map(int cpu)
{
    return cpu;
}

int cpu_number_map(int cpu)
{
    return cpu;
}

/* Check if the task is on any runqueue */
int task_on_runqueue(Task *p)
{
    /* Go through the runqueue */
    /* In linux, the thread would know itself through p->run_list.next */
    if (p->run_queue == -1)
        p->run_queue = 1;

    int rq = p->run_queue;
    return ((find(run_queues[rq].active.begin(), run_queues[rq].active.end()
         , p) !=
            run_queues[rq].active.end()) /*||
        (find(run_queues[rq].expired.begin(), run_queues[rq].expired.end(), p
            ) !=
        run_queues[rq].expired.end())*/);
}

/* Retrive number of ticks since last schedule in this cpu */
/* This is updated for every tick */
cycles_t last_schedule(int cpu)
{
    return cpu_schedule_data[cpu].last_schedule;
}

/* Get System Ticks */
cycles_t get_cycles()
{
    return jiffies;
}

void sched_init(Task *idle_task)
{
    int cpu = smp_processor_id();
    if (!cpu)
    {
        current = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
        idle_tasks = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
        cpu_schedule_data = (struct schedule_data*)
            malloc(sizeof(struct schedule_data)*smp_num_cpus);
    }
```

```
    idle_tasks[cpu] = idle_task;
    /* set the idle task as the active task the CPU */
    task_set_cpu(idle_task, cpu);
}
```

sched-round_robin.cc

```
#include <list>
#include <vector>
#include <algorithm>
#include <math.h>
#include "simulator_interface.h"
#include "sched_interface.h"
#include "sched.h"
#include "globals.h"

using namespace std;

/* A list of running programs on the cpu's */
Task ** current;
Task ** idle_tasks;
/* Per CPU data */
struct schedule_data *cpu_schedule_data;

#define unlikely(x) x
/* Actually this is the original define */
#define prepare_to_switch()   do { } while(0)

/* Assembler macro - This sets up the new task.
 * This is not needed in this simulator */
#define switch_to(prev,next,last) do { } while(0)

/* The run queue */
list<Task*> run_queue;

#define HZ 100
#define PROC_CHANGE_PENALTY 15


/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x)   ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x)   ((x) >> 1)
#elif HZ < 800
```



```
#define TICK_SCALE(x)   (x)
#elif HZ < 1600
#define TICK_SCALE(x)   ((x) << 1)
#else
#define TICK_SCALE(x)   ((x) << 2)
#endif

#define NICE_TO_TICKS(nice)     (TICK_SCALE(20-(nice))+1)


int preemption_goodness(Task *prev, Task *p, int cpu)
{
    return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->
         active_mm);
}



void reschedule_idle(Task *p)
{
        int this_cpu = smp_processor_id();
        Task *tsk, *target_tsk;
        int cpu, best_cpu, i, max_prio;
        cycles_t oldest_idle;

        /*
         * shortcut if the woken up task's last CPU is
         * idle now.
         */
        best_cpu = p->processor;
        if (can_schedule(p, best_cpu)) {
                tsk = idle_task(best_cpu);
                if (cpu_curr(best_cpu) == tsk) {
                        int need_resched;
send_now_idle:
                        /*
                         * If need_resched == -1 then we can skip sending
                         * the IPI altogether, tsk->need_resched is
                         * actively watched by the idle thread.
                         */
                        need_resched = tsk->need_resched;
                        tsk->need_resched = 1;
                        if ((best_cpu != this_cpu) && !need_resched)
                                smp_send_reschedule(best_cpu);
                        return;
                }
        }

        /*
         * We know that the preferred CPU has a cache-affine current
         * process, lets try to find a new idle CPU for the woken-up
         * process. Select the least recently active idle CPU. (that
         * one will have the least active cache context.) Also find
         * the executing process which has the least priority.
         */
        oldest_idle = (cycles_t) -1;
        target_tsk = NULL;
        max_prio = 0;
```

```
        for (i = 0; i < smp_num_cpus; i++) {
                cpu = cpu_logical_map(i);
                if (!can_schedule(p, cpu))
                        continue;
                tsk = cpu_curr(cpu);
                /*
                 * We use the first available idle CPU. This creates
                 * a priority list between idle CPUs, but this is not
                 * a problem.
                 */
                if (tsk == idle_task(cpu)) {
/* Added in 2.4.17
#if defined(__i386__) && defined(CONFIG_SMP)
                      *//*
                         * Check if two siblings are idle in the same
                         * physical package. Use them if found.
                         *//*
                         if (smp_num_siblings == 2) {
                                if (cpu_curr(cpu_sibling_map[cpu]) ==
                                    idle_task(cpu_sibling_map[cpu])) {
                                        oldest_idle = last_schedule(cpu);
                                        target_tsk = tsk;
                                        break;
                                }

                        }
#endif
*/
                        if (last_schedule(cpu) < oldest_idle) {
                                oldest_idle = last_schedule(cpu);
                                target_tsk = tsk;
                        }
                } /*else {
                        if (oldest_idle == -1ULL) {
                                int prio = preemption_goodness(tsk, p, cpu);

                                if (prio > max_prio) {
                                        max_prio = prio;
                                        target_tsk = tsk;
                                }
                        }
                        }*/
        }
        tsk = target_tsk;
        if (tsk) {
                if (oldest_idle != -1ULL) {
                        best_cpu = tsk->processor;
                        goto send_now_idle;
                }
                tsk->need_resched = 1;
                if (tsk->processor != this_cpu)
                        smp_send_reschedule(tsk->processor);
        }
        return;
}

/*
```



```
 * Wake up a process. Put it on the run-queue if it's not
 * already there.  The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you're allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
static inline int try_to_wake_up(Task *p, int synchronous)
{
    //unsigned long flags;
        int success = 0;

        /*
         * We want the common case fall through straight, thus the goto.
         */
        /* spin_lock_irqsave(&runqueue_lock, flags); */
        p->state = TASK_RUNNING;
        if (task_on_runqueue(p))
                goto out;
        add_to_runqueue(p);
        if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
                reschedule_idle(p);
        success = 1;
out:
        /* spin_unlock_irqrestore(&runqueue_lock, flags); */
        return success;
}

int wake_up_process(Task * p)
{
        return try_to_wake_up(p, 0);
}


/* Calculate the goodness of the process */
int goodness(Task* p, int this_cpu, int this_mm)
{
    /* This is soly based on quantum and nice-level */
    int weight = -1;
    if (p->policy & SCHED_YIELD)
        return weight;

    if (p->policy == SCHED_OTHER) {
        /*
         * Give the process a first-approximation goodness value
         * according to the number of clock-ticks it has left.
         *
         * Don't do any other calculations if the time slice is
         * over..
         */
        weight = p->counter;
        if (!weight)
            return weight;

        if (p->processor == this_cpu)
            weight += PROC_CHANGE_PENALTY;

        /* .. and a slight advantage to the current MM */
```

```
        if (p->mm == this_mm || !p->mm)
            weight += 1;


        weight += 20 - p->nice;

        return weight;
    }

    /*
     * Realtime process, select the first one on the
     * runqueue (taking priorities within processes
     * into account).
     */
    weight = 1000 + p->rt_priority;
    return weight;
}

void __schedule_tail(Task* prev)
{
    task_release_cpu(prev);
}



void schedule_tail(Task* prev)
{
    __schedule_tail(prev);
}



void schedule()
{
    int this_cpu = smp_processor_id();
    struct schedule_data * sched_data;
    Task *prev, *next;
    int c;

need_resched_back:

    prev = get_current();
    next = idle_task(this_cpu);


    /* This is not modeled in the simulator
    if (unlikely(in_interrupt())) {
        printk("Scheduling in interrupt\n");
        BUG();
    }
    */

    /* release_kernel_lock(prev, this_cpu); */

    /*
     * 'sched_data' is protected by the fact that we can run
```



```
     * only one process per CPU.
     */
    sched_data = &cpu_schedule_data[this_cpu];

    /* spin_lock_irq(&runqueue_lock); */

    /* move an exhausted RR process to be last.. */

    /*
      if (unlikely(prev->policy == SCHED_RR))
      if (!prev->counter) {
      prev->counter = NICE_TO_TICKS(prev->nice);
      move_last_runqueue(prev);
      }*/
    /*
    switch (prev->state) {
        case TASK_INTERRUPTIBLE:
            if (signal_pending(prev)) {
                prev->state = TASK_RUNNING;
                break;
            }
        default:
             del_from_runqueue(prev);
        case TASK_RUNNING:;
    }
    */
    /* Since tasks themself set the state running, no need to check for
     * a pending signal. All in all this boild down to the following: */
    if (prev->state != TASK_RUNNING)
        del_from_runqueue(prev);
    else
        move_last_runqueue(prev);


    prev->need_resched = 0;

    /*
     * this is the scheduler proper:
     */

// repeat_schedule:

    /*
     * Default process to select..
     */
    c = -1000;
    list<Task*>::iterator iter;
    for (iter = run_queue.begin(); iter != run_queue.end(); iter++)
    {
        if (can_schedule((*iter), this_cpu)) {
            next=(*iter);
            if (next->pid > 0)
                break;
        }
    }
    next->counter=NICE_TO_TICKS(next->nice);

    /*
```

```
     * from this point on nothing can prevent us from
     * switching to the next task, save this fact in
     * sched_data.
     */

    /* Add the task to the running list */
    /* Mark which cpu the task is running on */

    /* Sched data is locale for the cpu itself */

    //sched_data->curr = next;
    schedule_tail(prev);
    task_set_cpu(next, this_cpu);
    /* spin_unlock_irq(&runqueue_lock); */

    if (unlikely(prev == next)) {
        /* We won't go through the normal tail, so do this by hand */
        prev->policy &= ~SCHED_YIELD;
        goto same_process;
    }

/*#ifdef CONFIG_SMP*/
    /*
     * maintain the per-process 'last schedule' value.
     * (this has to be recalculated even if we reschedule to
     * the same process) Currently this is only used on SMP,
     * and it's approximate, so we do not have to maintain
     * it while holding the runqueue spinlock.
     */

//    sched_data->last_schedule = get_cycles();

    /*
     * We drop the scheduler lock early (it's a global spinlock),
     * thus we have to lock the previous process from getting
     * rescheduled during switch_to().
     */

/*#endif*/ /* CONFIG_SMP */

    /* kstat.context_swtch++; */
    /*
     * there are 3 processes which are affected by a context switch:
     *
     * prev == .... ==> (last => next)
     *
     * It's the 'much more previous' 'prev' that is on next's stack,
     * but prev is set to (the just run) 'last' process by switch_to().
     * This might sound slightly confusing but makes tons of sense.
     */
    prepare_to_switch();
    /*
    {
        struct mm_struct *mm = next->mm;
        struct mm_struct *oldmm = prev->active_mm;
        if (!mm) {
            if (next->active_mm) BUG();
            next->active_mm = oldmm;
```



```
            atomic_inc(&oldmm->mm_count);
            enter_lazy_tlb(oldmm, next, this_cpu);
        } else {
            if (next->active_mm != mm) BUG();
            switch_mm(oldmm, mm, next, this_cpu);
        }

        if (!prev->mm) {
            prev->active_mm = NULL;
            mmdrop(oldmm);
        }
    }
    */

    /*
     * This just switches the register state and the
     * stack.
     */
    switch_to(prev, next, prev);
    __schedule_tail(prev);

 same_process:

    /* reacquire_kernel_lock(current); */
    if (get_current()->need_resched)
        goto need_resched_back;
    return;
}

void sched_init(Task *idle_task)
{
    int cpu = smp_processor_id();
    if (!cpu)
    {
        current = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
        idle_tasks = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
        cpu_schedule_data = (struct schedule_data*)
            malloc(sizeof(struct schedule_data)*smp_num_cpus);
    }

    idle_tasks[cpu] = idle_task;
    /* set the idle task as the active task the CPU */
    task_set_cpu(idle_task, cpu);
}

/* Called for each timer interrupt on each processor. */
void do_timer()
{
    int cpu = smp_processor_id();
    /* A tick actually shouldent be cpu specific. */
    /* For now we work out the single cpu problem. */
    Task *p = get_current();
    if (p == idle_task(cpu))
    {
        p->need_resched = 1;
    }

    if (--p->counter <= 0) {
```

```
        p->counter = 0;
        p->need_resched = 1;
    }
}

/* Return the idle task for the spec. cpu */
Task* idle_task(int cpu)
{
    return idle_tasks[cpu];
}

/* Return the current task on this cpu */
Task* get_current()
{
    return current[smp_processor_id()];
}


/* Remove a task from the run_queue */
void del_from_runqueue(Task *task)
{
    run_queue.remove(task);
}

/* Set the mask for the task */
void task_set_cpu(Task *task, int cpu)
{
    task->set_state(TASK_RUNNING);
    current[cpu] = task;
    task->processor = cpu;
    task->cpus_runnable = 1UL << cpu;
}

/* Change to mask to have no cpu */
void task_release_cpu(Task *task)
{
    task->cpus_runnable = ~0UL;
}

/* Examine if the task is running on a cpu */
int task_has_cpu(Task *task)
{
    return (task->cpus_runnable != ~0UL);
}

/* Move a task to the last of the runqueue */
void move_last_runqueue(Task *task)
{
    del_from_runqueue(task);
    run_queue.push_back(task);
}

/* Move a task to the front of a runqueue */
void move_first_runqueue(Task *task)
{
    del_from_runqueue(task);
    run_queue.push_front(task);
}
```



```
/* Add a task to the runqueue */
void add_to_runqueue(Task *task)
{
    run_queue.push_front(task);
    /* nr_running++; */
}

/* Examine if a task can run on the spec. CPU */
int can_schedule(Task *p, int cpu)
{
    return (p->cpus_runnable & p->cpus_allowed & (1 << cpu));
}

Task* cpu_curr(int cpu)
{
    return current[cpu];
}

/*
 * On x86 all CPUs are mapped 1:1 to the APIC space.
 * This simplifies scheduling and IPI sending and
 * compresses data structures.
 */
int cpu_logical_map(int cpu)
{
    return cpu;
}

int cpu_number_map(int cpu)
{
    return cpu;
}

/* Check if the task is on any runqueue */
int task_on_runqueue(Task *p)
{
    /* Go through the runqueue */
    /* In linux, the thread would know itself through p->run_list.next */
    return (find(run_queue.begin(), run_queue.end(), p) != run_queue.end());
}

/* Retrive number of ticks since last schedule in this cpu */
/* This is updated for every tick */
cycles_t last_schedule(int cpu)
{
    return cpu_schedule_data[cpu].last_schedule;
}

/* Get System Ticks */
cycles_t get_cycles()
{
    return jiffies;
}
```

sched-second_level.cc

```
#include <vector>
#include <math.h>
#include <algorithm>
#include "simulator_interface.h"
#include "sched_interface.h"
#include "sched.h"
#include "sched-first_level.h"
#include "globals.h"


using namespace std;

/* A list of running programs on the cpu's */
Task ** current;
Task ** idle_tasks;
/* Per CPU data */
struct schedule_data *cpu_schedule_data;

/* The run queue */
list<Task*> run_queue;

#define PROC_CHANGE_PENALTY 15

#define unlikely(x) x
/* Actually this is the original define */

#define prepare_to_switch()   do { } while(0)
/* Assembler macro - This sets up the new task.
 * This is not needed in this simulator */
#define switch_to(prev,next,last) do { } while(0)


/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x)   ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x)   ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x)   (x)
#elif HZ < 1600
#define TICK_SCALE(x)   ((x) << 1)
#else
#define TICK_SCALE(x)   ((x) << 2)
#endif

#define NICE_TO_TICKS(nice)     (TICK_SCALE(20-(nice))+1)
```



```
/* Setup idle tasks.
   These will only be run when there is no items in
   the run_queue */
void init_cpu(int cpu, Task *idle_task)
{
    idle_tasks[cpu] = idle_task;
    /* set the idle task as the active task the CPU */
    task_set_cpu(idle_task, cpu);
}

int preemption_goodness(Task *prev, Task *p, int cpu)
{
    return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->
         active_mm);
}



void reschedule_idle(Task *p)
{
        int this_cpu = smp_processor_id();
        Task *tsk, *target_tsk;
        int cpu, best_cpu, i, max_prio;
        cycles_t oldest_idle;

        /*
         * shortcut if the woken up task's last CPU is
         * idle now.
         */
        best_cpu = p->processor;
        if (can_schedule(p, best_cpu)) {
                tsk = idle_task(best_cpu);
                if (cpu_curr(best_cpu) == tsk) {
                        int need_resched;
send_now_idle:
                        /*
                         * If need_resched == -1 then we can skip sending
                         * the IPI altogether, tsk->need_resched is
                         * actively watched by the idle thread.
                         */
                        need_resched = tsk->need_resched;
                        tsk->need_resched = 1;
                        if ((best_cpu != this_cpu) && !need_resched)
                                smp_send_reschedule(best_cpu);
                        return;
                }
        }

        /*
         * We know that the preferred CPU has a cache-affine current
         * process, lets try to find a new idle CPU for the woken-up
         * process. Select the least recently active idle CPU. (that
         * one will have the least active cache context.) Also find
         * the executing process which has the least priority.
         */
        oldest_idle = (cycles_t) -1;
```

```
        target_tsk = NULL;
        max_prio = 0;

        for (i = 0; i < smp_num_cpus; i++) {
                cpu = cpu_logical_map(i);
                if (!can_schedule(p, cpu))
                        continue;
                tsk = cpu_curr(cpu);
                /*
                 * We use the first available idle CPU. This creates
                 * a priority list between idle CPUs, but this is not
                 * a problem.
                 */
                if (tsk == idle_task(cpu)) {
/* Added in 2.4.17
#if defined(__i386__) && defined(CONFIG_SMP)
                        *//*
                         * Check if two siblings are idle in the same
                         * physical package. Use them if found.
                         *//*
                         if (smp_num_siblings == 2) {
                                if (cpu_curr(cpu_sibling_map[cpu]) ==
                                    idle_task(cpu_sibling_map[cpu])) {
                                        oldest_idle = last_schedule(cpu);
                                        target_tsk = tsk;
                                        break;
                                }

                        }
#endif
*/
                        if (last_schedule(cpu) < oldest_idle) {
                                oldest_idle = last_schedule(cpu);
                                target_tsk = tsk;
                        }
                } else {
                        if (oldest_idle == -1ULL) {
                                int prio = preemption_goodness(tsk, p, cpu);

                                if (prio > max_prio) {
                                        max_prio = prio;
                                        target_tsk = tsk;
                                }
                        }
                }
        }
        tsk = target_tsk;
        if (tsk) {
                if (oldest_idle != -1ULL) {
                        best_cpu = tsk->processor;
                        goto send_now_idle;
                }
                tsk->need_resched = 1;
                if (tsk->processor != this_cpu)
                        smp_send_reschedule(tsk->processor);
        }
        return;
}
```



```
/*
 * Wake up a process. Put it on the run-queue if it's not
 * already there.  The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you're allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
static inline int try_to_wake_up(Task *p, int synchronous)
{
    //unsigned long flags;
        int success = 0;

        /*
         * We want the common case fall through straight, thus the goto.
         */
        /* spin_lock_irqsave(&runqueue_lock, flags); */
        p->state = TASK_RUNNING;
        if (task_on_runqueue(p))
                goto out;
        add_to_runqueue(p);
        if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
                reschedule_idle(p);
        success = 1;
out:
        /* spin_unlock_irqrestore(&runqueue_lock, flags); */
        return success;
}

int wake_up_process(Task * p)
{
        return try_to_wake_up(p, 0);
}


/* Calculate the goodness of the process */
int goodness(Task* p, int this_cpu, int this_mm)
{
    /* This is soly based on quantum and nice-level */
    int weight = -1;
    if (p->policy & SCHED_YIELD)
        return weight;

    if (p->policy == SCHED_OTHER) {
        /*
         * Give the process a first-approximation goodness value
         * according to the number of clock-ticks it has left.
         *
         * Don't do any other calculations if the time slice is
         * over..
         */
        weight = p->counter;
        if (!weight)
            return weight;

        if (p->processor == this_cpu)
            weight += PROC_CHANGE_PENALTY;
```

```
        /* .. and a slight advantage to the current MM */

        if (p->mm == this_mm || !p->mm)
            weight += 1;

        weight += 20 - p->nice;

        return weight;
    }

    /*
     * Realtime process, select the first one on the
     * runqueue (taking priorities within processes
     * into account).
     */
    weight = 1000 + p->rt_priority;
    return weight;
}

void __schedule_tail(Task* prev)
{
    task_release_cpu(prev);
}



void schedule_tail(Task* prev)
{
    __schedule_tail(prev);
}



void schedule()
{
    int this_cpu = smp_processor_id();
    struct schedule_data * sched_data;
    Task *prev, *next;
    int c;

need_resched_back:

    prev = get_current();
    next = idle_task(this_cpu);


    /* This is not modeled in the simulator
    if (unlikely(in_interrupt())) {
        printk("Scheduling in interrupt\n");
        BUG();
    }
    */

    /* release_kernel_lock(prev, this_cpu); */
```



```
    /*
     * 'sched_data' is protected by the fact that we can run
     * only one process per CPU.
     */
    sched_data = &cpu_schedule_data[this_cpu];

    /* spin_lock_irq(&runqueue_lock); */

    /* move an exhausted RR process to be last.. */
    if (unlikely(prev->policy == SCHED_RR))
        if (!prev->counter) {
            prev->counter = NICE_TO_TICKS(prev->nice);
            move_last_runqueue(prev);
        }
    /*
    switch (prev->state) {
        case TASK_INTERRUPTIBLE:
            if (signal_pending(prev)) {
                prev->state = TASK_RUNNING;
                break;
            }
        default:
             del_from_runqueue(prev);
        case TASK_RUNNING:;
    }
    */
    /* Since tasks themself set the state running, no need to check for
     * a pending signal. All in all this boild down to the following: */
    if (prev->state != TASK_RUNNING)
        del_from_runqueue(prev);

    prev->need_resched = 0;

    /*
     * this is the scheduler proper:
     */

repeat_schedule:

    /*
     * Default process to select..
     */
    c = -1000;
    list<Task*>::iterator iter;
    for (iter = run_queue.begin(); iter != run_queue.end(); iter++)
    {
        if (can_schedule((*iter), this_cpu)) {
            int weight = goodness((*iter), this_cpu, prev->active_mm);
            if (weight > c)
                c = weight, next = (*iter);
        }
    }

    /* Do we need to re-calculate counters? */
    if (!c) {
        /* Recalculate all counter on all tasks. */
        list<Task*>::iterator p;
        /* spin_unlock_irq(&runqueue_lock);
```

```
         * read_lock(&tasklist_lock);
         */
        for_all_tasks(p)
            (*p)->counter = ((*p)->counter >> 1) +
            NICE_TO_TICKS((*p)->nice);

        /*
         * read_unlock(&tasklist_lock);
         * spin_lock_irq(&runqueue_lock);
         */
        goto repeat_schedule;
    }

    /*
     * from this point on nothing can prevent us from
     * switching to the next task, save this fact in
     * sched_data.
     */

    /* Add the task to the running list */
    /* Mark which cpu the task is running on */

    /* Sched data is locale for the cpu itself */

    //sched_data->curr = next;
    schedule_tail(prev);
    task_set_cpu(next, this_cpu);
    /* spin_unlock_irq(&runqueue_lock); */

    if (unlikely(prev == next)) {
        /* We won't go through the normal tail, so do this by hand */
        prev->policy &= ~SCHED_YIELD;
        goto same_process;
    }

/*#ifdef CONFIG_SMP*/
    /*
     * maintain the per-process 'last schedule' value.
     * (this has to be recalculated even if we reschedule to
     * the same process) Currently this is only used on SMP,
     * and it's approximate, so we do not have to maintain
     * it while holding the runqueue spinlock.
     */

//    sched_data->last_schedule = get_cycles();

    /*
     * We drop the scheduler lock early (it's a global spinlock),
     * thus we have to lock the previous process from getting
     * rescheduled during switch_to().
     */

/*#endif*/ /* CONFIG_SMP */

    /* kstat.context_swtch++; */
    /*
     * there are 3 processes which are affected by a context switch:
     *
```



```
     * prev == .... ==> (last => next)
     *
     * It's the 'much more previous' 'prev' that is on next's stack,
     * but prev is set to (the just run) 'last' process by switch_to().
     * This might sound slightly confusing but makes tons of sense.
     */
    prepare_to_switch();
    /*
    {
        struct mm_struct *mm = next->mm;
        struct mm_struct *oldmm = prev->active_mm;
        if (!mm) {
            if (next->active_mm) BUG();
            next->active_mm = oldmm;
            atomic_inc(&oldmm->mm_count);
            enter_lazy_tlb(oldmm, next, this_cpu);
        } else {
            if (next->active_mm != mm) BUG();
            switch_mm(oldmm, mm, next, this_cpu);
        }

        if (!prev->mm) {
            prev->active_mm = NULL;
            mmdrop(oldmm);
        }
    }
    */

    /*
     * This just switches the register state and the
     * stack.
     */
    switch_to(prev, next, prev);
    __schedule_tail(prev);

 same_process:

    /* reacquire_kernel_lock(current); */
    if (get_current()->need_resched)
        goto need_resched_back;
    return;
}

/* Called for each timer interrupt on each processor. */
void do_timer()
{
    int cpu = smp_processor_id();
    /* Send a tick to the high level scheduler */
    if (!cpu)
        high_level_tick();

    /* A tick actually shouldent be cpu specific. */
    /* For now we work out the single cpu problem. */
    Task *p = get_current();
    if (p == idle_task(cpu))
    {
        p->need_resched = 1;
    }
```