CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 7

Process Scheduling

February 18, 2025

Huaicheng Li

Acknowledgement: Credits to Dr. Changwoo Min for the original LKP lecture slides.

Agenda

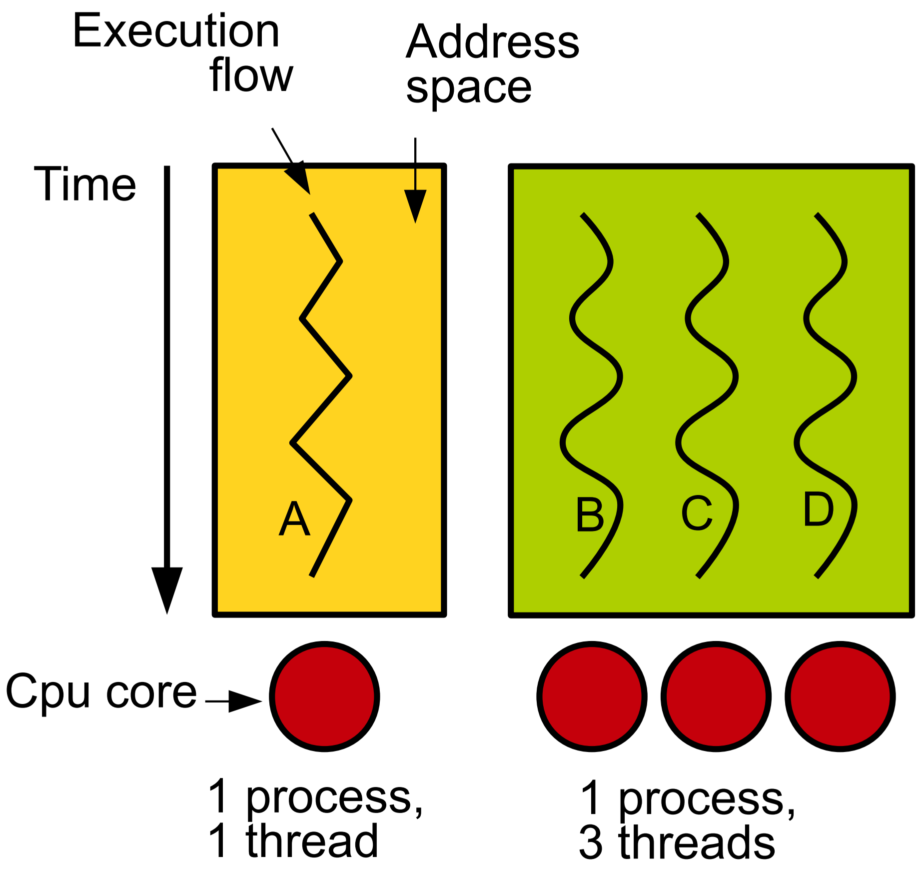
- Process
- Linux PCB: task_struct
- Process creation
- Threads
- Kernel thread API

Forking

- fork() is implemented by the "clone()" system call
- kernel_clone() calls copy_process() and starts the new task
- copy_process()
  - dup_task_struct(), which duplicates kernel stack, task_struct, and thread_info
  - Check that we do not overflow the process number limit
  - Various members of the task_struct are cleared
  - Calls sched_fork() to set the child state set to TASK_NEW
  - Copies parent information such as files, signal handlers, etc.
  - Gets a new PID using alloc(pid)
  - Returns a pointer to the new child task_struct
- Finally, wake_up_new_task()
  - The new child task becomes TASK_RUNNING

Thread

- Threads are concurrent flows of execution belong to the same process sharing the address space



Thread

- There is no concept of a thread in Linux kernel
  - No scheduling for threads
- Linux implements all threads as standard processes
  - A thread is just another process sharing some information with other processes so each thread has its own "task_struct"
  - Create through clone() system call with specific flags indicating sharing
  - clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);

Kernel Thread

- Use to perform background operations in the kernel
- Very similar to sue space threads
  – They are schedulable entities (lie regular processes)
- However, they do not have their own addr space
  – task_struct->mm is NULL
  – why?
- Kernel threads are all forked from the “kthreadd” thread (PID 2)
- Use cases (ps –ppid 2)
  – Work queues (kworker)
  – Load balancing among CPUs (migration)
  – ...

Kernel Thread

- To create a kernel thread, use "kthread_create()"
- When created through kthread_create(), the thread is not in a runnable state
- Need to call wake_up_process() or use kthread_run()
- Other threads can asks a kernel thread to stop using kthread_stop()
  - A kernel thread should check kthread_should _stop() to decide to continue or stop

```c
/**
 * kthread_create - create a kthread on the current node
 * @threadfn: the function to run in the thread
 * @data: data pointer for @threadfn()
 * @namefmt: printf-style format string for the thread name
 * @...: arguments for @namefmt.
 *
 * This macro will create a kthread on the current node, leaving it in
 * the stopped state.
 */
#define kthread_create(threadfn, data, namefmt, arg...) ...

/**
 * wake_up_process - Wake up a specific process
 * @p: The process to be woken up.
 *
 * Attempt to wake up the nominated process and move it to the set of runnable
 * processes.
 *
 * Return: 1 if the process was woken up, 0 if it was already running.
 */
int wake_up_process(struct task_struct *p);
```

```c
/**
 * kthread_run - create and wake a thread.
 * @threadfn: the function to run until signal_pending(current).
 * @data: data ptr for @threadfn.
 * @namefmt: printf-style name for the thread.
 *
 * Description: Convenient wrapper for kthread_create() followed by
 * wake_up_process().  Returns the kthread or ERR_PTR(-ENOMEM).
 */
#define kthread_run(threadfn, data, namefmt, ...) ...

/**
 * kthread_stop - stop a thread created by kthread_create().
 * @k: thread created by kthread_create().
 *
 * Sets kthread_should_stop() for @k to return true, wakes it, and
 * waits for it to exit. If threadfn() may call do_exit() itself,
 * the caller must ensure task_struct can't go away.
 */
int kthread_stop(struct task_struct *k);
```

Kernel Thread Example

- Ext4 file system uses a kernel thread to finish file system initialization in the background

```
/* linux/fs/ext4/super.c */
static int ext4_run_lazyinit_thread(void)
{
    ext4_lazyinit_task = kthread_run(ext4_lazyinit_thread,
                                     ext4_li_info, "ext4lazyinit");
    /* ... */
}

static int ext4_lazyinit_thread(void *arg)
{
    while (true) {
        if (kthread_should_stop()) {
            goto exit_thread;
        }
        /* ... */
```

Example

```
static void ext4_destroy_lazyinit_thread(void)
{
    /* ... */
    kthread_stop(ext4_lazyinit_task);
}

static void __exit ext4_exit_fs(void)
{
    ext4_destroy_lazyinit_thread();
    /* ... */
}

module_exit(ext4_exit_fs)
```

Process Termination

- Termination on invoking the exit() system call
  - Can be implicitly inserted by the compiler on return from main()
  - sys_exit() calls do_exit()
- do_exit() (linux/kernel/exit.c)
  - Cals exit_signals() which set the PF_EXITTInG flag in the task_struct
  - Set the exit code in the exit_code field of the task_struct, which will be retrieved by the parent
  - Calls exit_mm() to release the mm_struct of the task
  - Calls exit_sem(), if the process is queued waiting for a semaphore, dequeue here
  - Calls exit_files() and exit_fs() to decrement the reference counter of file descriptors and filesystem data, respectively. If a refenrece counter becomes zero, that object is no longer in use by any process, and it is destroyed.

- Calls exit_notify()
  - Sends signals to parent
  - Re-parent any of tis children to another thread in the thread group or the init process
  - Set exit_state in task_struct to EXIT_ZOMBIE
- Calls do_task_dead()
  - Set the state to TASK_DEAD
  - Calls schedule() to switch to a new process. Because process is now not schedulable, do_exit() never returns.
- At this point, what is left is task_structu, thread_info, and kernel stack
- This is required to provide information to the parent
  - pid_t wait(int *wstatus)
- After the parent retrieves the information, the remaining memory held by the process is freed
- Cleanup implemented in release_task() called from wait()
  - Remove the task from the task list and release remaining resources

Zombie Process

- What happens if a parent task exits before its child?
- A child must be re-parented
- exit_notify() calls forget_original_parent(), that calls find_new_reaper()
  - Returns the task_struct of another task in the thread group if it exists, other init
  - Then, all the children of the currently dying task are re-parented to the reaper

Further Readings

- Kernel Korner – Sleeping in the Kernel
- Exploiting Stack Overflows in the Linux Kernel

Next Lecture

- Process scheduling!

Processor Scheduler

- Decides which process runs next, when, and for how long
- Responsible for making the best use of processor (CPU)
  - E.g., Do not waste CPU cycles for waiting process
  - E.g., Give higher priority to higher-priority processes
  - E.g., Do not starve low-priority processes

Multitasking

- Simultaneously interleave execution of more than one process
- Single core
  - The processor scheduler gives illusion of multiple processes running concurrently
- Multi-core
  - The processor scheduler enables true parallelism
- Types of multitasking
  - Cooperative multitasking: A process continues running until it yields CPU
  - Preemptive multitasking:
    - » The OS can interrupt the execution of a process (i.e., preemption) after the process exhausts its *timeslice*, which is decided by *process priority*

Process #100

```
long count = 0;
void foo(void) {
 while(1) {
 count++;
 }
}
```

Process #200

```
long val = 2;
void bar(void) {
 while(1) {
 val *= 3;
 }
}
```

Process #300

```
void baz(void) {
 while(1) {
 printf(“hi”);
 }
}
```

Operating system: scheduler

CPU0

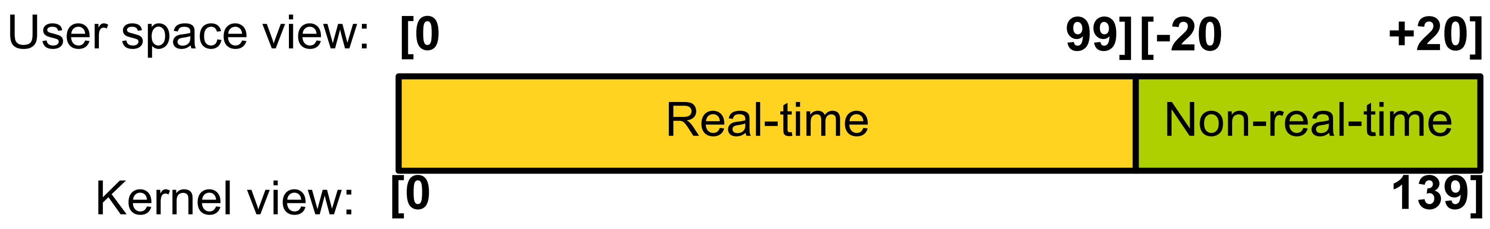
How does the preemptive scheduler take control of the infinite loop?

I/O vs. CPU-bound Tasks

- Scheduling policy: a set of rules determining what runs and when
- I/O-bound processes
  - Spend most of their time waiting for I/O: disk, network, keyboard, mouse, etc.
  - Runs for only short duration
  - Response time is important (i.e., low-latency)
- CPU-bound processes
  - Heavy use of CPU for computations: scientific computations
  - Caches stay hot when they run for a long time

Linux Process Priority

- Priority-based scheduling
  - Rank processes based on their worth and need for processor time
  - Processes with higher priorities run before those with a lower priority
- Priorities in Linux
  - Nice value: [-20, 19], default: 0, high values means lower priority
  - Real-time priority: [0, 99], higher values means higher priority
    - » Real-time processes always executes before standard (nice) processes
  - ps ax –eo pid,ni,rtprio,cmd



Scheduling Policy: timeslice

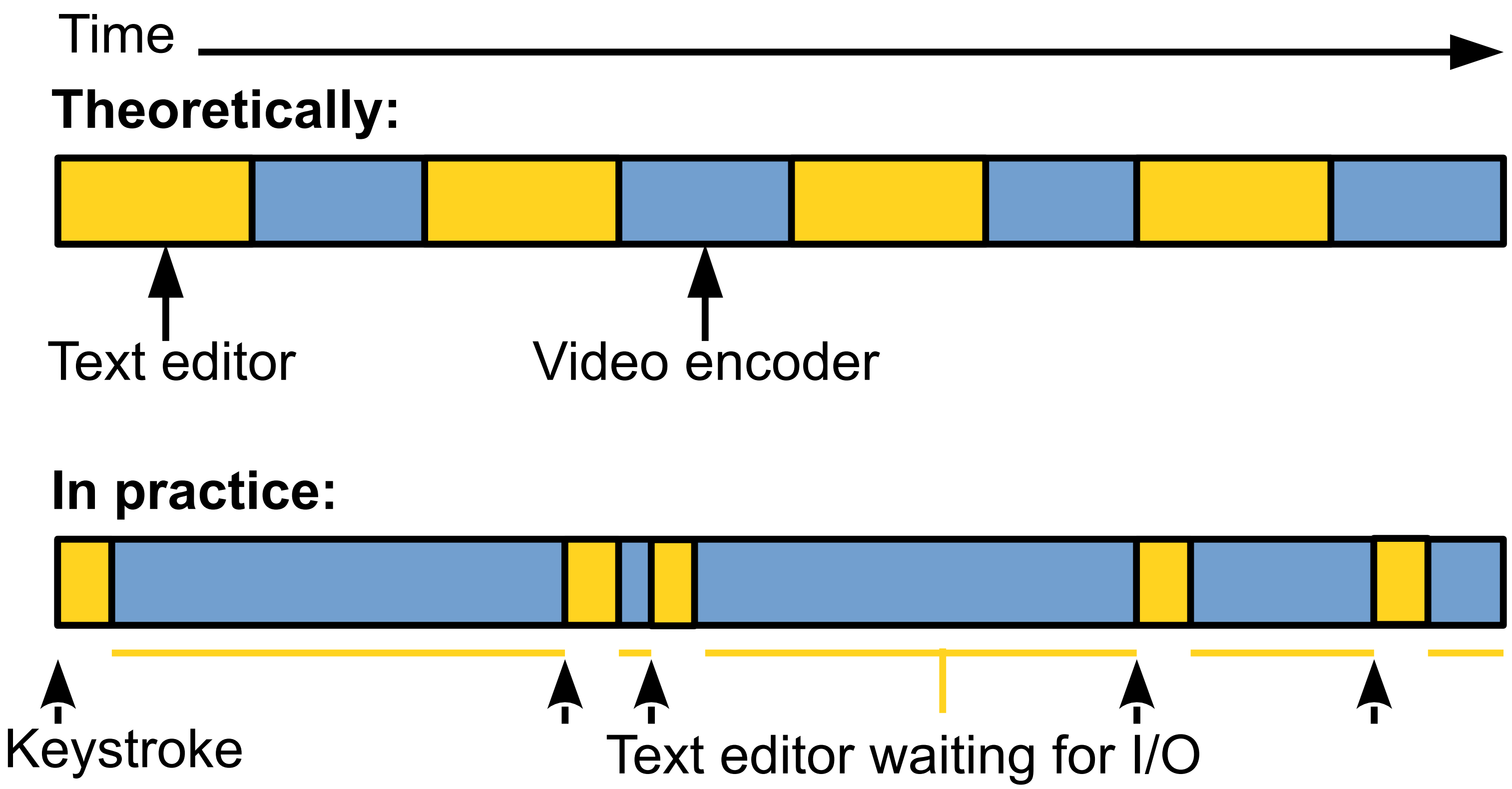
- How much time a process should execute before being preempted
- Trade-offs on setting the right timeslice
  - Too long → poor interactive performance
  - Too short → high context switch overhead

Scheduling Policy: Example

- Two tasks in the system
  - Text editor: I/O-bound, latency sensitive (interactive)
  - Video encoding: CPU-bound, background job
- Scheduling goal
  - Text editor: when ready to run, need to preempt the video encoder
  - Video encoder: run as long as possible for better CPU cache usage
- Example policy
  - Prioritize text editor
  - b/c ...

Linux CFS timeslice

- Linux CFS does not use an absolute timeslice
  - The timeslice a process receives is a function of the load of the system (ie, a proportion of the CPU)
  - In addition, the timeslice is weighted by the process priority
  - When a process P becomes runnable, P will preempt the currently running process C if
    - » P consumes a smaller proportion of the CPU than C
- CFS guarantees the text editor a specific proportion of CPU time
  - CFS keeps track of the actual CPU time used by each program
- e.g., text editor : video encoder = 50% : 50%
  - The text editor mostly sleeps for user inputs and video encoder keeps running until preempted
  - When the text editor wakes up
    - » CFS sees that text editor actually uses less CPU time than the video encoder
    - » Thus, the text editor preempts the video encoder

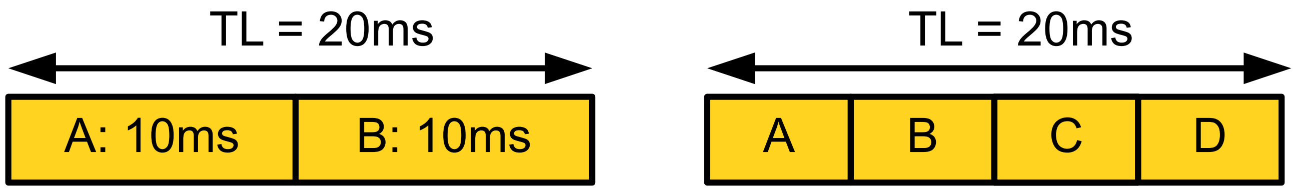


- Good interactive performance
- Good background, CPU-bound performance

Linux CFS Design

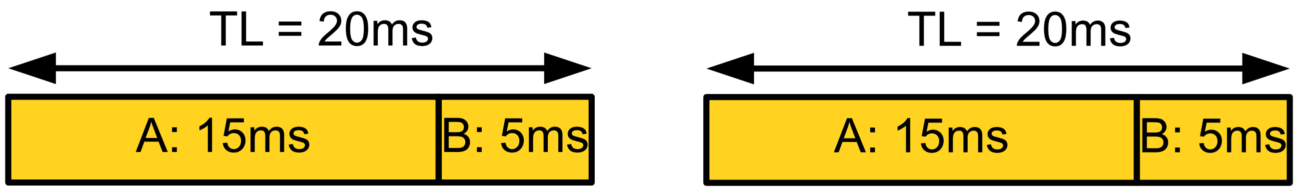
- Completely Fair Scheduler (CFS)
  - More later about EEVDF, successor of CFS
  - An evolution of rotating staircase deadline scheduler (RSDL)
  - Each process of the same priority receives the same amount of CPU time
    - » For n parallel tasks on the CPU, each process should be given 1/n CPU share
  - CFS runs a process for some time, and repeated schedule other tasks
  - No default timeslice, CFS calculates how long a process should run according to the E of runnable processes
    - » The dynamic timeslice is weighted by the process priority (nice)
    - » timeslice = weight of a task / total weight of runnable tasks
  - To calculate the actual timeslice, CFS sets a targets latency
    - » Targeted latency: period during which all runnable processes should be scheduled at least once
    - » Minimum granularity: floor at 1ms (default)

- Example: processes with the same priority



(A_{nice} = B_{nice} = C_{nice} = D_{nice})

- Example: processes with the different priority



(A_{nice} = 0 ; B_{nice} = 5) **(A_{nice} = 10 ; B_{nice} = 15)**

Scheduler Class Design

- The Linux scheduler is modular and provides a pluggable interface for scheduling algorithms
  - Enables different scheduling algorithms to co-exist, scheduling their own types of processes
- Scheduler class is a scheduling algorithm
  - Each scheduler class has a priority
  - e.g., SCHED_FIFO, SCHED_RR, SCHED_BATCH/OTHER, *SCHED_DEADLINE*
- The base scheduler code iterates over each scheduler in priority order
  - linux/kernel/sched/core.c: scheduler_tick(), schedule()
- Time-sharing scheduling: SCHED_BATCH
  - SCHED_NORMAL in kernel code
  - CFS, linux/kernel/sched/fair.c
- Real-time scheduling
  - SCHED_FIFO: first in first out scheduling
  - SCHED_RR: round-robin scheduling
  - SCHED_DEADLINE: sporadic task model deadline scheduling

Scheduler Class Implementation

- sched_class: an abstract class for all scheduler classes

```
/* linux/kernel/sched/sched.h */
struct sched_class {
    /* Called when a task enters a runnable state */
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* Called when a task becomes unrunnable */
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* Yield the processor (dequeue then enqueue back immediately) */
    void (*yield_task) (struct rq *rq);
    /* Preempt the current task with a newly woken task if needed */
    void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);
    /* Choose a next task to run */
    struct task_struct * (*pick_next_task) (struct rq *rq,
                                            struct task_struct *prev,
                                            struct rq_flags *rf);
    /* Called periodically (e.g., 10 msec) by a system timer tick handler */
    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
    /* Update the current task's runtime statistics */
    void (*update_curr) (struct rq *rq);
};
```

- Each scheduler class implements its own functions

```
/* linux/kernel/sched/fair.c */
DEFINE_SCHED_CLASS(fair) = {
    /* const struct sched_class fair_sched_class = { */
    .enqueue_task       = enqueue_task_fair,
    .dequeue_task       = dequeue_task_fair,
    .yield_task         = yield_task_fair,
    .check_preempt_curr = check_preempt_wakeup,
    .pick_next_task     = pick_next_task_fair,
    .task_tick          = task_tick_fair,
    .update_curr        = update_curr_fair, /* ... */
};
/* scheduler tick hitting a task of our scheduling class: */
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &curr->se;
    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);
```

- task_struct

```
/* linux/include/linux/sched.h */
struct task_struct {
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity      se; /* for time-sharing scheduling */
    struct sched_rt_entity   rt; /* for real-time scheduling */
    /* ... */
};
struct sched_entity {
    /* For load-balancing: */
    struct load_weight  load;
    struct rb_node      run_node;
    struct list_head    group_node;
    unsigned int        on_rq;

    u64                 exec_start;
    u64                 sum_exec_runtime;
    u64                 vruntime; /* how much time a process
```

- The base scheduler code triggers scheduling operations in two cases
  - when processing a timer interrupt (schedule_tick())
  - when the kernel calls schedule()

```c
/* linux/kernel/sched/core.c */
/* This function gets called by the timer code, with HZ frequency. */
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;
    struct rq_flags rf;

    /* call task_tick handler for the current process */
    sched_clock_tick();
    rq_lock(rq, &rf);
    update_rq_clock(rq);
    curr->sched_class->task_tick(rq, curr, 0); /* e.g., task_tick_fair in CFS */
    cpu_load_update_active(rq);
    calc_global_load_tick(rq);
    rq_unlock(rq, &rf);

    /* load balancing among CPUs */
    rq->idle_balance = idle_cpu(cpu);
    trigger_load_balance(rq);
    rq_last_tick_reset(rq);
}
```

```c
/* linux/kernel/sched/core.c */
/* __schedule() is the main scheduler function. */
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    /* pick up the highest-prio task */
    next = pick_next_task(rq, prev, &rf);

    if (likely(prev != next)) {
        /* switch to the new MM and the new thread's register state */
        rq->curr = next;
        rq = context_switch(rq, prev, next, &rf);
    }
    /* ... */
}
```

```c
/* linux/kernel/sched/core.c */
/* Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* ... */
again:
    for_each_class(class) {
        /* In CFS, pick_next_task_fair() will be called */
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }

    /* The idle class should always have a runnable task: */
    BUG();
}
```

Time Accounting in CFS

- virtual runtime: how much CPU time a process has used

```
/* linux/include/linux/sched.h */
struct task_struct {
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity      se; /* for time-sharing scheduling */
    struct sched_rt_entity   rt; /* for real-time scheduling */
    /* ... */
};
struct sched_entity {
    /* For load-balancing: */
    struct load_weight  load;
    struct rb_node      run_node;
    struct list_head    group_node;
    unsigned int        on_rq;

    u64                 exec_start;
    u64                 sum_exec_runtime;
    u64                 vruntime; /* how much time a process
```

- Upon every timer interrupt, CFS accounts for the task's execution time

```
/* linux/kernel/sched/fair.c */
/* scheduler_tick() calls task_tick_fair() for CFS.
 * task_tick_fair() calls update_curr() for time accounting. */
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;

    if (unlikely(!curr))
        return;

    delta_exec = now - curr->exec_start; /* Step 1. calc exec duration */
    if (unlikely((s64)delta_exec <- 0))
        return;

    curr->exec_start = now;
    /* continue in a next slide ... */
}
```

```c
static void update_curr(struct cfs_rq *cfs_rq)
{
    /* continue from the previous slide ... */

    schedstat_set(curr->statistics.exec_max,
                  max(delta_exec, curr->statistics.exec_max));

    curr->sum_exec_runtime += delta_exec;
    schedstat_add(cfs_rq->exec_clock, delta_exec);

    /* update vruntime with delta_exec and nice value */
    curr->vruntime += calc_delta_fair(delta_exec, curr); /* CODE */
    update_min_vruntime(cfs_rq);

    if (entity_is_task(curr)) {
        struct task_struct *curtask = task_of(curr);

        trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
        cpuacct_charge(curtask, delta_exec);
        account_group_exec_runtime(curtask, delta_exec);
    }
```

Process Selection in CFS

- CFS maintains a rbtree of tasks indexed by vruntime
- Always pick a task with the smallest vruntime, the left-most node

```
/* linux/kernel/sched/fair.c */
struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq) /* CODE */
{
    struct rb_node *left = cfs_rq->rb_leftmost;

    if (!left)
        return NULL;

    return rb_entry(left, struct sched_entity, run_node);
}
```

Add a Task to Runqueue

- When a task is woken up or migrated, it's added to a runqueue

```
/* linux/kernel/sched/fair.c */
void enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
    bool curr = cfs_rq->curr == se;

    /* Update run-time statistics */
    update_curr(cfs_rq);

    update_load_avg(se, UPDATE_TG);
    enqueue_entity_load_avg(cfs_rq, se);
    update_cfs_shares(se);
    account_entity_enqueue(cfs_rq, se);
    /* ... */

    /* Add this to the rbtree */
    if (!curr)
        __enqueue_entity(cfs_rq, se);
```

```c
/* linux/kernel/sched/fair.c */
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
    struct rb_node *parent = NULL;
    struct sched_entity *entry;
    int leftmost = 1;
    /* Find the right place in the rbtree: */
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        if (entity_before(se, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = 0;
        }
    }
    /* Maintain a cache of leftmost tree entries (it is frequently used): */
    if (leftmost)
        cfs_rq->rb_leftmost = &se->run_node;
    rb_link_node(&se->run_node, parent, link);
```

Remove a Task from Runqueue

- When a task goes to sleep or is migrated, it is removed from a runqueue

```
/* linux/kernel/sched/fair.c */
void dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    /* Update run-time statistics of the 'current'. */
    update_curr(cfs_rq);
    update_load_avg(se, UPDATE_TG);
    dequeue_entity_load_avg(cfs_rq, se);
    update_stats_dequeue(cfs_rq, se, flags);
    clear_buddies(cfs_rq, se);

    /* Remove this to the rbtree */
    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);
    se->on_rq = 0;
    account_entity_dequeue(cfs_rq, se);

static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    if (cfs_rq->rb_leftmost == &se->run_node) {
        struct rb_node *next_node;

        next_node = rb_next(&se->run_node);
        cfs_rq->rb_leftmost = next_node;
    }

    rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Entry Point: schedule()

```
/* linux/kernel/sched/core.c */
/* __schedule() is the main scheduler function. */
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    /* pick up the highest-prio task */
    next = pick_next_task(rq, prev, &rf);

    if (likely(prev != next)) {
        /* switch to the new MM and the new thread's register state */
        rq->curr = next;
        rq = context_switch(rq, prev, next, &rf);
    }
    /* ... */
}
```

```c
/* linux/kernel/sched/core.c */
/* Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* ... */
again:
    for_each_class(class) {
        /* In CFS, pick_next_task_fair() will be called.
         * pick_next_task_fair() eventually calls __pick_first_entity() */
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }
    /* The idle class should always have a runnable task: */
    BUG();
}
```

Sleep and Wake-up

- Reasons for a task to sleep
  - waiting for I/O, blocking on a mutex, etc.
- Steps to sleep
  - Mark a task sleeping
  - Put the task into a waitqueue
  - Dequeue the task from the rbtree
  - The task calls schedule() to select a new process to run
- Waking up a process is the reverse
- Two states associated with sleeping
  - TASK_INTERRUPTIBLE: wake up the sleeping task upon signal
  - TASK_UNINTERRUPTIBLE: defer signal delivery until wake up

Waitqueue: Sleeping

- List of processes waiting for an event to occur (similar to concept of condition variable)

```
/* linux/include/linux/wait.h */
struct wait_queue_entry {
    unsigned int        flags;
    void               *private;
    wait_queue_func_t   func;
    struct list_head    entry;
};
struct wait_queue_head {
    spinlock_t       lock;
    struct list_head    head;
};
typedef struct wait_queue_head wait_queue_head_t;
#define DEFINE_WAIT(name) ...
void add_wait_queue(struct wait_queue_head *wq_head,
                    struct wait_queue_entry *wq_entry);
void prepare_to_wait(struct wait_queue_head *wq_head,
       struct wait_queue_entry *wq_entry, int state);
void finish_wait(struct wait_queue_head *wq_head,
       struct wait_queue_entry *wq_entry);
```

```c
DEFINE_WAIT(wait); /* Initialize a wait queue entry */

/* 'q' is the wait queue that we wish to sleep on */
add_wait_queue(q, &wait); /* Add itself to a wait queue */
while (!condition) { /* event we are waiting for */
    /* Change process status to TASK_INTERRUPTIBLE */
    prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);/* prevent the lost wake-up */
    /* Since the state is TASK_INTERRUPTIBLE, a signal can wake up the task.
     * If there is a pending signal, handle signals */
    if(signal_pending(current)) {
        /* This is a spurious wake up, not caused
         * by the oocurance of the waiting event */
        /* Handle signal */
    }
    /* Go to sleep */
    schedule();
    /* Now, the task is woken up.
     * Check condition if the event occurs */
}

/* Set the process status to TASK_RUNNING
 * and remove itself from the wait queue */
finish_wait(&q, &wait);
```

- Or use one of `wait_event*()` macros

```
/* linux/include/linux/wait.h */

/**
 * wait_event_interruptible - sleep until a condition gets true
 * @wq: the waitqueue to wait on
 * @condition: a C expression for the event to wait for
 *
 * The process is put to sleep (TASK_INTERRUPTIBLE) until the
 * @condition evaluates to true or a signal is received.
 * The @condition is checked each time the waitqueue @wq is woken up.
 */
#define wait_event_interruptible(wq, condition)             \
({                                                          \
    int __ret = 0;                                          \
    might_sleep();                                          \
    if (!(condition))                                       \
        __ret = __wait_event_interruptible(wq, condition);  \
    __ret;                                                  \
})
```

Wake up

- Waking up is taken care of by `wake_up()`
    - By default, wake up *all* the processes on a waitqueue
    - Exclusive tasks are added using prepare_to_wait_exclusive()

```
#define wake_up(x)          __wake_up(x, TASK_NORMAL, 1, NULL)

/* __wake_up() calls __wake_up_common() */
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
            int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;
    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;
        if (curr->func(curr, mode, wake_flags, key) && /* wake-up function */
                (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

- A wait queue entry contains a pointer to a wake-up function

```
/* linux/include/linux/wait.h */

typedef struct __wait_queue wait_queue_t;
typedef int (*wait_queue_func_t)(wait_queue_t *wait, unsigned mode,
                                  int flags, void *key);
int default_wake_function(wait_queue_t *wait, unsigned mode,
                          int flags, void *key);
struct wait_queue_entry {
    unsigned int        flags;
    void               *private;
    wait_queue_func_t   func;
    struct list_head    entry;
};
```