Handling of active – inactive CPUs

```
include/linux/cpu.h:     * cpu_active mask right after SCHED_ACTIVE.  During
include/linux/cpu.h:     * This ordering guarantees consistent cpu_active mask and
include/linux/cpumask.h: *     cpu_active_mask  - has bit 'cpu' set iff cpu available to migration
include/linux/cpumask.h:extern const struct cpumask *const cpu_active_mask;
include/linux/cpumask.h:#define num_active_cpus()    cpumask_weight(cpu_active_mask)
include/linux/cpumask.h:#define cpu_active(cpu)      cpumask_test_cpu((cpu), cpu_active_mask)
include/linux/cpumask.h:#define cpu_active(cpu)      ((cpu) == 0)
include/linux/cpumask.h:void set_cpu_active(unsigned int cpu, bool active);
```

```
init/main.c:      set_cpu_active(cpu, true);
```

```
kernel/cpu.c:static DECLARE_BITMAP(cpu_active_bits, CONFIG_NR_CPUS) __read_mostly;
kernel/cpu.c:const struct cpumask *const cpu_active_mask = to_cpumask(cpu_active_bits);
kernel/cpu.c:EXPORT_SYMBOL(cpu_active_mask);


kernel/cpu.c:           cpumask_set_cpu(cpu, to_cpumask(cpu_active_bits));

kernel/cpu.c:void set_cpu_active(unsigned int cpu, bool active)
kernel/cpu.c:           cpumask_set_cpu(cpu, to_cpumask(cpu_active_bits));
kernel/cpu.c:           cpumask_clear_cpu(cpu, to_cpumask(cpu_active_bits));

kernel/cpuset.c:  if (!cpumask_equal(top_cpuset.cpus_allowed, cpu_active_mask))
kernel/cpuset.c:        if (!cpumask_subset(trialcs->cpus_allowed, cpu_active_mask))
kernel/cpuset.c: * synchronized to cpu_active_mask and N_MEMORY, which is necessary in
kernel/cpuset.c:  cpumask_copy(&new_cpus, cpu_active_mask);
kernel/cpuset.c:  /* synchronize cpus_allowed to cpu_active_mask */
kernel/cpuset.c:  cpumask_copy(top_cpuset.cpus_allowed, cpu_active_mask);

kernel/sched/core.c:                if (!cpu_active(dest_cpu))
kernel/sched/core.c:                if (!cpu_active(dest_cpu))

kernel/sched/core.c:    if (likely(cpu_active(dest_cpu))) {

kernel/sched/core.c:    dest_cpu = cpumask_any_and(cpu_active_mask, cpumask);
kernel/sched/core.c:    if (!cpumask_intersects(new_mask, cpu_active_mask)) {
kernel/sched/core.c:    dest_cpu = cpumask_any_and(cpu_active_mask, new_mask);

kernel/sched/core.c:    if (unlikely(!cpu_active(dest_cpu)))
kernel/sched/core.c:static int __cpuinit sched_cpu_active(struct notifier_block *nfb,
kernel/sched/core.c:          set_cpu_active((long)hcpu, true);
kernel/sched/core.c:          set_cpu_activecpu_active((long)hcpu, false);
kernel/sched/core.c:    cpu_notifier(sched_cpu_active, CPU_PRI_SCHED_ACTIVE);
kernel/sched/core.c:    if (cpumask_test_cpu(rq->cpu, cpu_active_mask))
kernel/sched/core.c:          cpumask_andnot(doms_new[0], cpu_active_mask, cpu_isolated_map);
kernel/sched/core.c: * Update cpusets according to cpu_active mask.  If cpusets are
kernel/sched/core.c:static int cpuset_cpu_active(struct notifier_block *nfb, unsigned long action,
kernel/sched/core.c:    init_sched_domains(cpu_active_mask);
kernel/sched/core.c:    hotcpu_notifier(cpuset_cpu_active, CPU_PRI_CPUSET_ACTIVE);
```

```
kernel/sched/fair.c:    cpumask_copy(cpus, cpu_active_mask);
kernel/sched/fair.c:    if (!cpu_active(cpu))

kernel/sched/sched.h:   if (!cpu_active(cpu_of(rq)))

kernel/stop_machine.c:  BUG_ON(cpu_active(raw_smp_processor_id()));
kernel/stop_machine.c:  queue_stop_cpus_work(cpu_active_mask, stop_machine_cpu_stop, &smdata,
```

init/main.c:

```
/*
 *    Activate the first processor.
 */

static void __init boot_cpu_init(void)
{
      int cpu = smp_processor_id();
      /* Mark the boot cpu "present", "online" etc for SMP and UP case */
      set_cpu_online(cpu, true);
      set_cpu_active(cpu, true);
      set_cpu_present(cpu, true);
      set_cpu_possible(cpu, true);
}
```

kernel/cpu.c:

```
void set_cpu_active(unsigned int cpu, bool active)
{
      if (active)
            cpumask_set_cpu(cpu, to_cpumask(cpu_active_bits));
      else
            cpumask_clear_cpu(cpu, to_cpumask(cpu_active_bits));
}
```

kernel/cpuset.c:

```
/*
 * Rebuild scheduler domains.
 *
 * If the flag 'sched_load_balance' of any cpuset with non-empty
 * 'cpus' changes, or if the 'cpus' allowed changes in any cpuset
 * which has that flag enabled, or if any cpuset with a non-empty
 * 'cpus' is removed, then call this routine to rebuild the
 * scheduler's dynamic sched domains.
 *
 * Call with cpuset_mutex held.  Takes get_online_cpus().
 */
static void rebuild_sched_domains_locked(void)
{
      struct sched_domain_attr *attr;
      cpumask_var_t *doms;
      int ndoms;

      lockdep_assert_held(&cpuset_mutex);
      get_online_cpus();

      /*
       * We have raced with CPU hotplug. Don't do anything to avoid
       * passing doms with offlined cpu to partition_sched_domains().
       * Anyways, hotplug work item will rebuild sched domains.
       */
      if (!cpumask_equal(top_cpuset.cpus_allowed, cpu_active_mask))
            goto out;kernel/cpuset.c:

      /* Generate domain masks and attrs */
      ndoms = generate_sched_domains(&doms, &attr);

      /* Have scheduler rebuild the domains */
      partition_sched_domains(ndoms, doms, attr);
out:
      put_online_cpus();
}
```

```c
/**
 * update_cpumask - update the cpus_allowed mask of a cpuset and all tasks in it
 * @cs: the cpuset to consider
 * @buf: buffer of cpu numbers written to this cpuset
 */
static int update_cpumask(struct cpuset *cs, struct cpuset *trialcs,
                    const char *buf)
{
	struct ptr_heap heap;
	int retval;
	int is_load_balanced;

	/* top_cpuset.cpus_allowed tracks cpu_online_mask; it's read-only */
	if (cs == &top_cpuset)
		return -EACCES;

	/*
	 * An empty cpus_allowed is ok only if the cpuset has no tasks.
	 * Since cpulist_parse() fails on an empty mask, we special case
	 * that parsing.  The validate_change() call ensures that cpusets
	 * with tasks have cpus.
	 */
	if (!*buf) {
		cpumask_clear(trialcs->cpus_allowed);
	} else {
		retval = cpulist_parse(buf, trialcs->cpus_allowed);
		if (retval < 0)
			return retval;

		if (!cpumask_subset(trialcs->cpus_allowed, cpu_active_mask))
			return -EINVAL;
	}
	retval = validate_change(cs, trialcs);
	if (retval < 0)
		return retval;

	/* Nothing to do if the cpus didn't change */
	if (cpumask_equal(cs->cpus_allowed, trialcs->cpus_allowed))
		return 0;

	retval = heap_init(&heap, PAGE_SIZE, GFP_KERNEL, NULL);
	if (retval)
```

```
		return retval;

	is_load_balanced = is_sched_load_balance(trialcs);

	mutex_lock(&callback_mutex);
	cpumask_copy(cs->cpus_allowed, trialcs->cpus_allowed);
	mutex_unlock(&callback_mutex);

	/*
	 * Scan tasks in the cpuset, and update the cpumasks of any
	 * that need an update.
	 */
	update_tasks_cpumask(cs, &heap);

	heap_free(&heap);

	if (is_load_balanced)
		rebuild_sched_domains_locked();
	return 0;
}
```

kernel/sched/core.c:

```
static int select_fallback_rq(int cpu, struct task_struct *p)
{
      int nid = cpu_to_node(cpu);
      const struct cpumask *nodemask = NULL;
      enum { cpuset, possible, fail } state = cpuset;
      int dest_cpu;

      /*
       * If the node that the cpu is on has been offlined, cpu_to_node()
       * will return -1. There is no cpu on the node, and we should
       * select the cpu on the other node.
       */
      if (nid != -1) {
            nodemask = cpumask_of_node(nid);

            /* Look for allowed, online CPU in same node. */
            for_each_cpu(dest_cpu, nodemask) {
                  if (!cpu_online(dest_cpu))
                        continue;
                  if (!cpu_active(dest_cpu))
                        continue;
                  if (cpumask_test_cpu(dest_cpu, tsk_cpus_allowed(p)))
                        return dest_cpu;
            }
      }

      for (;;) {
            /* Any allowed, online CPU? */
            for_each_cpu(dest_cpu, tsk_cpus_allowed(p)) {
                  if (!cpu_online(dest_cpu))
                        continue;
                  if (!cpu_active(dest_cpu))
                        continue;
                  goto out;
            }

            switch (state) {
            case cpuset:
                  /* No more Mr. Nice Guy. */
                  cpuset_cpus_allowed_fallback(p);
```

```
                        state = possible;
                        break;

                case possible:
                        do_set_cpus_allowed(p, cpu_possible_mask);
                        state = fail;
                        break;

                case fail:
                        printk(KERN_ERR "select_fallback_rq failed\n");
                        printk(KERN_ERR " system_state = %d\n", system_state);
                        printk(KERN_ERR " cpu = %d\n", cpu);
                        printk(KERN_ERR " task = %s\n", p->comm);
                        printk(KERN_ERR " allowed_cpus  = %#010lx\n",   p->cpus_allowed.bits[0]);
                        printk(KERN_ERR " nr_cpus_allowed = %d\n", p->nr_cpus_allowed);
                        printk(KERN_ERR " tsk_cpus_allowed = %#010lx\n", tsk_cpus_allowed(p)->bits[0]);
#ifdef CONFIG_PREEMPT_RT_FULL
                        printk(KERN_ERR " migrate_disable = %x\n" , p->migrate_disable);
#endif
#ifdef CONFIG_SCHED_DEBUG
                        printk(KERN_ERR " migrate_disable_atomic = %x\n" , p->migrate_disable_atomic);
#endif
                        BUG();
                        break;
                }
        }

out:
        if (state != cpuset) {
                /*
                 * Don't tell them about moving exiting tasks or
                 * kernel threads (both mm NULL), since they never
                 * leave kernel.
                 */
                if (p->mm && printk_ratelimit()) {
                        printk_deferred("process %d (%s) no longer affine to cpu%d\n",
                                        task_pid_nr(p), p->comm, cpu);
                }
        }

        return dest_cpu;
}
```

```
/*
 * sched_exec - execve() is a valuable balancing opportunity, because at
 * this point the task has the smallest effective memory and cache footprint.
 */
void sched_exec(void)
{
	struct task_struct *p = current;
	unsigned long flags;
	int dest_cpu;

	raw_spin_lock_irqsave(&p->pi_lock, flags);
	dest_cpu = p->sched_class->select_task_rq(p, SD_BALANCE_EXEC, 0);
	if (dest_cpu == smp_processor_id())
		goto unlock;

	if (likely(cpu_active(dest_cpu))) {
		struct migration_arg arg = { p, dest_cpu };

		raw_spin_unlock_irqrestore(&p->pi_lock, flags);
		stop_one_cpu(task_cpu(p), migration_cpu_stop, &arg);
		return;
	}
unlock:
	raw_spin_unlock_irqrestore(&p->pi_lock, flags);
}
```

```
/**
 * migrate_me - try to move the current task off this cpu
 *
 * Used by the pin_current_cpu() code to try to get tasks
 * to move off the current CPU as it is going down.
 * It will only move the task if the task isn't pinned to
 * the CPU (with migrate_disable, affinity or NO_SETAFFINITY)
 * and the task has to be in a RUNNING state. Otherwise the
 * movement of the task will wake it up (change its state
 * to running) when the task did not expect it.
 *
 * Returns 1 if it succeeded in moving the current task
 *         0 otherwise.
 */
int migrate_me(void)
{
      struct task_struct *p = current;
      struct migration_arg arg;
      struct cpumask *cpumask;
      struct cpumask *mask;
      unsigned long flags;
      unsigned int dest_cpu;
      struct rq *rq;

      /*
       * We can not migrate tasks bounded to a CPU or tasks not
       * running. The movement of the task will wake it up.
       */
      if (p->flags & PF_NO_SETAFFINITY || p->state)
            return 0;

      mutex_lock(&sched_down_mutex);
      rq = task_rq_lock(p, &flags);

      cpumask = &__get_cpu_var(sched_cpumasks);
      mask = &p->cpus_allowed;

      cpumask_andnot(cpumask, mask, &sched_down_cpumask);

      if (!cpumask_weight(cpumask)) {
            /* It's only on this CPU? */
            task_rq_unlock(rq, p, &flags);
```

```
		mutex_unlock(&sched_down_mutex);
		return 0;
	}

	dest_cpu = cpumask_any_and(cpu_active_mask, cpumask);

	arg.task = p;
	arg.dest_cpu = dest_cpu;

	task_rq_unlock(rq, p, &flags);

	stop_one_cpu(cpu_of(rq), migration_cpu_stop, &arg);
	tlb_migrate_finish(p->mm);
	mutex_unlock(&sched_down_mutex);

	return 1;
}
```

```c
/*
 * This is how migration works:
 *
 * 1) we invoke migration_cpu_stop() on the target CPU using
 *    stop_one_cpu().
 * 2) stopper starts to run (implicitly forcing the migrated thread
 *    off the CPU)
 * 3) it checks whether the migrated task is still in the wrong runqueue.
 * 4) if it's in the wrong runqueue then the migration thread removes
 *    it and puts it into the right queue.
 * 5) stopper completes and stop_one_cpu() returns and the migration
 *    is done.
 */

/*
 * Change a given task's CPU affinity. Migrate the thread to a
 * proper CPU and schedule it away if the CPU it's executing on
 * is removed from the allowed bitmask.
 *
 * NOTE: the caller must have a valid reference to the task, the
 * task must not exit() & deallocate itself prematurely. The
 * call is not atomic; no spinlocks may be held.
 */
int set_cpus_allowed_ptr(struct task_struct *p, const struct cpumask *new_mask)
{
	unsigned long flags;
	struct rq *rq;
	unsigned int dest_cpu;
	int ret = 0;

	rq = task_rq_lock(p, &flags);

	if (cpumask_equal(&p->cpus_allowed, new_mask))
		goto out;

	if (!cpumask_intersects(new_mask, cpu_active_mask)) {
		ret = -EINVAL;
		goto out;
	}

	do_set_cpus_allowed(p, new_mask);
```

```
	/* Can the task run on the task's current CPU? If so, we're done */
	if (cpumask_test_cpu(task_cpu(p), new_mask) || __migrate_disabled(p))
		goto out;

	dest_cpu = cpumask_any_and(cpu_active_mask, new_mask);
	if (p->on_rq) {
		struct migration_arg arg = { p, dest_cpu };
		/* Need help from migration thread: drop lock and wait. */
		task_rq_unlock(rq, p, &flags);
		stop_one_cpu(cpu_of(rq), migration_cpu_stop, &arg);
		tlb_migrate_finish(p->mm);
		return 0;
	}
out:
	task_rq_unlock(rq, p, &flags);

	return ret;
}
EXPORT_SYMBOL_GPL(set_cpus_allowed_ptr);
```

```
/*
 * Move (not current) task off this cpu, onto dest cpu. We're doing
 * this because either it can't run here any more (set_cpus_allowed()
 * away from this CPU, or CPU going down), or because we're
 * attempting to rebalance this task on exec (sched_exec).
 *
 * So we race with normal scheduler movements, but that's OK, as long
 * as the task is no longer on this CPU.
 *
 * Returns non-zero if task was successfully migrated.
 */
static int __migrate_task(struct task_struct *p, int src_cpu, int dest_cpu)
{
	struct rq *rq_dest, *rq_src;
	int ret = 0;

	if (unlikely(!cpu_active(dest_cpu)))
		return ret;

	rq_src = cpu_rq(src_cpu);
	rq_dest = cpu_rq(dest_cpu);

	raw_spin_lock(&p->pi_lock);
	double_rq_lock(rq_src, rq_dest);
	/* Already moved. */
	if (task_cpu(p) != src_cpu)
		goto done;
	/* Affinity changed (again). */
	if (!cpumask_test_cpu(dest_cpu, tsk_cpus_allowed(p)))
		goto fail;

	/*
	 * If we're not on a rq, the next wake-up will ensure we're
	 * placed properly.
	 */
	if (p->on_rq) {
		dequeue_task(rq_src, p, 0);
		set_task_cpu(p, dest_cpu);
		enqueue_task(rq_dest, p, 0);
		check_preempt_curr(rq_dest, p, 0);
	}
done:
```

```
        ret = 1;
fail:
        double_rq_unlock(rq_src, rq_dest);
        raw_spin_unlock(&p->pi_lock);
        return ret;
}
```

```
static int __cpuinit sched_cpu_active(struct notifier_block *nfb,
                              unsigned long action, void *hcpu)
{
      switch (action & ~CPU_TASKS_FROZEN) {
      case CPU_DOWN_FAILED:
            set_cpu_active((long)hcpu, true);
            return NOTIFY_OK;
      default:
            return NOTIFY_DONE;
      }
}

static int __cpuinit sched_cpu_inactive(struct notifier_block *nfb,
                              unsigned long action, void *hcpu)
{
      switch (action & ~CPU_TASKS_FROZEN) {
      case CPU_DOWN_PREPARE:
            set_cpu_active((long)hcpu, false);
            return NOTIFY_OK;
      default:
            return NOTIFY_DONE;
      }
}

static int __init migration_init(void)
{
      void *cpu = (void *)(long)smp_processor_id();
      int err;

      /* Initialize migration for the boot CPU */
      err = migration_call(&migration_notifier, CPU_UP_PREPARE, cpu);
      BUG_ON(err == NOTIFY_BAD);
      migration_call(&migration_notifier, CPU_ONLINE, cpu);
      register_cpu_notifier(&migration_notifier);

      /* Register cpu active notifiers */
      cpu_notifier(sched_cpu_active, CPU_PRI_SCHED_ACTIVE);
      cpu_notifier(sched_cpu_inactive, CPU_PRI_SCHED_INACTIVE);

      return 0;
}
early_initcall(migration_init);
```

```
static void rq_attach_root(struct rq *rq, struct root_domain *rd)
{
	struct root_domain *old_rd = NULL;
	unsigned long flags;

	raw_spin_lock_irqsave(&rq->lock, flags);

	if (rq->rd) {
		old_rd = rq->rd;

		if (cpumask_test_cpu(rq->cpu, old_rd->online))
			set_rq_offline(rq);

		cpumask_clear_cpu(rq->cpu, old_rd->span);

		/*
		 * If we dont want to free the old_rt yet then
		 * set old_rd to NULL to skip the freeing later
		 * in this function:
		 */
		if (!atomic_dec_and_test(&old_rd->refcount))
			old_rd = NULL;
	}

	atomic_inc(&rd->refcount);
	rq->rd = rd;

	cpumask_set_cpu(rq->cpu, rd->span);
	if (cpumask_test_cpu(rq->cpu, cpu_active_mask))
		set_rq_online(rq);

	raw_spin_unlock_irqrestore(&rq->lock, flags);

	if (old_rd)
		call_rcu_sched(&old_rd->rcu, free_rootdomain);
}
```

```c
/*
 * Partition sched domains as specified by the 'ndoms_new'
 * cpumasks in the array doms_new[] of cpumasks. This compares
 * doms_new[] to the current sched domain partitioning, doms_cur[].
 * It destroys each deleted domain and builds each new domain.
 *
 * 'doms_new' is an array of cpumask_var_t's of length 'ndoms_new'.
 * The masks don't intersect (don't overlap.) We should setup one
 * sched domain for each mask. CPUs not in any of the cpumasks will
 * not be load balanced. If the same cpumask appears both in the
 * current 'doms_cur' domains and in the new 'doms_new', we can leave
 * it as it is.
 *
 * The passed in 'doms_new' should be allocated using
 * alloc_sched_domains.  This routine takes ownership of it and will
 * free_sched_domains it when done with it. If the caller failed the
 * alloc call, then it can pass in doms_new == NULL && ndoms_new == 1,
 * and partition_sched_domains() will fallback to the single partition
 * 'fallback_doms', it also forces the domains to be rebuilt.
 *
 * If doms_new == NULL it will be replaced with cpu_online_mask.
 * ndoms_new == 0 is a special case for destroying existing domains,
 * and it will not create the default domain.
 *
 * Call with hotplug lock held
 */
void partition_sched_domains(int ndoms_new, cpumask_var_t doms_new[],
                             struct sched_domain_attr *dattr_new)
{
      int i, j, n;
      int new_topology;

      mutex_lock(&sched_domains_mutex);

      /* always unregister in case we don't destroy any domains */
      unregister_sched_domain_sysctl();

      /* Let architecture update cpu core mappings. */
      new_topology = arch_update_cpu_topology();

      n = doms_new ? ndoms_new : 0;
```

```
        /* Destroy deleted domains */
        for (i = 0; i < ndoms_cur; i++) {
                for (j = 0; j < n && !new_topology; j++) {
                        if (cpumask_equal(doms_cur[i], doms_new[j])
                            && dattrs_equal(dattr_cur, i, dattr_new, j))
                              goto match1;
                }
                /* no match - a current sched domain not in new doms_new[] */
                detach_destroy_domains(doms_cur[i]);
match1:
                ;
        }

        if (doms_new == NULL) {
                ndoms_cur = 0;
                doms_new = &fallback_doms;
                cpumask_andnot(doms_new[0], cpu_active_mask, cpu_isolated_map);
                WARN_ON_ONCE(dattr_new);
        }

        /* Build new domains */
        for (i = 0; i < ndoms_new; i++) {
                for (j = 0; j < ndoms_cur && !new_topology; j++) {
                        if (cpumask_equal(doms_new[i], doms_cur[j])
                            && dattrs_equal(dattr_new, i, dattr_cur, j))
                              goto match2;
                }
                /* no match - add a new doms_new */
                build_sched_domains(doms_new[i], dattr_new ? dattr_new + i : NULL);
match2:
                ;
        }

        /* Remember the new sched domains */
        if (doms_cur != &fallback_doms)
                free_sched_domains(doms_cur, ndoms_cur);
        kfree(dattr_cur); /* kfree(NULL) is safe */
        doms_cur = doms_new;
        dattr_cur = dattr_new;
        ndoms_cur = ndoms_new;

        register_sched_domain_sysctl();
```

```
        mutex_unlock(&sched_domains_mutex);
}
```

```c
/*
 * Update cpusets according to cpu_active mask.  If cpusets are
 * disabled, cpuset_update_active_cpus() becomes a simple wrapper
 * around partition_sched_domains().
 *
 * If we come here as part of a suspend/resume, don't touch cpusets because we
 * want to restore it back to its original state upon resume anyway.
 */
static int cpuset_cpu_active(struct notifier_block *nfb, unsigned long action,
                       void *hcpu)
{
      switch (action) {
      case CPU_ONLINE_FROZEN:
      case CPU_DOWN_FAILED_FROZEN:

            /*
             * num_cpus_frozen tracks how many CPUs are involved in suspend
             * resume sequence. As long as this is not the last online
             * operation in the resume sequence, just build a single sched
             * domain, ignoring cpusets.
             */
            num_cpus_frozen--;
            if (likely(num_cpus_frozen)) {
                  partition_sched_domains(1, NULL, NULL);
                  break;
            }

            /*
             * This is the last CPU online operation. So fall through and
             * restore the original sched domains by considering the
             * cpuset configurations.
             */

      case CPU_ONLINE:
      case CPU_DOWN_FAILED:
            cpuset_update_active_cpus(true);
            break;
      default:
            return NOTIFY_DONE;
      }
      return NOTIFY_OK;
}
```

```
void __init sched_init_smp(void)
{
	cpumask_var_t non_isolated_cpus;

	alloc_cpumask_var(&non_isolated_cpus, GFP_KERNEL);
	alloc_cpumask_var(&fallback_doms, GFP_KERNEL);

	sched_init_numa();

	get_online_cpus();
	mutex_lock(&sched_domains_mutex);
	init_sched_domains(cpu_active_mask);
	cpumask_andnot(non_isolated_cpus, cpu_possible_mask, cpu_isolated_map);
	if (cpumask_empty(non_isolated_cpus))
		cpumask_set_cpu(smp_processor_id(), non_isolated_cpus);
	mutex_unlock(&sched_domains_mutex);
	put_online_cpus();

	hotcpu_notifier(sched_domains_numa_masks_update, CPU_PRI_SCHED_ACTIVE);
	hotcpu_notifier(cpuset_cpu_active, CPU_PRI_CPUSET_ACTIVE);
	hotcpu_notifier(cpuset_cpu_inactive, CPU_PRI_CPUSET_INACTIVE);

	/* RT runtime code needs to handle some hotplug events */
	hotcpu_notifier(update_runtime, 0);

	init_hrtick();

	/* Move init over to a non-isolated CPU */
	if (set_cpus_allowed_ptr(current, non_isolated_cpus) < 0)
		BUG();
	sched_init_granularity();
	free_cpumask_var(non_isolated_cpus);

	init_sched_rt_class();
}
#else
void __init sched_init_smp(void)
{
	sched_init_granularity();
}
```

kernel/sched/fair.c:

```
/*
 * Check this_cpu to ensure it is balanced within domain. Attempt to move
 * tasks if there is an imbalance.
 */
static int load_balance(int this_cpu, struct rq *this_rq,
                  struct sched_domain *sd, enum cpu_idle_type idle,
                  int *balance)
{
      int ld_moved, cur_ld_moved, active_balance = 0;
      struct sched_group *group;
      struct rq *busiest;
      unsigned long flags;
      struct cpumask *cpus = __get_cpu_var(load_balance_mask);

      struct lb_env env = {
            .sd         = sd,
            .dst_cpu    = this_cpu,
            .dst_rq           = this_rq,
            .dst_grpmask    = sched_group_cpus(sd->groups),
            .idle       = idle,
            .loop_break = sched_nr_migrate_break,
            .cpus       = cpus,
      };

      /*
       * For NEWLY_IDLE load_balancing, we don't need to consider
       * other cpus in our group
       */
      if (idle == CPU_NEWLY_IDLE)
            env.dst_grpmask = NULL;

      cpumask_copy(cpus, cpu_active_mask);

      schedstat_inc(sd, lb_count[idle]);

redo:
      group = find_busiest_group(&env, balance);

      if (*balance == 0)
            goto out_balanced;
```

```
	if (!group) {
		schedstat_inc(sd, lb_nobusyg[idle]);
		goto out_balanced;
	}

	busiest = find_busiest_queue(&env, group);
	if (!busiest) {
		schedstat_inc(sd, lb_nobusyq[idle]);
		goto out_balanced;
	}

	BUG_ON(busiest == env.dst_rq);

	schedstat_add(sd, lb_imbalance[idle], env.imbalance);

	ld_moved = 0;
	if (busiest->nr_running > 1) {
		/*
		 * Attempt to move tasks. If find_busiest_group has found
		 * an imbalance but busiest->nr_running <= 1, the group is
		 * still unbalanced. ld_moved simply stays zero, so it is
		 * correctly treated as an imbalance.
		 */
		env.flags |= LBF_ALL_PINNED;
		env.src_cpu   = busiest->cpu;
		env.src_rq    = busiest;
		env.loop_max  = min(sysctl_sched_nr_migrate, busiest->nr_running);

		update_h_load(env.src_cpu);
more_balance:
		local_irq_save(flags);
		double_rq_lock(env.dst_rq, busiest);

		/*
		 * cur_ld_moved - load moved in current iteration
		 * ld_moved     - cumulative load moved across iterations
		 */
		cur_ld_moved = move_tasks(&env);
		ld_moved += cur_ld_moved;
		double_rq_unlock(env.dst_rq, busiest);
		local_irq_restore(flags);
```

```
/*
 * some other cpu did the load balance for us.
 */
if (cur_ld_moved && env.dst_cpu != smp_processor_id())
      resched_cpu(env.dst_cpu);

if (env.flags & LBF_NEED_BREAK) {
      env.flags &= ~LBF_NEED_BREAK;
      goto more_balance;
}

/*
 * Revisit (affine) tasks on src_cpu that couldn't be moved to
 * us and move them to an alternate dst_cpu in our sched_group
 * where they can run. The upper limit on how many times we
 * iterate on same src_cpu is dependent on number of cpus in our
 * sched_group.
 *
 * This changes load balance semantics a bit on who can move
 * load to a given_cpu. In addition to the given_cpu itself
 * (or a ilb_cpu acting on its behalf where given_cpu is
 * nohz-idle), we now have balance_cpu in a position to move
 * load to given_cpu. In rare situations, this may cause
 * conflicts (balance_cpu and given_cpu/ilb_cpu deciding
 * _independently_ and at _same_ time to move some load to
 * given_cpu) causing exceess load to be moved to given_cpu.
 * This however should not happen so much in practice and
 * moreover subsequent load balance cycles should correct the
 * excess load moved.
 */
if ((env.flags & LBF_SOME_PINNED) && env.imbalance > 0) {

      env.dst_rq   = cpu_rq(env.new_dst_cpu);
      env.dst_cpu  = env.new_dst_cpu;
      env.flags   &= ~LBF_SOME_PINNED;
      env.loop     = 0;
      env.loop_break     = sched_nr_migrate_break;

      /* Prevent to re-select dst_cpu via env's cpus */
      cpumask_clear_cpu(env.dst_cpu, env.cpus);

      /*
```

```
		 * Go back to "more_balance" rather than "redo" since we
		 * need to continue with same src_cpu.
		 */
		goto more_balance;
	}

	/* All tasks on this runqueue were pinned by CPU affinity */
	if (unlikely(env.flags & LBF_ALL_PINNED)) {
		cpumask_clear_cpu(cpu_of(busiest), cpus);
		if (!cpumask_empty(cpus)) {
			env.loop = 0;
			env.loop_break = sched_nr_migrate_break;
			goto redo;
		}
		goto out_balanced;
	}
}

if (!ld_moved) {
	schedstat_inc(sd, lb_failed[idle]);
	/*
	 * Increment the failure counter only on periodic balance.
	 * We do not want newidle balance, which can be very
	 * frequent, pollute the failure counter causing
	 * excessive cache_hot migrations and active balances.
	 */
	if (idle != CPU_NEWLY_IDLE)
		sd->nr_balance_failed++;

	if (need_active_balance(&env)) {
		raw_spin_lock_irqsave(&busiest->lock, flags);

		/* don't kick the active_load_balance_cpu_stop,
		 * if the curr task on busiest cpu can't be
		 * moved to this_cpu
		 */
		if (!cpumask_test_cpu(this_cpu,
				tsk_cpus_allowed(busiest->curr))) {
			raw_spin_unlock_irqrestore(&busiest->lock,
						    flags);
			env.flags |= LBF_ALL_PINNED;
			goto out_one_pinned;
```

```
			}

			/*
			 * ->active_balance synchronizes accesses to
			 * ->active_balance_work.  Once set, it's cleared
			 * only after active load balance is finished.
			 */
			if (!busiest->active_balance) {
				busiest->active_balance = 1;
				busiest->push_cpu = this_cpu;
				active_balance = 1;
			}
			raw_spin_unlock_irqrestore(&busiest->lock, flags);

			if (active_balance) {
				stop_one_cpu_nowait(cpu_of(busiest),
					active_load_balance_cpu_stop, busiest,
					&busiest->active_balance_work);
			}

			/*
			 * We've kicked active balancing, reset the failure
			 * counter.
			 */
			sd->nr_balance_failed = sd->cache_nice_tries+1;
		}
	} else
		sd->nr_balance_failed = 0;

	if (likely(!active_balance)) {
		/* We were unbalanced, so reset the balancing interval */
		sd->balance_interval = sd->min_interval;
	} else {
		/*
		 * If we've begun active balancing, start to back off. This
		 * case may not be covered by the all_pinned logic if there
		 * is only 1 task on the busy runqueue (because we don't call
		 * move_tasks).
		 */
		if (sd->balance_interval < sd->max_interval)
			sd->balance_interval *= 2;
	}
```

```
	goto out;

out_balanced:
	schedstat_inc(sd, lb_balanced[idle]);

	sd->nr_balance_failed = 0;

out_one_pinned:
	/* tune up the balancing interval */
	if (((env.flags & LBF_ALL_PINNED) &&
			sd->balance_interval < MAX_PINNED_INTERVAL) ||
			(sd->balance_interval < sd->max_interval))
		sd->balance_interval *= 2;

	ld_moved = 0;
out:
	return ld_moved;
}
```

```c
/*
 * This routine will record that the cpu is going idle with tick stopped.
 * This info will be used in performing idle load balancing in the future.
 */
void nohz_balance_enter_idle(int cpu)
{
	/*
	 * If this cpu is going down, then nothing needs to be done.
	 */
	if (!cpu_active(cpu))
		return;

	if (test_bit(NOHZ_TICK_STOPPED, nohz_flags(cpu)))
		return;

	cpumask_set_cpu(cpu, nohz.idle_cpus_mask);
	atomic_inc(&nohz.nr_cpus);
	set_bit(NOHZ_TICK_STOPPED, nohz_flags(cpu));
}
```

kernel/stop_machine.c:

```
/**
 * stop_machine_from_inactive_cpu - stop_machine() from inactive CPU
 * @fn: the function to run
 * @data: the data ptr for the @fn()
 * @cpus: the cpus to run the @fn() on (NULL = any online cpu)
 *
 * This is identical to stop_machine() but can be called from a CPU which
 * is not active.  The local CPU is in the process of hotplug (so no other
 * CPU hotplug can start) and not marked active and doesn't have enough
 * context to sleep.
 *
 * This function provides stop_machine() functionality for such state by
 * using busy-wait for synchronization and executing @fn directly for local
 * CPU.
 *
 * CONTEXT:
 * Local CPU is inactive.  Temporarily stops all active CPUs.
 *
 * RETURNS:
 * 0 if all executions of @fn returned 0, any non zero return value if any
 * returned non zero.
 */
int stop_machine_from_inactive_cpu(int (*fn)(void *), void *data,
                          const struct cpumask *cpus)
{
      struct stop_machine_data smdata = { .fn = fn, .data = data,
                                   .active_cpus = cpus };
      struct cpu_stop_done done;
      int ret;

      /* Local CPU must be inactive and CPU hotplug in progress. */
      BUG_ON(cpu_active(raw_smp_processor_id()));
      smdata.num_threads = num_active_cpus() + 1;    /* +1 for local */

      /* No proper task established and can't sleep - busy wait for lock. */
      while (!mutex_trylock(&stop_cpus_mutex))
            cpu_relax();

      /* Schedule work on other CPUs and execute directly for local CPU */
      set_state(&smdata, STOPMACHINE_PREPARE);
      cpu_stop_init_done(&done, num_active_cpus());
```

```
	queue_stop_cpus_work(cpu_active_mask, stop_machine_cpu_stop, &smdata,
			     &done, true);
	ret = stop_machine_cpu_stop(&smdata);

	/* Busy wait for completion. */
	while (atomic_read(&done.nr_todo))
		cpu_relax();

	mutex_unlock(&stop_cpus_mutex);
	return ret ?: done.ret;
}
```