

07 Process Scheduling

Outline

- **Scheduling Policy**
- The Scheduling Algorithm
- Data Structures Used by the Scheduler
- Functions Used by the Scheduler
- Runqueue Balancing in Multiprocessor Systems



Scheduling Policy – Time Sharing

- In the current Linux, the system tick is set to 1 ms.
- Scheduling policy is the set of rules used to determine when and how to select a new process.
- Linux scheduling is based on the *time sharing* technique, which divides the CPU time into *time slices or quanta*.
  - If a currently running process is not terminated when its time slice or *quantum* expires, a process switch may take place.
  - Time sharing relies on timer interrupts and is thus transparent to processes.
- In Linux, process priority is dynamic.
  - The scheduler keeps track of what processes are doing and adjusts their priorities periodically.
- Processes are classified as I/O-bound or CPU-bound.



Classes of Processes

- *Interactive processes*
  - Interact constantly with their users.
  - The average delay must fall between 50 and 150 milliseconds so as to be responsive.
  - Typical interactive programs are command shells, text editors, and graphical applications.
- *Batch processes*
  - Do not need user interaction, and often run in the background.
  - Typical batch programs are programming language compilers, database search engines, and scientific computations.
- *Real-time processes*
  - Should never be blocked by lower-priority processes and should have a short guaranteed response time with a minimum variance.
  - Typical RT programs are video/sound applications, robot controllers.



System Calls Related Scheduling

System call	Description
nice( )	Change the static priority of a conventional process
getpriority( )	Get the maximum static priority of a group of conventional processes
setpriority( )	Set the static priority of a group of conventional processes
sched_getscheduler( )	Get the scheduling policy of a process
sched_setscheduler( )	Set the scheduling policy and the real-time priority of a process
sched_getparam( )	Get the real-time priority of a process
sched_setparam( )	Set the real-time priority of a process
sched_yield( )	Relinquish the processor voluntarily without blocking
sched_get_priority_min( )	Get the minimum real-time priority value for a policy
sched_get_priority_max( )	Get the maximum real-time priority value for a policy
sched_rr_get_interval( )	Get the time quantum value for the Round Robin policy
sched_setaffinity( )	Set the CPU affinity mask of a process
sched_getaffinity( )	Get the CPU affinity mask of a process



Process Preemption

- When a process enters the TASK_RUNNING state and its priority is granter than the currently running process, the execution of *current* is interrupted and the scheduler is invoked to select another process to run.
- A process also may be preempted when its time quantum expires.
  - The TIF_NEED_RESCHED flag in the *thread_info* structure of the current process is set, so the scheduler is invoked when the timer interrupt handler terminates.
- The Linux 2.6 kernel is preemptive, which means that a process can be preempted either when executing in Kernel Mode or in User Mode.



How Long Must a Quantum?

- The quantum duration is critical for system performance.
  - For instance, suppose that a process switch requires 5 milliseconds and the quantum is also set to 5 milliseconds: 50% overhead.
- The choice of the average quantum duration is always a compromise.
  - The rule of thumb adopted by Linux is to choose a duration as long as possible, while keeping good system response time.

Outline

- Scheduling Policy
- **The Scheduling Algorithm**
- Data Structures Used by the Scheduler
- Functions Used by the Scheduler
- Runqueue Balancing in Multiprocessor Systems



Scheduling Algorithm

- In Linux 2.6:
  - Selects the process to run in constant time, independently of the number of runnable processes.
  - Scales well with the number of processors because each CPU has its own queue of runnable processes.
- The new algorithm does a better job of distinguishing interactive processes and batch processes.
- At least one runnable process, the *swapper* process, which has PID 0 and executes only when the CPU cannot execute other processes.
  - Every CPU of a multiprocessor system has its own *swapper* process with PID equal to 0.



Scheduling Classes of Linux Processes

- SCHED_FIFO
  - A First-In, First-Out real-time process.
- SCHED_RR
  - A Round Robin real-time process.
- SCHED_NORMAL
  - A conventional, time-shared process.



Scheduling of Conventional Processes

- Every conventional process has its own *static priority*, which is a value used by the scheduler to rate the process.
  - Ranging from 100 (highest priority) to 139 (lowest priority).
- A new process always inherits the static priority of its parent.
  - The nice( ) and setpriority( ) system calls can the static priority.
  - E.g., shell command: nice –n 5 vi &



Base Time Quantum

- The static priority essentially determines the *base time quantum* of a process.

$$\text{base time quantum (in milliseconds)} = \begin{cases} (140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\ (140 - \text{static priority}) \times 5 & \text{if static priority} \geq 120 \end{cases} \quad (1)$$

- The higher the static priority, the longer the base time quantum.

Typical priority values for a conventional process

Description	Static priority	Nice value	Base time quantum	Interactivedelta	Sleep time threshold
Highest static priority	100	−20	800 ms	−3	299 ms
High static priority	110	-10	600 ms	-1	499 ms
Default static priority	120	0	100 ms	+2	799 ms
Low static priority	130	+10	50 ms	+4	999 ms
Lowest static priority	139	+19	5 ms	+6	1199 ms



Dynamic Priority and Average Sleep Time

- A conventional process also has a *dynamic priority*.
  - Ranging from 100 (highest priority) to 139 (lowest priority).
- The dynamic priority is the number looked up by the scheduler when selecting the new process to run.

$$dynamic\ priority = \max(100, \min(static\ priority - bonus + 5, 139)) \qquad (2)$$

  - The *bonus* is a value ranging from 0 to 10.
- Average sleep time
  - The value of the bonus is related to the *average sleep time* of the process.
  - The average sleep time decreases while a process is running.
  - The average sleep time can is no larger 1 second.

Dynamic Priority and Average Sleep Time (Cont.)

Unit: time slice

Average sleep times, bonus values, and time slice granularity

Average sleep time	Bonus	Granularity
Greater than or equal to 0 but smaller than 100 ms	0	5120
Greater than or equal to 100 ms but smaller than 200 ms	1	2560
Greater than or equal to 200 ms but smaller than 300 ms	2	1280
Greater than or equal to 300 ms but smaller than 400 ms	3	640
Greater than or equal to 400 ms but smaller than 500 ms	4	320
Greater than or equal to 500 ms but smaller than 600 ms	5	160
Greater than or equal to 600 ms but smaller than 700 ms	6	80
Greater than or equal to 700 ms but smaller than 800 ms	7	40
Greater than or equal to 800 ms but smaller than 900 ms	8	20
Greater than or equal to 900 ms but smaller than 1000 ms	9	10
1 second	10	10



Average Sleep Time

- A process is considered "interactive" if it satisfies:

$$dynamic\ priority \leq 3 \times static\ priority / 4 + 28 \qquad (3)$$

or

$$bonus - 5 \geq static\ priority / 4 - 28$$

Interactive delta (pp. 12)

- A process having highest static priority (100) is considered interactive when its bonus value exceeds 2 (or when its average sleep time exceeds 200 ms).
- A process having lowest static priority (139) is never considered as interactive, because the bonus value is always smaller than the value 11.

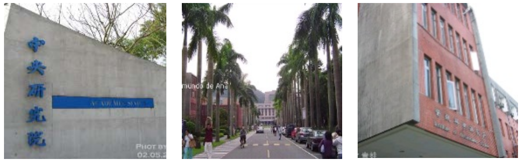

Active and Expired Processes

- The scheduler keeps two disjoint sets of runnable processes:
  - *Active processes*
    - These runnable processes have not yet exhausted their time quantum.
  - *Expired processes*
    - These runnable processes have exhausted their time quantum.
- An active batch process that finishes its time quantum always becomes expired.
- An active interactive process that finishes its time quantum usually remains active.
  - The scheduler refills its time quantum and leaves it in the set of active processes unless some expired processes have already waited for a long time.



Scheduling of Real-Time Processes

- Every real-time process has a *real-time priority.*
  - Ranging from 1 (highest priority) to 99 (lowest priority).
- Teal-time processes are always considered active.
- The sched_setparam() and sched_setscheduler() system calls can change the real-time priority of a process.
  - If several real-time runnable processes have the same highest priority, the scheduler chooses the process that occurs first.
- A real-time process is replaced at some conditions:
  - Preempted by another process with higher real-time priority.
  - The process performs a blocking operation and is put to sleep.
  - The process is stopped or killed.
  - The process voluntarily relinquishes the CPU with sched_yield()
  - The process is SCHED_RR and has exhausted its time quantum.



Outline

- Scheduling Policy
- The Scheduling Algorithm
- **Data Structures Used by the Scheduler**
- Functions Used by the Scheduler
- Runqueue Balancing in Multiprocessor Systems

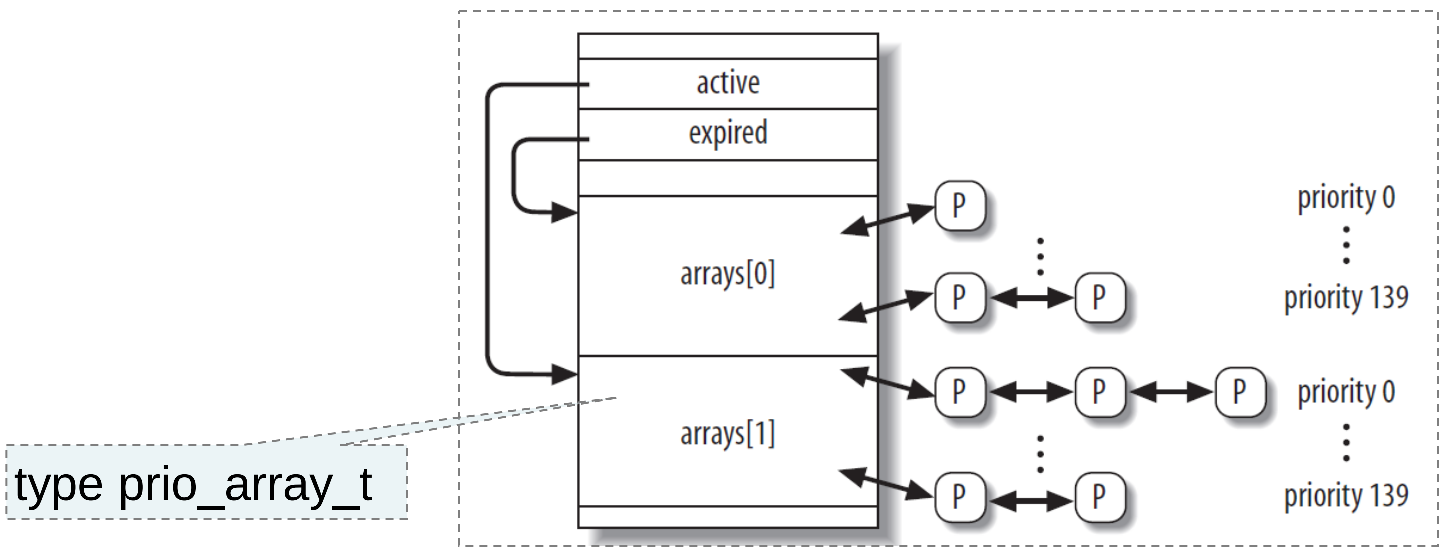


The Runqueue Data Structure

- Each CPU in the system has its own runqueue.
- All runqueue structures are stored in the *runqueues* per-CPU variable.
  - The this_rq() macro yields the address of the runqueue of the local CPU.
  - The cpu_rq(n) macro yields the address of the runqueue of the CPU having index *n*.
- Every runnable process in the system belongs to one runqueue.
- As long as a runnable process remains in the same runqueue, it can be executed only by the CPU owning that runqueue.

The Runqueue Data Structure (Cont.)

- The *arrays* field of the *runqueue* is an array consisting of two *prio_array_t* structures.
  - The active field points to the set of active processes.
  - The expired field points to the set of expired processes.
- The role of the two data structures in arrays changes by exchanging active and expired fields.



The runqueue structure and the two sets of runnable processes

Fields of the Runqueue Structure

Type	Name	Description
spinlock_t	lock	Spin lock protecting the lists of processes
unsigned long	nr_running	Number of runnable processes in the runqueue lists
unsigned long	cpu_load	CPU load factor based on the average number of processes in the runqueue
unsigned long	nr_switches	Number of process switches performed by the CPU
unsigned long	nr_uninterruptible	Number of processes that were previously in the run-queue lists and are now sleeping in TASK_UNINTERRUPTIBLE state (only the sum of these fields across all runqueues is meaningful)
unsigned long	expired_timestamp	Insertion time of the eldest process in the expired lists
unsigned long long	timestamp_last_tick	Timestamp value of the last timer interrupt
task_t *	curr	Process descriptor pointer of the currently running process (same as current for the local CPU)
task_t *	idle	Process descriptor pointer of the *swapper* process for this CPU
struct mm_struct *	prev_mm	Used during a process switch to store the address of the memory descriptor of the process being replaced

Fields of the Runqueue Structure (Cont.)

Type	Name	Description
prio_array_t *	active	Pointer to the lists of active processes
prio_array_t *	expired	Pointer to the lists of expired processes
prio_array_t [2]	arrays	The two sets of active and expired processes
int	best_expired_prio	The best static priority (lowest value) among the expired processes
atomic_t	nr_iowait	Number of processes that were previously in the run-queue lists and are now waiting for a disk I/O operation to complete
struct sched_domain *	sd	Points to the base scheduling domain of this CPU (see the section "Scheduling Domains" later in this chapter)
int	active_balance	Flag set if some process shall be *migrated* from this run-queue to another (runqueue balancing)
int	push_cpu	Not used
task_t *	migration_thread	Process descriptor pointer of the *migration* kernel thread
struct list_head	migration_queue	List of processes to be removed from the runqueue



Fields Process Descriptor for Scheduling

Type	Name	Description
unsigned long	thread_info->flags	Stores the TIF_NEED_RESCHED flag, which is set if the scheduler must be invoked (see the section "Returning from Interrupts and Exceptions" in Chapter 4)
unsigned int	thread_info->cpu	Logical number of the CPU owning the runqueue to which the runnable process belongs
unsigned long	state	The current state of the process (see the section "Process State" in Chapter 3)
int	prio	Dynamic priority of the process
int	static_prio	Static priority of the process
struct list_head	run_list	Pointers to the next and previous elements in the runqueue list to which the process belongs
prio_array_t *	array	Pointer to the runqueue's prio_array_t set that includes the process
unsigned long	sleep_avg	Average sleep time of the process
unsigned long long	timestamp	Time of last insertion of the process in the runqueue, or time of last process switch involving the process
unsigned long long	last_ran	Time of last process switch that replaced the process
int	activated	Condition code used when the process is awakened
unsigned long	policy	The scheduling class of the process (SCHED_NORMAL, SCHED_RR, or SCHED_FIFO)
cpumask_t	cpus_allowed	Bit mask of the CPUs that can execute the process
unsigned int	time_slice	Ticks left in the time quantum of the process
unsigned int	first_time_slice	Flag set to 1 if the process never exhausted its time quantum
unsigned long	rt_priority	Real-time priority of the process



Process Descriptor

- When a new process is created, *sched_fork()*, invoked by copy_process( ), sets the *time_slice* field of both *current* (the parent) and *p* (the child) processes.
- The number of ticks left to the parent is split in two halves.

```
p->time_slice = (current->time_slice + 1) >> 1;
current->time_slice >>= 1;
```

  – To prevent any process from refreshing time quantum by creating new child processes.

- The copy_process() function also initializes a few other fields of the child's process descriptor related to scheduling:

```
p->first_time_slice = 1;
p->timestamp = sched_clock();
```

The child has never exhausted its time quantum.

Initialize the child's timestamp.



Outline

- Scheduling Policy
- The Scheduling Algorithm
- Data Structures Used by the Scheduler
- **Functions Used by the Scheduler**
- Runqueue Balancing in Multiprocessor Systems



Important Scheduling Functions

`scheduler_tick()`
: Keeps the `time_slice` counter of `current` up-to-date

`try_to_wake_up()`
: Awakens a sleeping process

`recalc_task_prio()`
: Updates the dynamic priority of a process

`schedule()`
: Selects a new process to be executed

`load_balance()`
: Keeps the runqueues of a multiprocessor system balanced



The scheduler_tick( ) Function (1/5)

- The scheduler_tick() is invoked every tick:
  - 1. Stores in the timestamp_last_tick field of the local runqueue the current value of the TSC converted to nanoseconds, by invodking sched_clock().
  - 2. Checks if the *current* process is the *swapper* of the local CPU. If so.
    - a. it sets the TIF_NEED_RESCHED flag of the current process to force rescheduling if the local runqueue has another runnable process.
    - b. Jumps to step 7.
  - 3. Checks whether current->array points to the active list of the local runqueue. If not, set the TIF_NEED_RESCHED flag to force rescheduling, and jumps to step 7.
  - 4. Acquires the this_rq()->lock spin lock.
    - 5. Decreases the time slice counter of the current process.
  - 6. Releases the this_rq()->lock spin lock.
  - 7. Invokes the rebalance_tick() function to rebalance the number runnable processes on various CPUs.

Updating the Time Slice of a Real-time Process

- If the current process is a FIFO real-time process, scheduler_tick() has nothing to do.
- If current is a Round Robin real-time process, scheduler_tick()
  - (1) decreases its time slice counter and
  - (2) checks whether the quantum is exhausted:

```
if (current->policy == SCHED_RR && !--current->time_slice) {
    current->time_slice = task_timeslice(current);
    current->first_time_slice = 0;
    set_tsk_need_resched(current);
    list_del(&current->run_list);
    list_add_tail(&current->run_list,
                  this_rq()->active->queue+current->prio);
}
```

Time slice exhausted

Refill time slice

Set TIF_NEED_RESCHED flag

Enqueue into runqueue of the corresponding array



Updating the Time Slice of a Conventional Process

- 1. Decreases the time slice counter. (current->time_slice)
- 2. If the time quantum is exhausted:
  - a. Invokes dequeue_task() to remove current from the this_rq()->active set of runnable processes.
  - b. Invokes set_tsk_need_resched().
  - c. Updates the dynamic priority:

```
current->prio = effective_prio(current);
```

  - d. Refills the time quantum of the process:

```
current->time_slice = task_timeslice(current);
current->first_time_slice = 0;
```

  - e. If current->expired_timestamp = 0, update the expire time:

```
if (!this_rq()->expired_timestamp)
    this_rq()->expired_timestamp = jiffies;
```

  - f. Inserts the current process either in the active or in the expired set:

```
if (!TASK_INTERACTIVE(current) || EXPIRED_STARVING(this_rq())) {
    enqueue_task(current, this_rq()->expired);
    if (current->static_prio < this_rq()->best_expired_prio)
        this_rq()->best_expired_prio = current->static_prio;
} else
    enqueue_task(current, this_rq()->active);
```

Not Interactive process?

Some expired process is starving



Updating the Time Slice of a Conventional Process (Cont.)

- 3. if the time quantum is not exhausted, check whether the remaining time slice of the current process is too long:

```
if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
         p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
        (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
        (p->array == rq->active)) {
    list_del(&current->run_list);
    list_add_tail(&current->run_list,
                  this_rq()->active->queue+current->prio);
    set_tsk_need_resched(p);
}
```

The remaining time slice is too long

Put into the tail of runqueue

Set TIF_NEED_RESCHED flag



The try_to_wake_up( ) Function (2/5)

- Awake a sleeping or stopped process by setting its state to TASK_RUNNING and inserting it into the runqueue of the local CPU.
- Parameters:
  - The descriptor pointer (p) of the process to be awakened
  - A mask of the process states (state) that can be awakened
  - A flag (sync) that forbids the awakened process to preempt the process currently running on the local CPU



The try_to_wake_up( ) Function (2/5) (Cont.)

- 1. Invokes the task_rq_lock() function to disable local interrupts.
- 2. Checks if the state of the process p->state belongs to the mask of states *state*.
- 3. If the p->array field is not NULL, the process already belongs to a runqueue. Jump Step 8.
- 4. It checks whether the process to be awakened should be migrated in multiprocessor systems.
- 5. If the process is in the TASK_UNINTERRUPTIBLE state, it decreases the nr_uninterruptible field of the target runqueue.
- 6. Invokes the activate_task() function to activate the process.
- 7. Check whether reschedule is needed.
- 8. Sets the p->state field of the process to TASK_RUNNING.
- 9. Invokes task_rq_unlock() to unlock the runqueue and reenable the local interrupts; and then return.



The recalc_task_prio() Function (3/5)

- To update the average sleep time and the dynamic priority of a process.
- Parameter:
  - descriptor pointer p
  - timestamp now
- Operations:
  - 1. Stores in *sleep_time* local variable min(now − p->timestamp, 10^9)
  - 2. Checks (1) whether the process is not a kernel thread, (2) whether it is awakening from the TASK_UNINTERRUPTIBLE state, and (3) whether it has been continuously asleep beyond a given sleep time threshold: Set p->sleep_avg field to the equivalent of 900 ticks (an empirical value). Then jump to Step 8.
    - The empirical rule is to ensure that processes having been asleep for a long time in uninterruptible mode have a reasonable sleep average value.

sleep_time stores the number of nanoseconds that the process spent sleeping since its last execution

The timestamp of the process switch that put the process to sleep

The recalc_task_prio() Function (3/5) (Cont.)

- 4. Executes the *CURRENT_BONUS* macro to compute the *bonus* value of the previous average sleep time of the process.
- 5. If the process is in TASK_UNINTERRUPTIBLE mode and it is not a kernel thread, it limits the increment of the average sleep time of the process so as to prevent rewarding too much batch processes.
  - If the sum sleep_time + p->sleep_avg is greater than or equal to the sleep time threshold, it sets the p->sleep_avg = sleep time threshold, and sets sleep_time =0.
- 6. Adds sleep_time to the average sleep time of the process (p->sleep_avg).
- 7. Checks whether p->sleep_avg exceeds 1000 ticks: cut down to 1000 ticks.
- 8. Updates the dynamic priority of the process: `p->prio = effective_prio(p);`



The schedule( ) Function (4/5)

- The schedule( ) function implements the scheduler.
- *Direct invocation*
  - The scheduler is invoked directly when the current process must be blocked right away because the resource it needs is not available.
  - Steps:
    - 1. Inserts *current* in the proper wait queue.
    - 2. Changes the state of current either to TASK_INTERRUPTIBLE or to TASK_UNINTERRUPTIBLE.
    - 3. Invokes schedule().
    - 4. Checks whether the resource is available. If not, go to Step 2.
    - 5. Once the resource is available, remove current from the wait queue.



The schedule( ) Function (4/5) (Cont.)

- *Lazy invocation*
  - The scheduler can is invoked in a lazy way by setting the TIF_NEED_RESCHED flag of *current*.
  - Typical examples of lazy invocation:
    - When current has used up its quantum of CPU time: done by the *scheduler_tick()* function.
    - When a process is woken up and its priority is higher than that of the current process: done by the *try_to_wake_up()* function.
    - When a *sched_setscheduler( )* system call is issued.



Actions Performed by schedule( ) before a Process Switch

- The key outcome of the function is to set a local variable called *next*, so that it points to the descriptor of the process selected to replace *current*.
- The schedule() function starts by

```
need_resched:

preempt_disable();
prev = current;
rq = this_rq();
```

Disable interrupt

Put current to prev

Acquire the runqueue

```
if (prev->lock_depth >= 0)
    up(&kernel_sem);
```

:Makes sure that prev doesn’t hold the big kernel lock.



Actions Performed by schedule( ) before a Process Switch (Cont.)

- The sched_clock() function is invoked to read the TSC. The *run_time* value is used to charge the process for the CPU usage.

```
now = sched_clock();
run_time = now - prev->timestamp;
if (run_time > 1000000000)
    run_time = 1000000000;
```

- A process having a high average sleep time is favored:

```
run_time /= (CURRENT_BONUS(prev) ? : 1);
```

  - *CURRENT_BONUS* returns a value between 0 and 10 that is proportional to the average sleep time of the process.

- Before starting to look at the runnable processes, schedule() must disable the local interrupts:

```
spin_lock_irq(&rq->lock);
```



Actions Performed by schedule( ) before a Process Switch (Cont.)

- Look into the PF_DEAD flag to see whether *prev* is a to be terminated process.

```
if (prev->flags & PF_DEAD)
    prev->state = EXIT_DEAD;
```

not runnable and it has not been preempted in Kernel Mode

```
if (prev->state != TASK_RUNNING &&
    !(preempt_count() & PREEMPT_ACTIVE)) {
    if (prev->state == TASK_INTERRUPTIBLE && signal_pending(prev))
        prev->state = TASK_RUNNING;
    else {
        if (prev->state == TASK_UNINTERRUPTIBLE)
            rq->nr_uninterruptible++;
        deactivate_task(prev, rq);
    }
}
```

It has nonblocked pending signals and its state is TASK_INTERRUPTIBLE

deactivate_task function:

```
rq->nr_running--;
dequeue_task(p, p->array);
p->array = NULL;
```

Actions Performed by schedule( ) before a Process Switch (Cont.)

- Check the number of runnable processes left in the runqueue.
  - If there are some runnable processes, the function invokes the dependent_sleeper() function:

```
if (rq->nr_running) {
    if (dependent_sleeper(smp_processor_id(), rq)) {
        next = rq->idle;
        goto switch_tasks;
    }
}
```

Process descriptor pointer of the swapper process

If *hyper-threading* is supported, check the process that is going to be selected for execution has significantly lower priority than a sibling process already running on a logical CPU of the same physical CPU.

```
if (!rq->nr_running) {
    idle_balance(smp_processor_id(), rq);
    if (!rq->nr_running) {
        next = rq->idle;
        rq->expired_timestamp = 0;
        wake_sleeping_dependent(smp_processor_id(), rq);
        if (!rq->nr_running)
            goto switch_tasks;
    }
}
```

No runnable process exists

Idle_balance() fails to move some process into the local runqueue.

The set of expired process is empty.

Reschedule runnable processes.

Actions Performed by schedule( ) before a Process Switch (Cont.)

- Let's suppose that the schedule() function has determined that the runqueue includes some runnable processes.

```
array = rq->active;
if (!array->nr_active) {
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = 140;
}
```

No runnable process in the active array

Exchange active and expired fields

The set of expired process is empty.

No expired process

- Find a runnable process:

based on the *bsfl* assembly language instruction, which returns the bit index of the least significant bit set to one in a 32-bit word.

```
idx = sched_find_first_bit(array->bitmap);
next = list_entry(array->queue[idx].next, task_t, run_list);
```



Actions Performed by schedule( ) before a Process Switch (Cont.)

- The schedule() function looks at the next->activated field.

Table 7-6. The meaning of the activated field in the process descriptor

Value	Description
0	The process was in TASK_RUNNING state.
1	The process was in TASK_INTERRUPTIBLE or TASK_STOPPED state, and it is being awakened by a system call service routine or a kernel thread.
2	The process was in TASK_INTERRUPTIBLE or TASK_STOPPED state, and it is being awakened by an interrupt handler or a deferrable function.
–1	The process was in TASK_UNINTERRUPTIBLE state and it is being awakened.

```
if (next->prio >= 100 && next->activated > 0) {
    unsigned long long delta = now - next->timestamp;
    if (next->activated == 1)
        delta = (delta * 38) / 128;
    array = next->array;
    dequeue_task(next, array);
    recalc_task_prio(next, next->timestamp + delta);
    enqueue_task(next, array);
}
next->activated = 0;
```

a process awakened by an interrupt handler or deferrable function. The scheduler adds the whole runqueue waiting time.

A process awakened by a system call service routine or a kernel thread. it adds just a fraction of that time.



Actions Performed by schedule() to Make the Process Switch

- Perform context switch, started from bring the contents of the first fields of next's process descriptor.

```
switch_tasks:

prefetch(next);
```

To hint the hardware cache.

- Do some administrative work:

```
clear_tsk_need_resched(prev);
rcu_qsctr_inc(prev->thread_info->cpu);
```

clears the TIF_NEED_RESCHED flag

the CPU is going through a quiescent state

- Decrease the avg sleep time

```
prev->sleep_avg -= run_time;
if ((long)prev->sleep_avg <= 0)
    prev->sleep_avg = 0;
prev->timestamp = prev->last_ran = now;
```

```
if (prev == next) {
    spin_unlock_irq(&rq->lock);
    goto finish_schedule;
}
```

Skip context switch



Actions Performed by schedule() to Make the Process Switch (Cont.)

- Perform context switch

```
next->timestamp = now;
rq->nr_switches++;
rq->curr = next;
prev = context_switch(rq, prev, next);
```

  – If next is a kernel thread:

```
if (!next->mm) {
    next->active_mm = prev->active_mm;
    atomic_inc(&prev->active_mm->mm_count);
    enter_lazy_tlb(prev->active_mm, next);
}
```

  – If next is a regular process:

```
if (next->mm)
    switch_mm(prev->active_mm, next->mm, next);
```

  – If prev is a kernel thread or an exiting process:

```
if (!prev->mm) {
    rq->prev_mm = prev->active_mm;
    prev->active_mm = NULL;
}
```

- Perform context switch:

```
switch_to(prev, next, prev);
return prev;
```



Actions Performed by schedule() after a Process Switch

```
barrier();
finish_task_switch(prev);
```

yields an optimization barrier for the code

- Finish_task_switch():

```
mm = this_rq()->prev_mm;
this_rq()->prev_mm = NULL;
prev_task_flags = prev->flags;
spin_unlock_irq(&this_rq()->lock);
if (mm)
    mmdrop(mm);
if (prev_task_flags & PF_DEAD)
    put_task_struct(prev);
```

If prev is a kernel thread, the prev_mm field of the runqueue stores the address of the memory descriptor that was lent to prev

decreases the usage counter of the memory descriptor

Free the process descriptor reference counter and drop all remaining references to the process

```
finish_schedule:

prev = current;
if (prev->lock_depth >= 0)
    __reacquire_kernel_lock();
preempt_enable_no_resched();
if (test_bit(TIF_NEED_RESCHED, &current_thread_info()->flags)
   goto need_resched;
return;
```

Outline

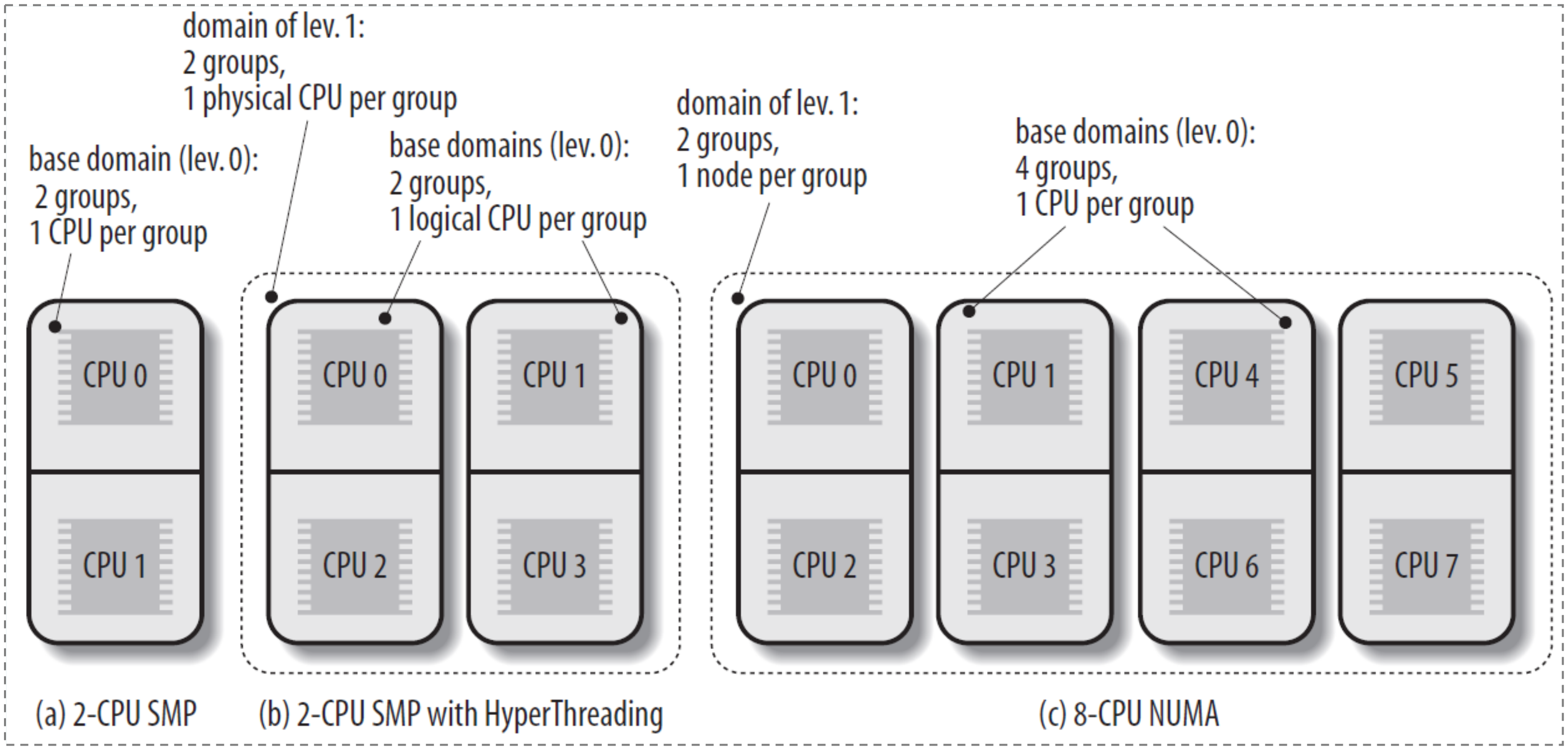
- Scheduling Policy
- The Scheduling Algorithm
- Data Structures Used by the Scheduler
- Functions Used by the Scheduler
- **Runqueue Balancing in Multiprocessor Systems**

Types of Multiprocessor Machines

- *Classic multiprocessor architecture*
  - These machines have a common set of RAM chips shared by all CPUs.
- *Hyper-threading*
  - A microprocessor that executes several threads of execution.
  - It includes several copies of the internal registers and quickly switches between them.
  - Allows the processor to exploit the machine cycles to execute another thread while the current thread is stalled for a memory access.
- *NUMA*
  - CPUs and RAM chips are grouped in local “nodes”.
  - When a CPU accesses a “local” RAM chip inside its own node, there is little or no contention.

Scheduling Domain

- A *scheduling domain* is a set of CPUs whose workloads should be kept balanced by the kernel.
- Every scheduling domain is partitioned, in turn, in one or more *groups*, each of which represents a subset of the CPUs of the scheduling domain.
- Workload balancing is always done between groups of a scheduling domain.





Scheduling Domain (Cont.)

- Every scheduling domain is represented by a sched_domain descriptor.
  - Every group inside a scheduling domain is represented by a sched_group descriptor.
  - Each sched_domain descriptor includes a field *groups*, which points to the first element in a list of group descriptors.
  - The parent field of the sched_domain structure points to the descriptor of the parent scheduling domain.



The rebalance_tick() Function

- To keep the runqueues in the system balanced.
- Invoked by by *scheduler_tick()* once every tick.
- Parameter:
  - The address *this_rq* of the local runqueue
  - A flag, *idle*, which can assume the following values
    - SCHED_IDLE
      - The CPU is currently idle. *current* is the swapper process.
    - NOT_IDLE
      - The CPU is not currently idle.



The rebalance_tick() Function (Cont.)

- Determine first the number of processes in the runqueue and updates the runqueue's average workload.
  - To do this, the function accesses the nr_running and cpu_load fields of the runqueue descriptor.
- Start a loop over all scheduling domains in the path from the base domain ((referenced by the *sd* field).
  - In each iteration the function determines whether the time has come to invoke the *load_balance()* function, thus executing a rebalancing operation on the scheduling domain.
  - If *idle* is equal to SCHED_IDLE, then the runqueue is empty, and rebalance_tick() invokes *load_balance()* quite often. (roughly once every10 milliseconds for scheduling domains corresponding to logical CPUs).

The load_balance() Function (5/5)

- The load_balance() function checks whether a scheduling domain is significantly unbalanced.
- It checks whether unbalancing can be reduced by moving some processes from the busiest group to the runqueue of the local CPU. If so, it attempts the migration.
- Parameter:
  - this_cpu
    - The index of the local CPU
  - this_rq
    - The address of the descriptor of the local runqueue
  - sd
    - Points to the descriptor of the scheduling domain to be checked
  - idle
    - Either SCHED_IDLE (local CPU is idle) or NOT_IDLE

The load_balance() Function (5/5) (Cont.)

- 1. Acquires the this_rq->lock spin lock.
- 2. Invokes the find_busiest_group() function to analyze the workloads of the groups inside the scheduling domain.
  - The function returns the address of the *sched_group* descriptor of the busiest group and the number of processes to be moved.
- 3. Release this_rq->lock if find_busiest_group() returns NULL.
- 4. Invokes the find_busiest_queue() to find the busiest CPUs in the group.
- 5. Acquires a second spin lock, namely the busiest->lock spin lock.
- 6. Invokes the *move_tasks()* function to try moving some processes from the busiest runqueue to the local runqueue this_rq.
- 7. If the move_task() function failed, wake up the *migration* kernel thread that walks the chain of the scheduling domain.
  - If an idle CPU is found, the kernel thread invokes move_tasks() to move one process into the idle runqueue.
- 8. Release the two locks.



The move_tasks() Function

- The move_tasks() function moves processes from a source runqueue to the local runqueue.
  – The function first analyzes the expired processes of the busiest runqueue, starting from the higher priority ones.
  – When all expired processes have been scanned, the function scans the active processes of the busiest runqueue.
  – For each candidate process, the function invokes can_migrate_task(), which returns 1 if all the following conditions hold:
    - The process is not being currently executed by the remote CPU.
    - The local CPU is included in the cpus_allowed bitmask of the process descriptor.
    - At least one of the following holds:
      - The local CPU is idle.
      - The kernel is having trouble in balancing the scheduling domain
      - The process to be moved is not “cache hot”.
  – If can_migrate_task() returns the value 1, move_tasks() invokes the pull_task() function to move the candidate process to the local runqueue.