Unit 4: Process Scheduling

This slice set is based on Prof. Shih-Kun Huang's material and the book "Understanding the Linux Kernel"

Scheduling Policy

schedule(): http://lxr.free-electrons.com/source/kernel/sched.c?v=2.6.32#L5696

sched.h: http://lxr.free-electrons.com/source/include/linux/sched.h?v=2.6.32

struct rq: http://lxr.free-electrons.com/source/kernel/sched.c?v=2.6.32#L484

Scheduling Policy

- Rules to determine when and how to select a new process to run
  - Among the processes in the TASK_RUNNING state
- Conflicting objectives
  - Fast response time (for foreground jobs)
  - Good throughput for background jobs
  - Avoidance of process starvation
  - Reconciliation of the needs of low- and high-priority processes

Scheduling Policy

- Linux Scheduling
  - Based on the time sharing technique; several processes run in "time multiplexing”
- A process is given a *time slice (or time quantum)* for running on the CPU, and is put sleep when the time slice depletes
  - Transparent to processes, need timer interrupts
- Linux scheduling policy
  - Based on ranking processes according to their priority
  - Dynamic and static process priority
  - scheduler keeps track of what processes are doing and adjusts their dynamic priorities periodically

Scheduling Policy

- Processes classified as I/O-bound or CPU-bound
- Alternative classification of three classes of processes:
  - Interactive processes:
    - Interact constantly with their users
    - Average delay between 50 ms and 150ms
  - Batch processes
    - Do not need user interaction, and often run in the background
  - Real-time processes
    - Timing predictability is the first concern

Scheduling Policy

Table 7-1. System calls related to 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

- A process enters TASK_RUNNING state
  - A newly created process
  - A process finishes its waiting
  - A process whose time quantum has been replenished
- The kernel checks whether its dynamic priority is greater than that of the currently running process
  - If it is, TIF_NEED_RESCHED flag in thread_info structure of the current process is set
  - The scheduler is invoked to select a new process for running

Process Preemption

- A process may also surrender the CPU when its time quantum expires
  - Scheduler invoked when the timer interrupt handler terminates
- A preempted process is *not* suspended
  - It remains in the TASK_RUNNING state, simply no longer use the CPU
- The Linux 2.6 *kernel* is preemptive
  - A process can be preempted either when executing in Kernel or in User Mode.

How Long Must a Quantum Last

- If too short, system overhead caused by process switches becomes high
- If too long, processes no longer appear to be executed concurrently

- The rule of thumb adopted by Linux
  - Choose a duration as long as possible
  - While keeping good system response time

The Scheduling Algorithm

The Scheduling Algorithm

- There is always at least one runnable process
  - The swapper (the idle process) whose PID is 0
  - Executes only when the CPU cannot execute other processes
  - Each CPU is with own swapper process with PID 0 (Question: how to handle this ?)

The Scheduling Algorithm

- Linux schedules every process with the following classes:
  - SCHED_FIFO
    - A First-In, First-Out real-time process
    - Runs on the CPU as long as it wishes
    - Priority-driven, preemption is still possible
  - SCHED_RR
    - A Round Robin real-time process
    - Ensures a fair share of CPU time (quantum) to all SCHED_RR real-time processes with the same priority
    - =SCHED_FIFO + time quantum
  - SCHED_NORMAL
    - A conventional, time-shared process
    - Uses dynamic priority

Scheduling of Conventional Processes

- Every conventional process has its own static priority
  - A number from 100 (highest priority) to 139 (lowest priority)
- Change the static priority of the processes by the "nice value"
  - nice( ) and setpriority( ) system calls
  - Being nice to other processes
- The static priority essentially determines the base time quantum of a process

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

Scheduling of Conventional Processes

Table 7-2. 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

Calculated from the formula in the last slice.
Higher priority processes have larger slices

=static priority /4 -28

Largest sleep time counted in priority calculation

Dynamic priority and average sleep time

- The longer a process sleeps, the higher its dynamic priority will be
  - To reward interactive processes
- Dynamic priority
  - A value ranging from 100 (highest priority) to 139 (lowest priority)
  - Looked up by the scheduler when selecting the new process to run

$$\text{Dynamic priority} = \max(100, \min(\text{static priority} - \text{bonus} + 5, 139))$$

Dynamic priority and average sleep time

Dynamic priority =
max (100, min (static priority – bonus + 5, 139))

- The bonus is a value ranging from 0 to 10
- Value less than 5 is a penalty that lowers the dynamic priority
- Value greater than 5 is a premium that raises the dynamic priority
- The value of the bonus, depends on the past history and is related to the average sleep time of the process

Dynamic priority and average sleep time

- The average sleep time
  - average number of nanoseconds that the process spent while sleeping
- The calculation of the sleep time is complicated
  - Running the process decreases the sleep time
  - Sleeping in TASK_INTERRUPTIBLE state contributes to the average sleep time in a different way from sleeping in TASK_UNINTERRUPTIBLE state
  - The sleep time has a ceiling of 1 second

Dynamic priority and average sleep time

Table 7-3. 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

Dynamic priority and average sleep time

- A process is considered "interactive" if it satisfies
  - dynamic priority <= 3 x static priority / 4 + 28
    - static priority – bonus + 5 <= 3 x static priority / 4 + 28
    - static priority / 4 - 28 <= bonus - 5
    - bonus - 5 >= static priority / 4 – 28
  - static priority/4 -28 : the interactive delta

Dynamic priority and average sleep time

- Interactive delta is static priority / 4 – 28
- How a process with the highest static priority (100) is considered interactive?
  - Interacrive deta = 100/4 – 28 = -3
  - bonus – 5 >= -3 ➔ bonus >= 2
  - when bonus value exceeds 2 (when its average sleep time exceeds 200 ms)
- A process of static priority 139 will never be interactive
  - Interactive delta = 6.75
  - bonus must be >= 11.75 (impossible, due to max bonus = 10 for max sleep time = 1 second)
- Higher priority is easier to become interactive (with shorter sleep time)
  - Uh-huh, but what's the benefit of being interactive???

Active and expired processes

- To avoid process starvation, when a process finishes its time quantum, it can be replaced by a lower priority process whose time quantum has not yet been exhausted
- Two disjoint sets of runnable processes:
  - Active processes
    - These runnable processes have not yet exhausted their time quantum and are allowed to run.
  - Expired processes
    - These runnable processes have exhausted their time quantum and are forbidden to run until all active processes expire

Active and expired processes

- An active batch process that finishes its time quantum always becomes expired
- An active interactive process that finishes its time quantum usually remains active!
  - Wow!
  - Except: If the eldest expired process has already waited for a long time
    - To prevent batch processes from starving

Scheduling of Real-Time Processes

- Scheduling real-time processes is different from scheduling conventional processes
  - Timing predictability is very important
  - In particular, preemption must be manageable

- Real-time scheduling is not about “fast”, it is about finishing tasks before their predefined deadlines
  - Hard real time vs. soft real time

Scheduling of Real-Time Processes

- Every real-time process is associated with a real-time priority from 1 (highest) to 99
  - They do not use dynamic priorities
  - They are always considered active
  - Programmers can change their priorities via sched_setparam( ) and sched_setscheduler( )
- Real-time processes are scheduled in a priority-driven manner
  - Preemption is possible
  - RT processes of the same priority are served in a first-come first-served manner

Scheduling of Real-Time Processes

- A real-time process is replaced by another process only when one of the following events occurs:
  - The process is preempted by another process having higher real-time priority
  - The process performs a blocking operation, and it is put to sleep (in state TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE).
  - The process is stopped (in state TASK_STOPPED or TASK_TRACED), or it is killed (in state EXIT_ZOMBIE or EXIT_DEAD).
  - The process voluntarily relinquishes the CPU by invoking the sched_yield( ) system
  - The process is Round Robin real-time (SCHED_RR), and it has exhausted its time quantum
    - Never happen for SCHED_FIFO processes

Data Structures Used by the Scheduler

The runqueue Data Structure

- The process list links all process descriptors
- The runqueue lists link the process descriptors of all processes in the TASK_RUNNING state, except the swapper (PID=0)
  - 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

Table 7-4. The 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 runqueue 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
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 runqueue 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 runqueue 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

The runqueue Data Structure

- Every CPU has one runqueue
- Every runnable process is in one runqueue
- As long as a runnable process remains in the same runqueue, it can be executed only by the CPU owning that runqueue.
- Runnable processes may migrate from one runqueue to another
  - For load balancing in multiprocessor systems

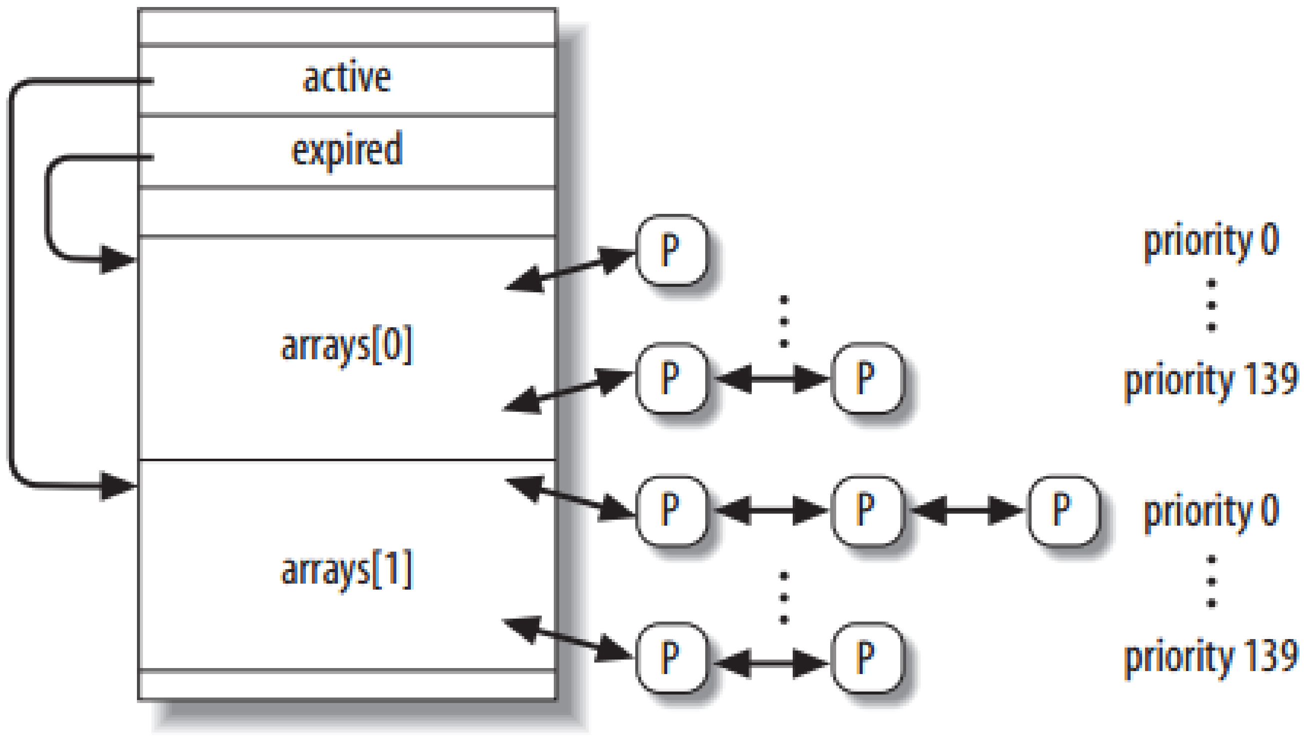
The runqueue Data Structure

- The arrays field of the runqueue is an array consisting of two prio_array_t structures
- Each data structure represents a set of runnable processes
  - 140 doubly linked list heads (one list for each possible process priority)
  - A priority bitmap
  - A counter of the processes included in the set

Table 3-2. The fields of the prio_array_t data structure

Type	Field	Description
`int`	`nr_active`	The number of process descriptors linked into the lists
`unsigned long [5]`	`bitmap`	A priority bitmap: each flag is set if and only if the corresponding priority list is not empty
`struct list_head [140]`	`queue`	The 140 heads of the priority lists

The runqueue Data Structure



The runqueue Data Structure

- The active field points to one of the two prio_array_t data structures in arrays:
  - The set of runnable processes includes the active processes
- The expired field points to the other prio_array_t data structure in arrays:
  - The set of runnable processes includes the expired processes
- The two pointers exchange when there is no runnable active processes

Table 7-5. Fields of the process descriptor related to the scheduler

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 in the following way:
  - p->time_slice = (current->time_slice + 1) >> 1;
  - current->time_slice >>= 1;
- The number of ticks left to the parent is split in two halves: one for the parent and one for the child.
- A process cannot hog resources (unless it has privileges to give itself a real-time policy) by forking multiple descendants

Functions Used by the Scheduler

Functions Used by the Scheduler

- The scheduler relies on several functions in order to do its work; the most important are:
  - 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.

scheduler_tick() function

- Invoked once every tick

1. Store the current TSC value in timestamp_last_tick of the local runqueue
2. The current process is the swapper of the local CPU?
   - If yes, check if another runnable process exists, set TIF_NEED_RESCHED flag
3. Checks if current->array is in active list, if not, time quantum has exhausted, TIF_NEED_RESCHED
4. Update the time slice counter of the current process, and check if the quantum is exhausted
5. Call rebalance_tick() to balance the number of processes in different CPUs

Updating the time slice of a real-time process

- Step 4 in the last slice
- FIFO RT processes need not to be updated
- RR RT processes:

```
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);
}
```

Updating Time Slice of a Conventional Process

- Decrease the time slice counter (current->time_slice)
- If time quantum is exhausted
  - dequeue_task() from this_rq()->active
  - set_tsk_need_resched() to set TIF_NEED_RESCHED flag
  - current->prio = effective_prio(current);
  - current->time_slice = task_timeslice(current)
  - current->first_time_slice = 0;
  - If (!this->rq()->expired_timestamp) this->rq()->expired_timestamp = jiffies;
  - Insert the process to the active or expired set (see the next slice)

Insert to the active or expired set

- Insert the process to the active or expired set
  - An active process is inserted to active list again, unless it is not or processes in the expired list are starving

```
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);
```

Divide A Large Time Slice

- If time quantum is not exhausted, check if the time slice is too long
  - If so, divide the time slice into small pieces and do process switches for better response
  - See Table 7-3 for the definition of the granularity

```
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 try_to_wake_up() function

- Awake a sleeping or stopped process by setting its state to TASK_RUNNING and insert into the runqueue of the local CPU. Its parameters:
  - Process descriptor
  - Mask of process state that can be awakened
  - Flag (sync) to forbid the awakened process to preempt the currently running process

try_to_wake_up() operations

- task_rq_lock() to disable local interrupts and acquire runqueue rq lock
- Check if the state of p->state belongs to the mask of state
- If multiprocessor, checks if the awaken process can be migrated to other idle processor for load balancing
- If in TASK_UNINTERRUPTIBLE, decreases the nr_uninteruptible and p->activated to -1
- Invokes activate_task():
  - Get the current timestamp
  - Call recalc_task_prio(p, timestamp);
  - enqueue_task(p,rq->active);
  - rq->nr_running++;

try_to_wake_up() operations

6. If the target is not the local CPU, or sync not set, checks if the new runnable process has a dynamic priority higher than the current process (p->prio < rq->curr->prio);
   - If so, preempt the rq->curr by resched_task()
7. Set p->state to TASK_RUNNING
8. task_rq_unlock()
9. Return 1 if the process is awakened successfully, 0 if not awakened

The recalc_task_prio() function

- Updates the average sleep time and the dynamic priority of a process
- Receives a process descriptor *p* and a timestamp *now*
- Operations
  1. sleep_time = min(now - p->timestamp, 10^9);
  2. p->timestamp is the timestamp of the time when the process was previously put to sleep (in nano seconds)
  3. Use CURRENT_BONUS() to compute the bonus of the process
  4. Add sleep_time to the average sleep of the process (p->sleep_avg)
  5. p->prio = effective_prio()

The schedule() function

- The CPU scheduler
- Find a process in the runqueue list and assign the CPU to it
- Invoked directly or in a lazy (deferred) way
  - Direct invocation
    - The current process is about to be blocked on a resource (e.g., fail to lock a mutex)
  - Lazy invocation
    - See later slices

Direct Invocation

1. Insert current in the proper queue
2. Change the state to TASK_INTERRUPIBLE or TASK_UNINTERRUPTIBLE
3. Invokes schedule()
4. Checks if the resource is available; if not, got to step 2
5. Once resource available, removes current from the wait queue

Lazy Invocation

- Invoked in a lazy way by setting TIF_NEED_RESCHED flag of current to 1
  - This flag is checked before resuming a User Mode process (returning from interrupts and exception), so schedule() will definitely be called in the near future
- When lazy invocation is used?
  - Current used up its quantum of CPU, this is done by the schedule_tick()
  - A process is woken up and its priority is higher than that of the current process; this task is performed by try_to_wake_up()
  - A sched_setscheduler() system call is used

Actions performed by schedule()

- Before a process switch
  - Code started with *need_resched:*
- Make the process switch
  - Code started with *switch_tasks:*
- After a process switch
  - barrier();
  - Code started with *finish_schedule:*

need_resched:

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

......

now = sched_clock( );
run_time = now - prev- >timestamp; if (run_time > 1000000000)
 run_time = 1000000000;
run_time /= (CURRENT_BONUS(prev) ? : 1);

 ......

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);
 }
}
```

need_resched:

```
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;
}
.

  ......

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


......


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;
```

next->activated:
1 or 2 indicate that the process next is INTERRUPTIBLE and is woken up by system call, a kernel thread, ISR, or a deferred function

Therefore *next* is Likely to be interactive process

To make the process switch

- After determine the next process to run, kernel will access the thread_info data structure of next (top of the next's process descriptor)
- switch_tasks: prefetch(next);
  - Load the first field of next's PD into cache, in parallel with schedule()
- clear_tsk_need_resched(prev);
- Decrease the average sleep time of prev,
  - prev->sleep_avg -= run_time;
  - prev->timestamp = prev->last_ran = now;

More operations of process switch

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

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

context_switch()

- Adjust active_mm if kernel threads borrow regular processes’ mm structure
- switch_to() for process switch

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

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

}

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

After a process switch

- Cleaning up. Check if TIF_NEED_RESCHED of the current process (now prev) is set
  - goto need_resched if set

```
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;
```

Runqueue Balancing in Multiprocessor Systems

Multiprocessor Architectures

- Symmetric multiprocessing (SMP)
  - A common set of RAM chips shared by identical CPUs
- Hyper-threading
  - A hyper-threaded CPU provides multiple identical register sets
  - To exploit the parallelism among execution units by interleaving execution cycles and memory cycles
  - A hyper-threaded CPU is seen by Linux as several different logical CPUs

Multiprocessor Architectures

- Non-Uniform Memory Access (NUMA)
  - A CPU has its local memory
  - Memory access latencies vary a lot, fast on local memory and slow on remote memory
  - Better scalability
  - More complicated programming models

- Combinations of the architectures are widely used
  - Two physical CPUs, each of which has two logical processors

Scheduling Domains

- A scheduling domain is a set of CPUs whose workloads should be kept balanced by the kernel
- Scheduling domains are hierarchically organized
  - 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.
- If the number of processes in a CPU group is significantly larger than that of another, the processes are migrated to keep the numbers balanced

Scheduling Domains

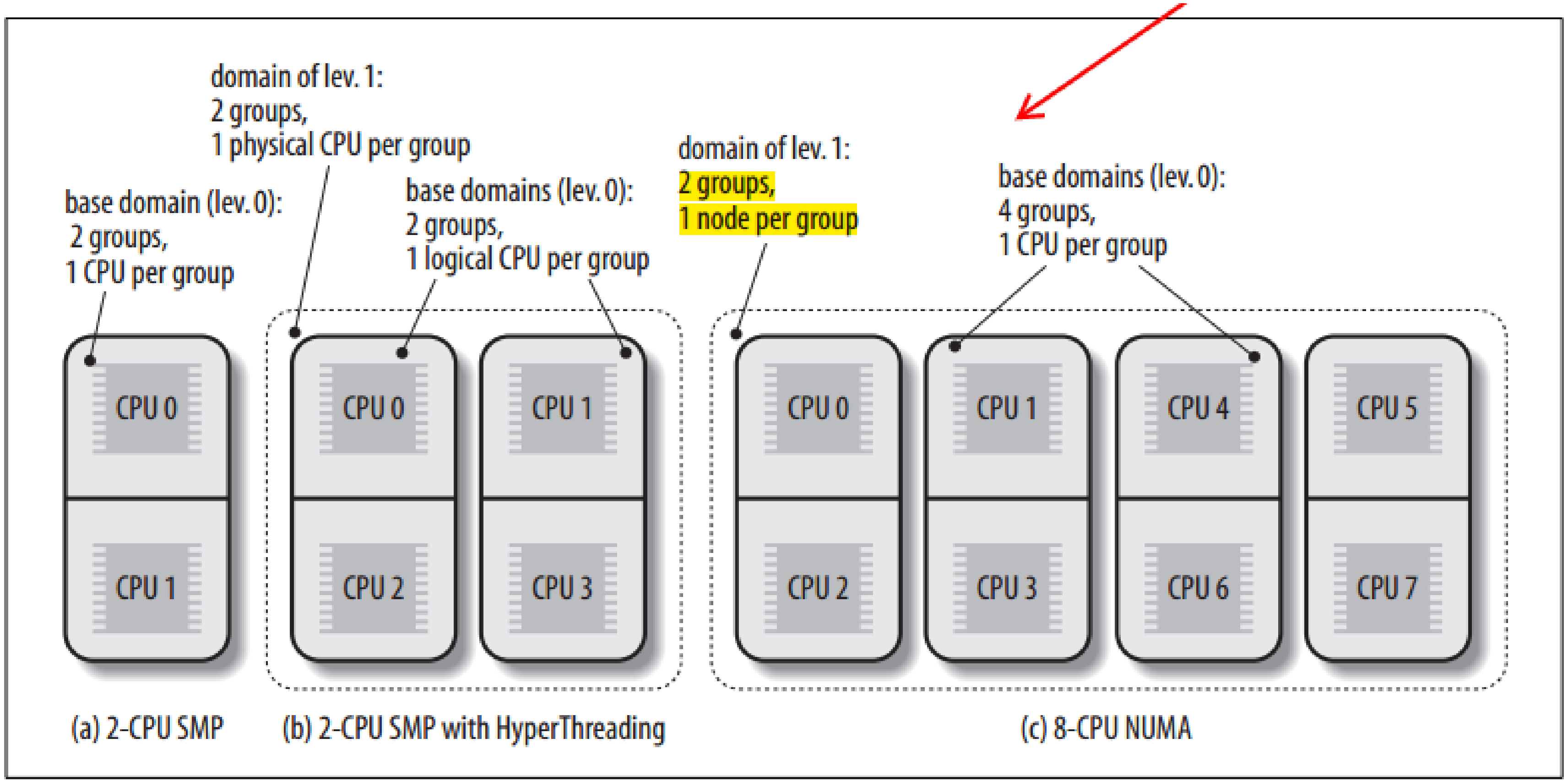


Figure 7-2. Three examples of scheduling domain hierarchies

Scheduling Domains

- (a) represents a hierarchy composed by a single scheduling domain for a 2-CPU classic multiprocessor architecture
- The scheduling domain includes only two groups, each of which includes one CPU

Scheduling Domains

- (b) represents a two-level hierarchy for a 2-CPU multiprocessor box with hyper-threading technology
- The top-level scheduling domain spans all four logical CPUs in the system, and it is composed by two groups
  - Each group of the top-level domain corresponds to a child scheduling domain and spans a physical CPU
- The bottom-level scheduling domains (also called base scheduling domains) include two groups, one for each logical CPU

Scheduling Domains

- (c) represents a two-level hierarchy for an 8-CPU NUMA architecture with two nodes and four CPUs per node
- The top-level domain is organized in two groups, each of which corresponds to a different node
- Every base scheduling domain spans the CPUs inside a single node and has four groups, each of which spans a single CPU

Scheduling Domains

- Every scheduling domain has a sched_domain descriptor
- Every group inside a scheduling domain has a sched_group descriptor
- sched_domain->groups points to the first element in a list of group descriptors
- sched_domain->parent points to its parent scheduling domain

The rebalance_tick( ) Function

- rebalance_tick( ) is invoked by scheduler_tick( ) once every tick.
  - Looping over all scheduling domains from the base domain to the top-level domain
  - Calling load_balance( ) for load balancing on the scheduling domain

The load_balance( ) Function

- The load_balance( ) function checks whether a scheduling domain is significantly unbalanced
- Try to move some processes from the busies group to the local runqueue

The load_balance( ) Function

- It receives four parameters:
- 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( ) Steps

1. Acquires the this_rq->lock spin lock
2. Invokes the find_busiest_group( ) function, it returns
   - the address of the sched_group descriptor of the busiest group and
   - the number of processes to be moved into the local runqueue
   - NULL if no load balancing is necessary

The load_balance( ) Function

3. If find_busiest_group( ) find load balancing unnecessary, it releases the this_rq->lock spin lock and delay the next invocation of load_balance( )
4. Invokes the find_busiest_queue( ) function to find the busiest CPUs in the group found in step 2 and returns the address of its runqueue
5. Acquires the spin lock busiest->lock

The load_balance( ) Function

6. Invokes the move_tasks( ) function to move some processes from the *busiest* runqueue to the local runqueue *this_rq*
7. Releases the busiest->lock and this_rq->lock spin locks
8. Terminates

The move_tasks( ) Function

- The move_tasks( ) function moves processes from a source runqueue to the local runqueue
- Scan process in the source runqueue using the following order
  - High priority to low priority
  - Expired processes and then active processes
- A process in the source runqueue can be moved if all of the following conditions are true
  - The process is not being currently executed
  - The process can execute on the local CPU (the cpus_allowed bitmask)

End of Unit 4