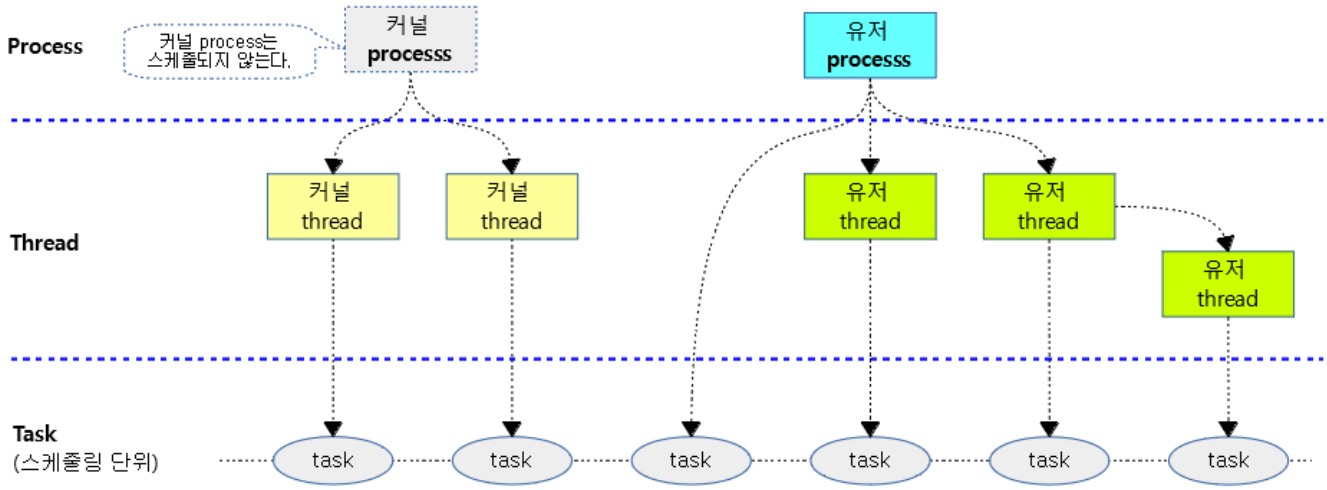
Scheduler -1- (Basic)

2017-04-24 (http://jake.dothome.co.kr/scheduler/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

What is a task?

- process
  - The general definition is that a program is in a running state.
  - When the OS kernel is booted up and ready, the user level loads the application from disk and retrieves resources (memory, files, paging, signals, stack, ...) is the minimum unit that can be operated by assigning
  - It is requested at the user level and prepared and generated by the OS kernel.
  - In particular, the OS kernel itself can be said to be a kernel process.
    - Kernel processes are created at bootup time, do not expire until just before termination, and are not included in the scheduling unit, so their existence is not obvious.
- thread
  - The general definition is that only the execution units are separated from the process.
    - For this reason, it is also called the Light-Weight Process (LWP).
    - Reference: Leveling up to mainstream OS "Linux Kernel 2.6" (http://www.zdnet.co.kr/news/news_view.asp?artice_id=00000039129093&type=det&re=) | ZDnet Korea
  - In the context of architecture, a thread is a hardware thread, or a virtual core, which is the smallest unit of device that can execute instructions.
    - e.g. a 2-core hardware thread with 86 hyper threads per core (virtual core in x4) has a total of 8 hardware threads.
  - Threads at the OS level are smaller pieces of software threads that are created inside a process.
    - In kernel code, depending on where it is used, threads can mean hardware threads or software threads, so it is necessary to distinguish them appropriately.
  - In Linux version 2.6, the kernel manages and creates threads entirely. Therefore, it is explained by this standard.
    - Finally, the NPTL (Native POSIX Thread Library) developed by the Red-hat team was adopted.
    - Even if it's called a user thread, you have to be careful about what the kernel creates and manages. Prior to Linux kernel 2.6, this was a real user thread that was used by clone processes or created through user libraries.
    - See: Native POSIX Thread Library (https://en.wikipedia.org/wiki/Native_POSIX_Thread_Library) | Wikipedia

    - Processes can share and use the resources they are allocated to. However, the stack and TLS (Thread Local Storage) zones are not shared, but created separately.
    - User threads are created from the user process or from the parent threads.
    - In particular, kernel threads can only be created at the OS kernel level, not at the user level.
- task
    - The smallest unit that can schedule a process or thread in Linux is a task.
    - If you rate scheduling in a user task, use the nice value to calculate the time slice for each task.
        - In the internal data structure, the task_struct represents the task, contains a schedule entity structure inside it, and uses this schedule entity to compute the time slice.
            - Tasks have three schedule entities that can be used by the scheduler: sched_entity, sched_rt_entity, and sched_dl_entity.
        - When scheduling a group using a cgroup, the schedule entity is linked to the group scheduling and calculates the time slice.
    - Unlike Unix, in the case of Linux, if there is no configuration, the process and thread use the same task representation and the scheduling ratio is the same.



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/task-1.png)

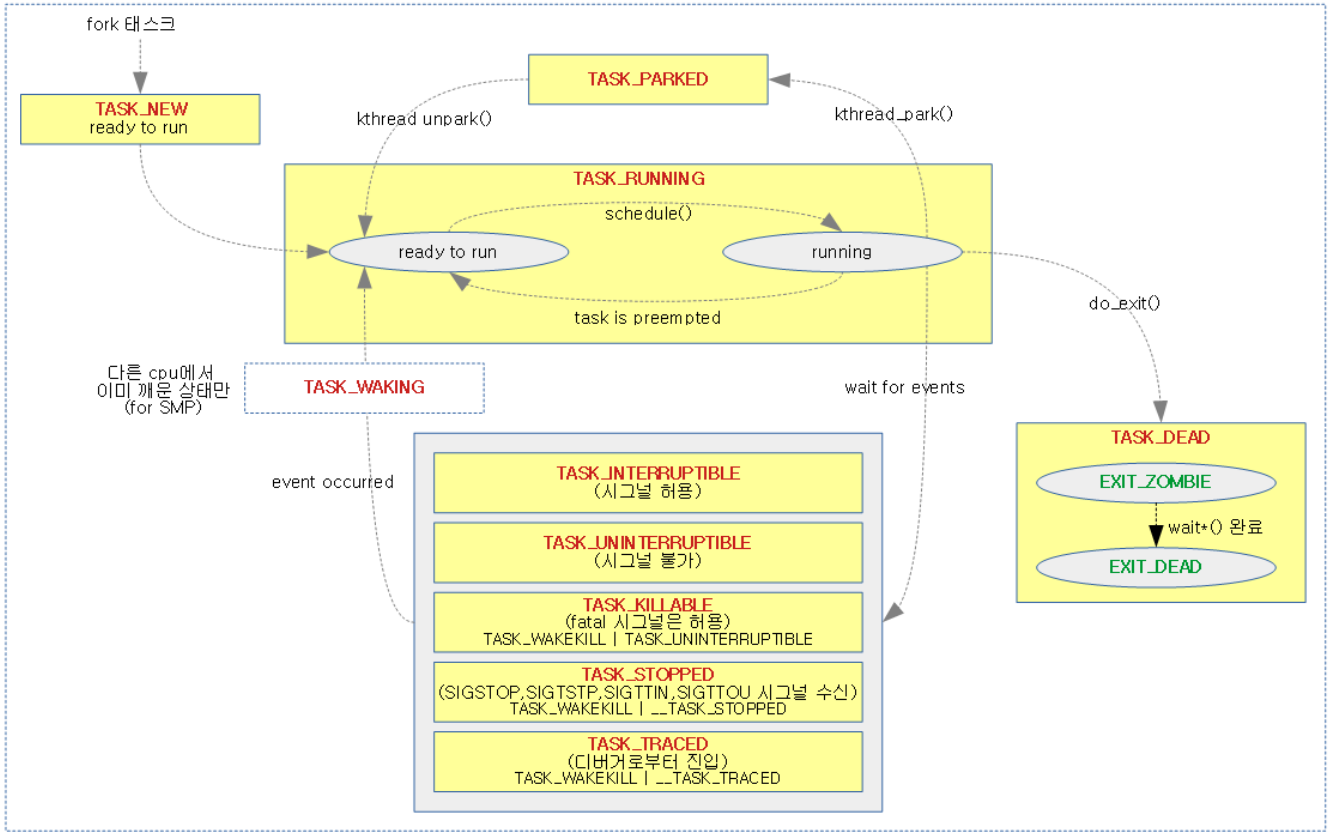
Task status

Each task has a flag to indicate its status. There are two types of flags, each of which looks like this:

- task->state
    - TASK_RUNNING
        - They are divided into two types:
            - The task is running on the CPU. (running on cpu)
            - The task is being prepared to run on RunQ. (ready to run)
    - TASK_INTERRUPTIBLE
        - The task has slipped and is ready to accept signals.

    - TASK_UNINTERRUPTIBLE
        - The task has slipped and is not interrupted by signals.
            - When used in conjunction with TASK_WAKEKILL flags, a fatal signal, SIGKILL, can be received.
        - It is used to wait for important things to be done in a slip state without interference from the signal.
            - e.g. to wait for the completion of the operation of moving the block device data to the buffer.
    - TASK_STOPPED
        - The task has been aborted at the next signal.
            - I received a request for a SIGSTOP signal from a user. This can be continued with a SIGCONT signal sent by the user.
            - I received a request for a SIGTSTP(^Z) signal from foreground.
            - background received a request for a SIGTTIN signal.
            - background received a request for a SIGTTOUT signal.
    - TASK_TRACED
        - The task is interrupted by a debugger request.
    - TASK_KILLABLE
        - TASK_WAKEKILL and TASK_UNINTERRUPTIBLE are combined.
        - It is uninterruptible, but it can only receive a SIGKILL request, which is a fatal signal.
    - TASK_WAKEKILL
        - It is a flag that can receive a SIGKILL request, which is a fatal signal, and is not used alone.
    - TASK_PARKED
        - ktrhead_create() is a special management method that allows the per-CPU kernel thread to perform sleep-state transitions.
        - You can slip with kthread_park () and break with kthread_unpark ().
- task->exit_state
    - EXIT_ZOMBIE
        - This is the first of two states that a task uses when it ends.
        - This state is maintained until just before the parent task collects information about the child task via a wait*() related function.
    - EXIT_DEAD
        - This is the last of the two states that a task uses when it ends.
        - Since the parent task has collected the information of the child task through the wait*() related function, the child task is in the process of expiating.

The following diagram shows the task state transition diagram.



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/task-state-1.png)

Multitasking (time slice, virtual runtime)

If you run two tasks on one CPU at the same time, theoretically two tasks should run in parallel at the same time with 2% of the power on that CPU, but since the real hardware CPU can only perform one task at a time, it satisfies it by alternating by the virtual runtime that you have divided the time to satisfy.

Priority

Priority

- There are a total of 0 steps from 139~140 as a priority that is classified and used by the kernel.
  - The lower the priority value, the higher the priority and the more time slices you receive.
- 0~99 100 steps are used in the RT scheduler.
- The 100 steps from 139~40 are used in the CFS scheduler.
  - This is the area that is converted into NICE.
- Unlike RT and CFS tasks, deadline tasks do not have priority and always take precedence over RT or CFS tasks.

Nice

- POSIX standard. There are a total of 20 levels ranging from -19 ~ +40 that can be specified in a general task.

    - -19, which has the smallest nice value, has a higher priority and takes a lot of time slices.
    - When converted to priority value, it has a priority of 100 ~ 139.
        - It is greater than the priority 0 ~ 99 value used by the RT scheduler, so it always has a lower priority than the RT scheduler.
    - When a process or thread is created with a user who is not root, the NICE value of the task, which is the unit of scheduling, starts with 0.
- In 2003, the +19 level time slice was set to be the same as 1 ms (1 jiffy), but then it was changed to 1000 ms because the time slice was too low on a system (desktop) with HZ=5.

Scheduler

There are five schedulers available in the kernel by default, and the user task is to select a schedule policy, each of which can be selected. However, the stop scheduler and idle-task scheduler are not available to users because they are used by the kernel itself. The processing priority for each scheduler works in the order of the scheduler below. However, in the case of the RT scheduler, it gives 5% (default) of time allocation to the CFS scheduler to prevent the CFS task from being starvated by the RT task.

- Stop Scheduler
- Deadline Scheduler
- RT Scheduler
- CFS Scheduler
- Idle Task Scheduler

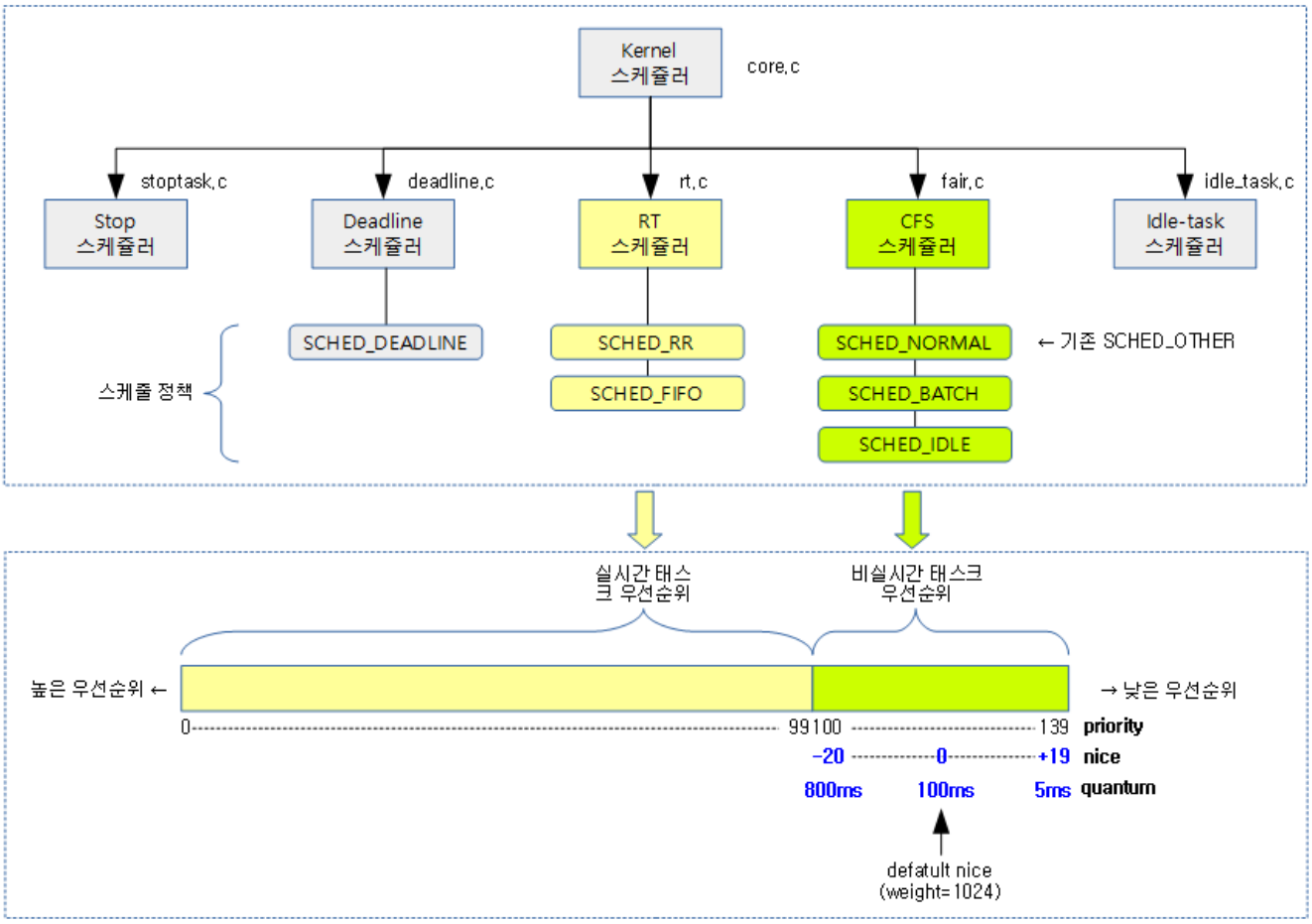
Scheduler Competition

When a task is registered with each scheduler, the first scheduler to process is Stop, and the priority is

Scheduling Policies

- SCHED_DEADLINE
    - Run the task in the deadline scheduler.
- SCHED_RR
    - Tasks of the same priority are run by the RT scheduler using the round-robin method in default 0.1 second increments.
- SCHED_FIFO
    - Make the task run in the RT scheduler in a first-in, first-out (FIFO) fashion.
- SCHED_NORMAL
    - Make the task run in the CFS scheduler.
- SCHED_BATCH
    - Make the task run in the CFS scheduler. However, unlike SCHED_NORMAL, it avoids yield so that the current task can handle it as much as possible.
- SCHED_ILDE

- Makes the task run with the lowest priority in the CFS scheduler. (It's slower than nice 19; there's no such thing as nice=20, but you can think of it like this.)
  - Note: Using this policy does not put you in the idle scheduler.



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/scheduler-1a-1.png)

CFS Scheduler

- NICE (-20 ~ +19) priority tasks assign weights according to the value of NICE to the tasks, and timeslice are assigned to the proportion of the weight of each task, so that the tasks work in a context siwthching.
- I started using Linux v2.6.23.
- Each task is given a virtual runtime and managed in the RB tree.
  - p->se.vruntime is sorted into the key.
  - The active and expired arrays used by the existing scheduler prior to CFS are no longer used by the CFS scheduler.
- It is used in three scheduling policies.
  - SCHED_NORMAL
  - SCHED_BATCH
  - SCHED_IDL
- Whereas the O(1) scheduler used just before cfs relied on jiffies and hz constants and used ms units, the cfs scheduler was designed in ns.
- It no longer uses statistical methods, but manages on a deadline basis at a given runtime.
- You can use the CFS bandwidth function.

- e.g. kworker tasks that work as workers in ksoftirqd or high priority workqueues run with nice -20 (the highest priority among nice) in the CFS scheduler.

Real-time (RT) scheduler

- Each RT task is assigned a priority (0~99, 0 is the fastest), and according to the priority, when RT tasks compete with each other, the RT task with the highest priority is processed first, and the next priority RT task is performed after processing is completed.
  - If priority 51 and 52 compete, do not switch to task context 51 while priority 52 is running. In other words, they don't compromise on RT tasks that are lower priority than you. However, it gives the CFS task a time split of 5% (1 second – sched_rt_runtime_us) by default so that the CFS scheduler can work.
- sched_rt_period_us
  - Determine the execution cycle (initial: 1000000 = 1 second)
  - If this value is a very small number, the system will not be able to respond to processing.
    - If it is smaller than hrtimer, etc.
- sched_rt_runtime_us
  - Determine the execution time (initial: 950000 = 0.95 seconds)
  - Assuming 100% of the total schedule time, 95% of the time will be available to the RT scheduler.
- You can choose from two scheduling policies that can determine whether to switch the task context for RT tasks with the same priority.
  - SCHED_FIFO
    - There is no concept of weight value and time slice used by the CFS scheduler. The first RT task will continue to run until it sleeps on its own or the task ends.
  - SCHED_RR
    - If a task with the same priority exists, it operates in a round-robin pattern (task context switching) in 0.1 seconds (default) increments.
- It is managed in an array of 0 priority lists from 99 to 100.
- RT bandwidth function is available.
- e.g. migration, threaded irq, and watchdog tasks run in the RT scheduler.

Deadline Scheduler

It is the most prioritized scheduler that can be specified by the user. It is prepared for audio, video frame playback and PWM drivers that need to run regularly, but they are not being used well yet.

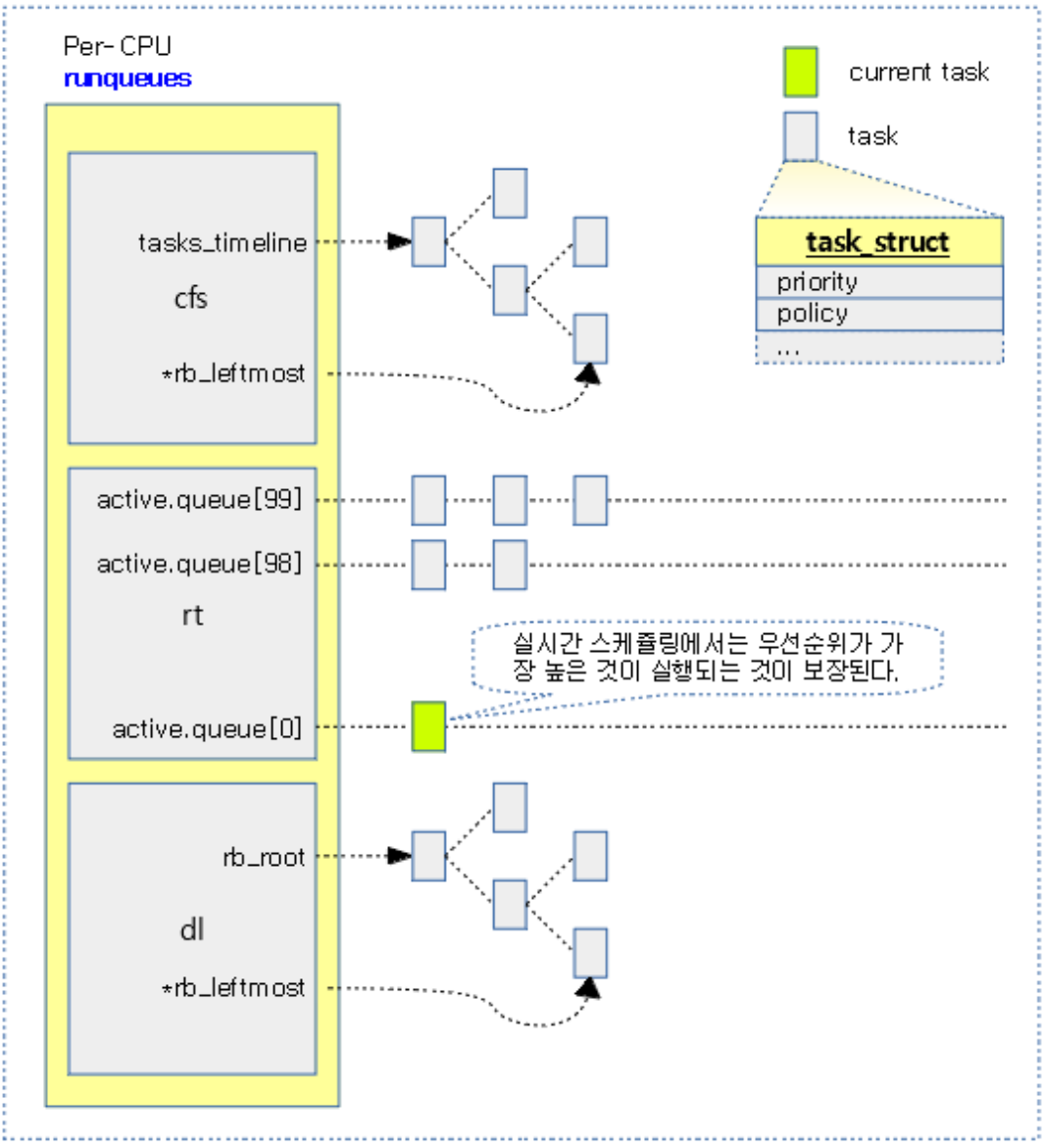
- It is used in three scheduling policies.
  - SCHED_DEADLINE
- The deadline is managed in the RB tree.

RunQ

- One is used for each CPU.
- Each scheduler works on RunQ.
  - There are CFS RunQ, RT RunQ, and Deadline RunQ at the bottom of RunQ.
- Tasks to be assigned to the CPU are enqueued to the runqueue in the form of schedule entities.
  - Tasks other than the current task are enqueued.
  - nr_current = current tasks + number of enqueued tasks.
- When a task is first executed, it is enqueued to the runqueue where the parent task is located, if possible.
  - If they are assigned to the same CPU, they have a higher cache affinity, which improves performance.

The following figure shows the main data structure used by each scheduler in RunQ to manage tasks.

- The stop scheduler and idle scheduler are used internally by the kernel and are not included in the figure below.



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/scheduler-2.png)

Kernels and Floating Points

The Linux kernel uses the following techniques to improve the performance of real computations when routines that require performance are repeated every time.

- To use int operations instead of float operations, real numbers are converted to binary integers.
- Complex calculations are made into pre-calculated tables.
  - The table range should be limited to a certain extent for the pre-calculated table.
  - e.g. If the input n value is limited to 0~3, the table reflects any factor
    - x[0] = 1024
    - x[1] = 820
    - x[2] = 655
- Use the Shift command after multiplication to replace the slow int division operation.
- Use the inline function to save time spent passing arguments. Instead, if there are many places where this function is called, the code size will increase proportionately.

Floating Point Operations in the Linux Kernel

Here's how to handle the use of floating points in Linux kernel code:

- When the compiler builds such kernel code, it ensures that it does not create assembly code that is used by the coprocess for real number operations.
  - The compiler uses the real number library provided by the compiler to handle mistakes.
  - This real number library consists only of general-purpose registers and commands that use only integers.
  - In addition, the kernel includes a real code that allows the kernel to emulate an instruction fault if a user task performs a mistake code and the instruction cannot be executed.
  - The only exception to this is that drivers such as FPU and GPU can use the Floating Point command directly.
- In the early days, most computer systems did not have co-processor units for real computations, or they were used as empty sockets.
- Even when coprocessors were installed, the performance varied greatly from system to system. And the real number instruction in simple multiplication and division has always been slower than the processing of integer commands, so it has always been and is the same now. Therefore, the Linux kernel initially forced the use of code that excluded coprocesses for floating point operations in the core design.

Binary Essence

When using real numbers as integers, integers are created and used according to accuracy in order to increase the accuracy of decimal points or less.

- e.g. 10 evolutionary integer
  - In order to treat real number 10 in decimal to a precision of -> 1.1, multiply real number 000 by 1 (1000^10).
- e.g. binarized integers

- In binary numbers, real number 2 is treated up to a precision of -> 1.1, which is multiplied by 000b1(0^10000000000).

Mult & Shift

Learn how to use the mult and shift commands to replace the slow int division command.

- formula
  - X/Y ——(alternate)—–> X*mult >> shift
- e.g. if you want to divide X by Y(820)
  - Remember the value 820/Y(1/1) inverse Y(820) as the binary integer value.
  - Once the precision to use (shift=32) is determined, the binarized integer value (mult) is determined as follows:
    - mult = (2^32) * 1/820 = (2^32) / 820 = 5,237,765
  - After all, X * mult(5,237,765) >> shift(32) can replace the division operation.
- This same technique is used in many routines in the kernel.
  - Example: Timer -10- (Timekeeping) (http://jake.dothome.co.kr/timekeeping) | Qc

---

CFS Runtime(Time Slice) & Virtual Runtime

Calculating CFS Task Runtime

Weight by NICE

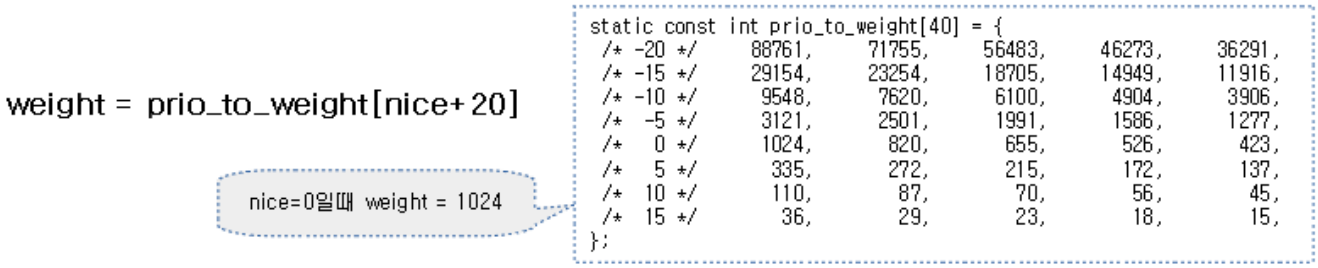
To calculate the time slice, each CFS task is given a weight value based on the NICE priority value assigned to each CFS task.

- The weight value for a CFS task with NICE at 0 is set to 1.0.
- The implementation code does not use the 1.0 floating point operation, but instead uses binary integer values.
  - The shift value corresponding to precision as a binary integer value on a 32-bit system is 10 (1024^2) using 10.
  - On a 64-bit system, the above value is multiplied by scale(1024) for greater precision, resulting in 1,048,576(2^20).

First, let's take a look at the pre-calculated weight table according to NICE priorities.

- The task with the medium priority NICE 0 has a weight of 1024, and the slowest task with a value of NICE 19 has a weight of 15.
- For the idle tasks that are not listed in the table below, but are used by the idle scheduler, use 3 that receives the least amount of time slices. You can see that the weight of the slowest CFS task is slower than 15.
- It is designed so that every priority 1 change in the NICE value changes a 25% weight, and the reason for this is explained below.

kernel/sched/core.c – sched_prio_to_weight[]



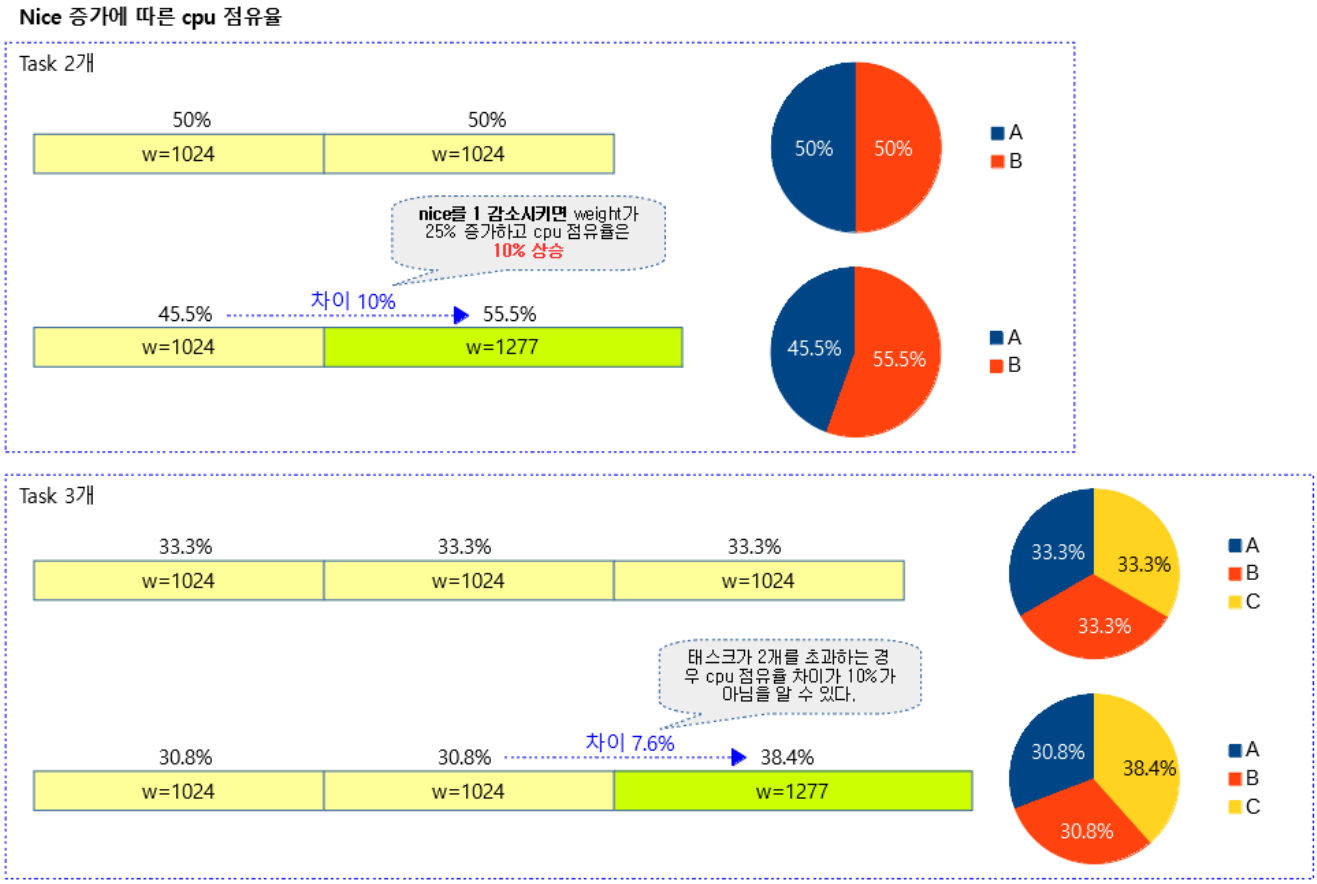
(http://jake.dothome.co.kr/wp-content/uploads/2017/04/prio_to_weight-1.png)

Amount of weight change

For every 2 decrease in the NICE value when there are 1 tasks in operation, the weight value is increased by 10% to receive 25% more Time Slice assignments. However, when comparing three or more tasks, the time slice allocation is increased by 3.10% instead of 7%.

As shown in the figure below, when the weight is increased by 25%, you can see the actual increase in CPU occupancy compared to before.

- 2% increase for 10 baselines and 3.7% increase for 6 criteria

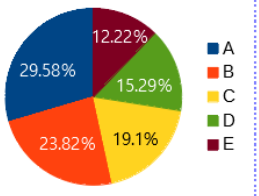


(http://jake.dothome.co.kr/wp-content/uploads/2017/04/nice-to-weight-2a.png)

The following figure calculates the weight value and its ratio according to each NICE value for each of the five tasks.

nice → weight 산정

	Task A (nice=-2)	Task B (nice=-1)	Task C (nice=0)	Task D (nice=1)	Task E (nice=2)
weight	1586	1277	1024	820	655
ratio	29.58%	23.82%	19.10%	15.29%	12.22%



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/nice-to-weight-1a.png)

CFS Task inverse weight

The following table shows a pre-computed inverse weighted mult value using the NICE value. It was created using 32-bit accuracy.

- For the purpose of improving performance, the kernel avoids division when it should use an arithmetic formula such as x / weight = y, and calculates the value of y as x * wmult >> 32 using the table values below.

$$wmult = \frac{2^{32}}{weight}$$

weight = 1024일때
wmult=4194304

```
static const u32 prio_to_wmult[40] = {
 /* -20 */     48388,     59856,     76040,     92818,    118348,
 /* -15 */    147320,    184698,    229616,    287308,    360437,
 /* -10 */    449829,    563644,    704093,    875809,   1099582,
 /*  -5 */   1376151,   1717300,   2157191,   2708050,   3363326,
 /*   0 */   4194304,   5237765,   6557202,   8165337,  10153587,
 /*   5 */  12820798,  15790321,  19976592,  24970740,  31350126,
 /*  10 */  39045157,  49367440,  61356676,  76695844,  95443717,
 /*  15 */ 119304647, 148102320, 186737708, 238609294, 286331153,
};
```

(http://jake.dothome.co.kr/wp-content/uploads/2017/04/prio_to_wmult-1.png)

CFS RunQ

Scheduling latency and scale policies

Tasks running in RunQ are repeated over a period of time to make them appear to be running simultaneously in real time. At this time, the relevant variables for calculating the periods (ns) are as follows. For a more detailed explanation, let's start with the scheduling latency calculation formula.

- rq->nr_running
  - Number of tasks running in RunQ
  - Current task + number of tasks enqueued in runqueue
- factor
  - This is the rate according to the scaling policy. This policy determines the ratio factor value based on the number of CPUs. (rpi2: factor=3)
  - Depending on the number of CPUs, periods affect the output. Specify whether you want the CPU to be proportional to the number of CPUs as it increases, or if you want to use 2log() to gradually reduce the percentage, or regardless of the number of CPUs.
  - Three policies are used, and in a separate section below,
- sched_latency_ns
  - Minimum schedule periods default (6000000) * factor(3) = 18000000 (ns)
  - "/proc/sys/kernel/sched_latency_ns"

- sched_min_granularity
  - Minimum schedule duration for the task, default (750000) * factor(3) = 2250000(ns)
  - Because the task context switching is frequent and inefficient, the time slice to be assigned to the task is not assigned less than this value.
  - "/proc/sys/kernel/sched_min_granularity_ns"
- sched_wakeup_granularity_ns
  - Default value (1000000) * factor(3) = 3000000(ns) as the wakeup period of the task.
  - "/proc/sys/kernel/sched_wakeup_granularity_ns"
- sched_nr_latency
  - Number of tasks for which sched_latency_ns is available.
  - This value is automatically calculated as sched_latency_ns/sched_min_granularity.

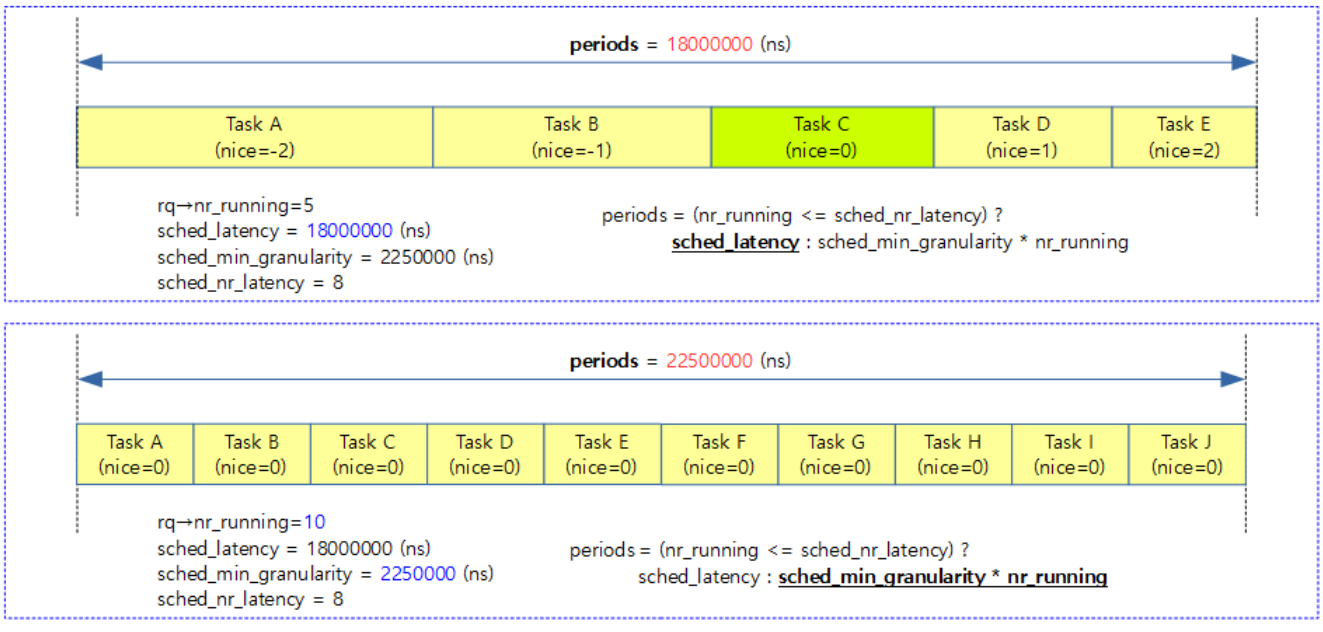
Scheduling latency calculation formula

The calculation is determined by comparing the nr_running and sched_nr_latency in one of two ways:

- sysctl_sched_latency (18 ms) <- Condition: nr_running <= sched_nr_latency(8)
- sysctl_sched_min_granularity (2.25 ms) * nr_running <- Condition: nr_running > sched_nr_latency(8)

The following figure shows how an RPI2 (CPU:4) system calculates the scheduling duration of 5 and 10 tasks on a single CPU, respectively.

- If there are less than a certain number of tasks, the calculated scheduling latency is used as periods.
- If the number of tasks exceeds a certain number, the minimum schedule duration (sched_min_granularity) for the task is not guaranteed when placing the tasks in the calculated scheduling latency, so the minimum schedule duration (sched_min_granularity) * the number of tasks is used to calculate the periods.



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/sched_latency-1a.png)

Scaling policies based on the number of CPUs

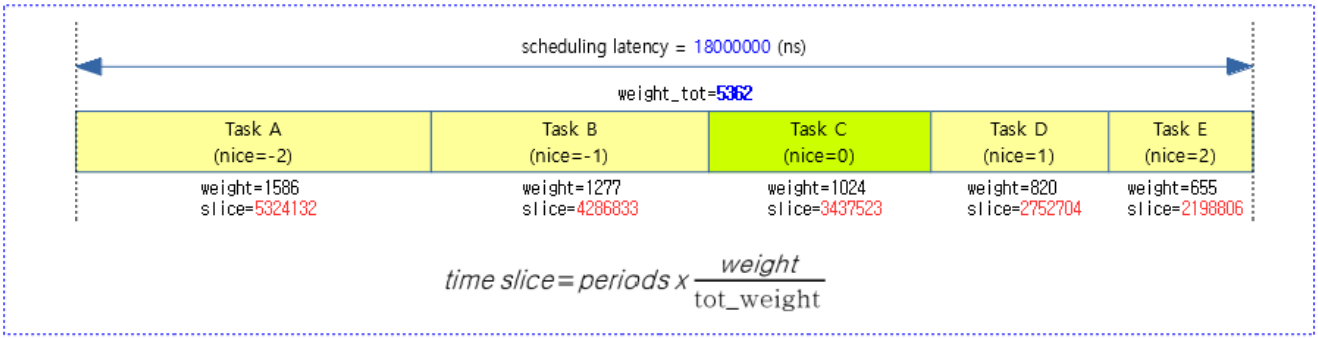
It is a ratio according to the scaling policy, and it changes the scale latency according to the number of CPUs.

- SCHED_TUNABLESCALING_NONE(0)
  - Always use 1 so that the scale doesn't change.
- SCHED_TUNABLESCALING_LOG(1)
  - 1 + ilog2 (CPU) depending on the number of CPUs. (default)
  - rpi2: CPU=4 and using this option, factor=3 is applied, which uses 3x the scale.
- SCHED_TUNABLESCALING_LINEAR(2)
  - Online CPU sorted by 8 units. (8, 16, 24, 32, ...)

Task-specific runtime (Time Slice)

Calculate the runtime time assigned to an individual task.

- Divide the duration of the 1st cycle of scheduling latency for each runqueue by the weight ratio of each task.



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/time_slice-1a.png)

VRuntime by task

A task's vruntime (virtual runtime) is the cumulative value of individual task execution times and the inverse proportion of the load weight to place it in the runqueue's RB tree. (The higher the priority, the smaller the cumulative weight value and adds it to the vruntime, so that it can be prioritized in the RB tree first.)

- The vruntime value to be incremented is the same as the runtime value given for the nice 0 task.
  - For example, if a NICE 6 task is running in a 0ms cycle, the runtime value given to the task is 6ms, and the vruntime value is 6ms.
  - For example, if there are two NICE 6 tasks running in a 0 ms cycle, the runtime value given to each task is 6 ms, which is half of the 3 ms cycle, and the vruntime value is 3 ms.
- If you interrupt the execution of the current task and do a task context switch, move the current task back by vruntime. In other words, the next task to be performed in the RB Tree of the CFS

Runqueue is selected as the one with the fastest vruntime for each task. In other words, the next task is used as a criterion for selection.

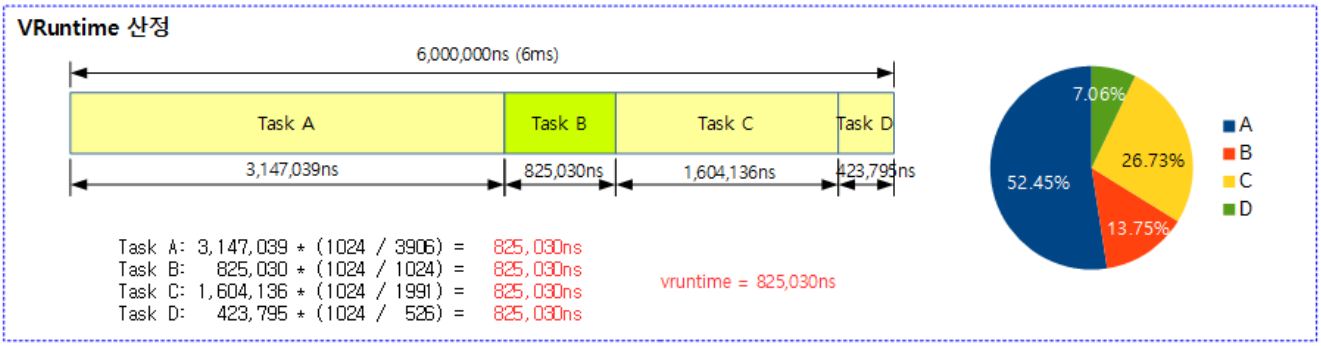
- The same vruntime is calculated for all tasks, and when the runtime of the current task is exhausted, the vruntime value for the task is updated as follows:
  - curr->vruntime += computed vruntime
  - The value of the task's vruntime is moved back by the calculated vruntime.

The virtual runtime slices to be added at the time of cumulation are calculated as follows:

vruntime slice = time slice * wieght-0 / weight

The following figure shows the results of obtaining the vruntimes for four tasks with different nice values.

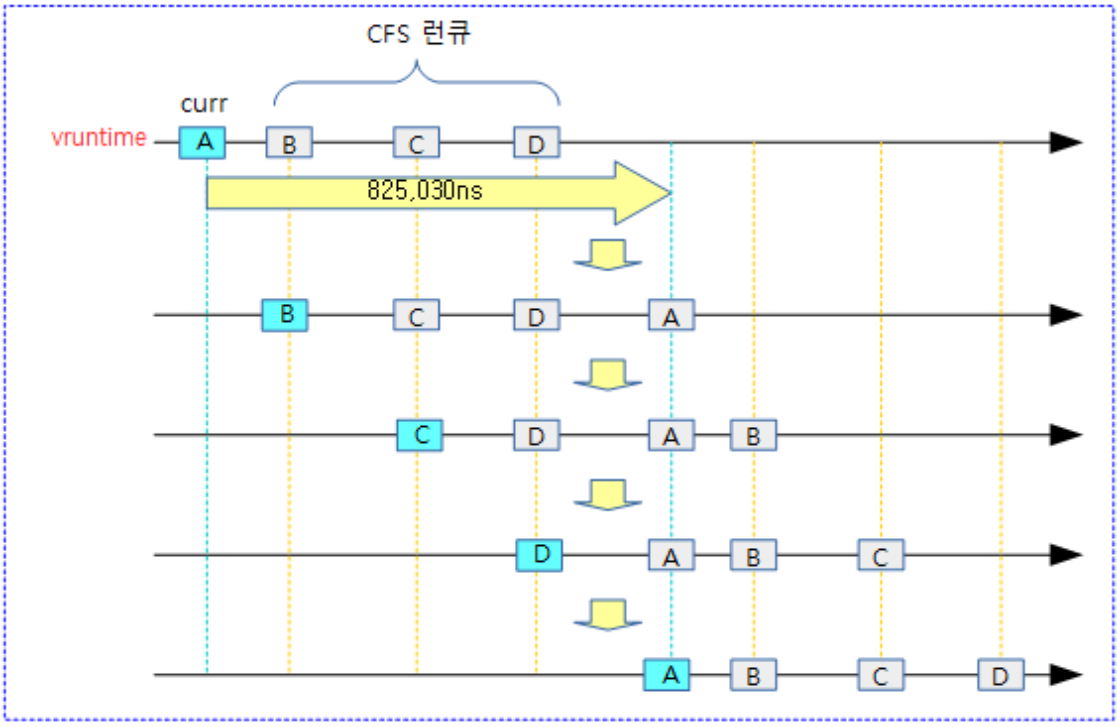
- In the case of Task B, which has a nice value of 0, you can see that the runtime value and the vruntime value are the same.



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/vruntime-2.png)

The following figure shows that if four tasks are continuously executed without idle, the vruntime of each task will continue to run cumulatively by the calculated vruntime slice.

- Note: This illustration is for the purpose of understanding a situation where HRTICK(default=no) is in action and each task is given the same opportunity, assuming that there are no preemption-requesting RT tasks. In the case of Task D, which does not work as shown in the figure below and has a low runtime, it does not have the same opportunity as Task A, which has a large runtime, because it runs more time than its own runtime in the regular schedule tick.



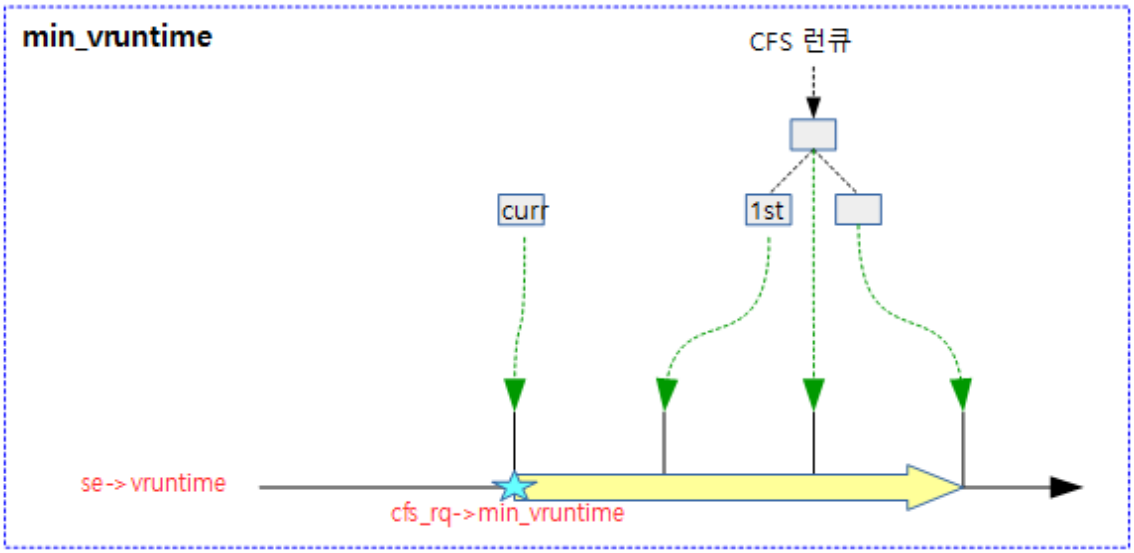
(http://jake.dothome.co.kr/wp-content/uploads/2017/04/vruntime-3.png)

cfs_rq->min_vruntime

If a new task is put on a CFS runqueue or is awakened and enqueued, and the vruntime starts at 0, the vruntime of these tasks will be the lowest, and execution will be concentrated on this task, and the other tasks will not be able to get CPU time. Therefore, in order to prevent this from happening, we have made it possible to use the min vruntime value as the standard for new entrants to the cfs runqueue.

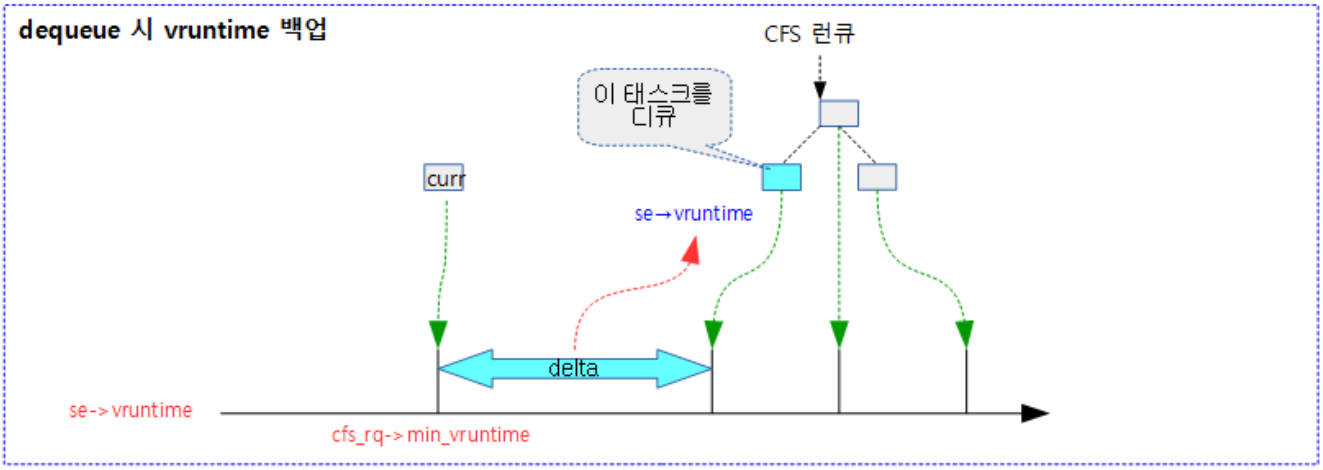
- The cfs_rq->min_vruntime value is used by the current task and the lowest number of tasks queued in the CFS runqueue, which are updated on a schedule tick and as needed.
- When entering the CFS runqueue, the min_vruntime value is added and used, and when exiting the CFS runqueue, the min_vruntime value is decremented.
    - In order for a task to slip or move to a runqueue on another CPU, it is dequeued from this CPU's runqueue, and it stores only the delta value of the CPU's runqueue minus min_vruntime.
    - When a task wakes up or is enqueued to that CPU's runqueue from another CPU, the current runqueue's min_vruntime value is added to the task's archived delta vruntime value.
- When entering the cfs runqueue, use a little rule about the vruntime.
    - A new task enters the CFS runqueue
        - min_vruntime value + vruntime slice added
        - If you used the option that the forked child task runs first, swap it with vruntime so that the current task takes precedence.
    - The idle task wakes up and enters the CFS runqueue
        - Using min_vruntime Values
    - Entering the CFS runqueue for various other reasons (moves within the task group, CPU migration, scheduler changes)

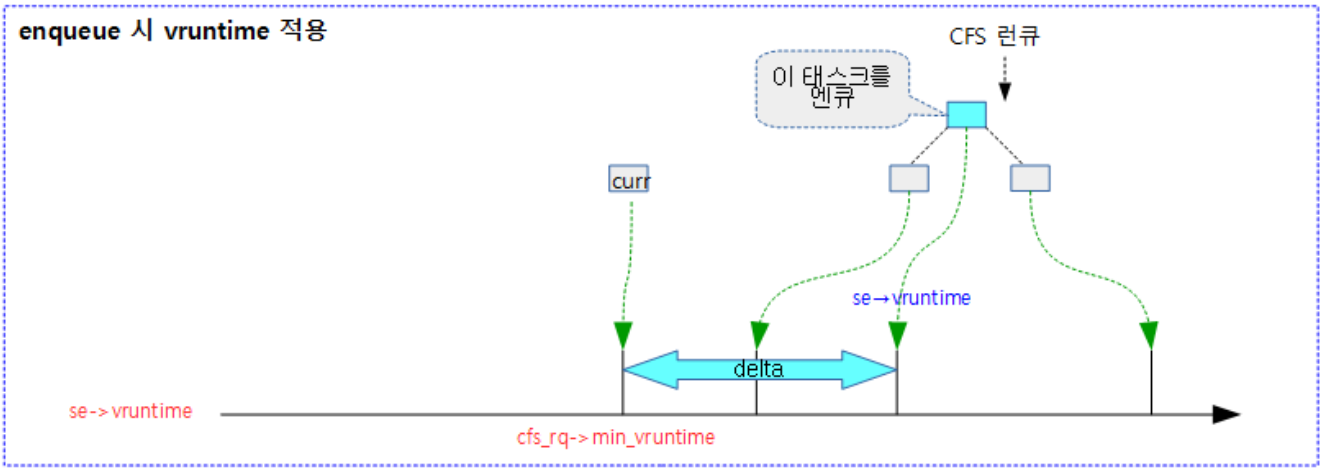
The following figure shows the min_vruntime being updated with the vruntime value used by the curr task with the lowest vruntime.



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/vruntime-4.png)

The vruntime value of the task shows that when you dequeue and enqueue in a runqueue, it is based on the min_vruntime of that runqueue.





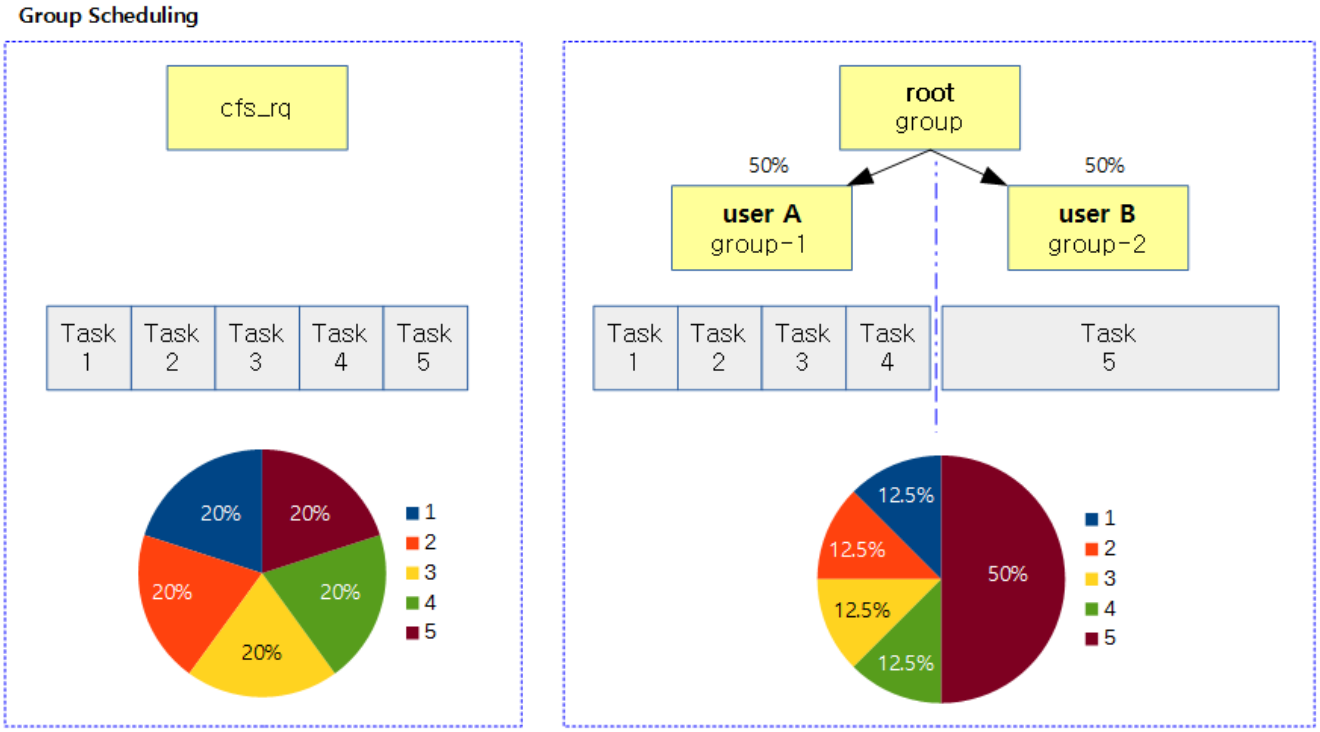
(http://jake.dothome.co.kr/wp-content/uploads/2017/04/vruntime-5b.png)

CFS Group Scheduling

- It was introduced in kernel v2.6.24 using the cgroup's CPU subsystem.
- Hierarchical Task Groups (TGs) can be set up to regulate the utilization of user processes.
    - The schedule entity is used to manage the load weight value of the task, which is divided into a scafline entity for the task and a schedule entity (tg->se[]) for the task group.
- The CFS bandwidth function of the schedule group allows you to adjust the utilization within the allotted time range.

The following figure compares the presence and absence of group scheduling.

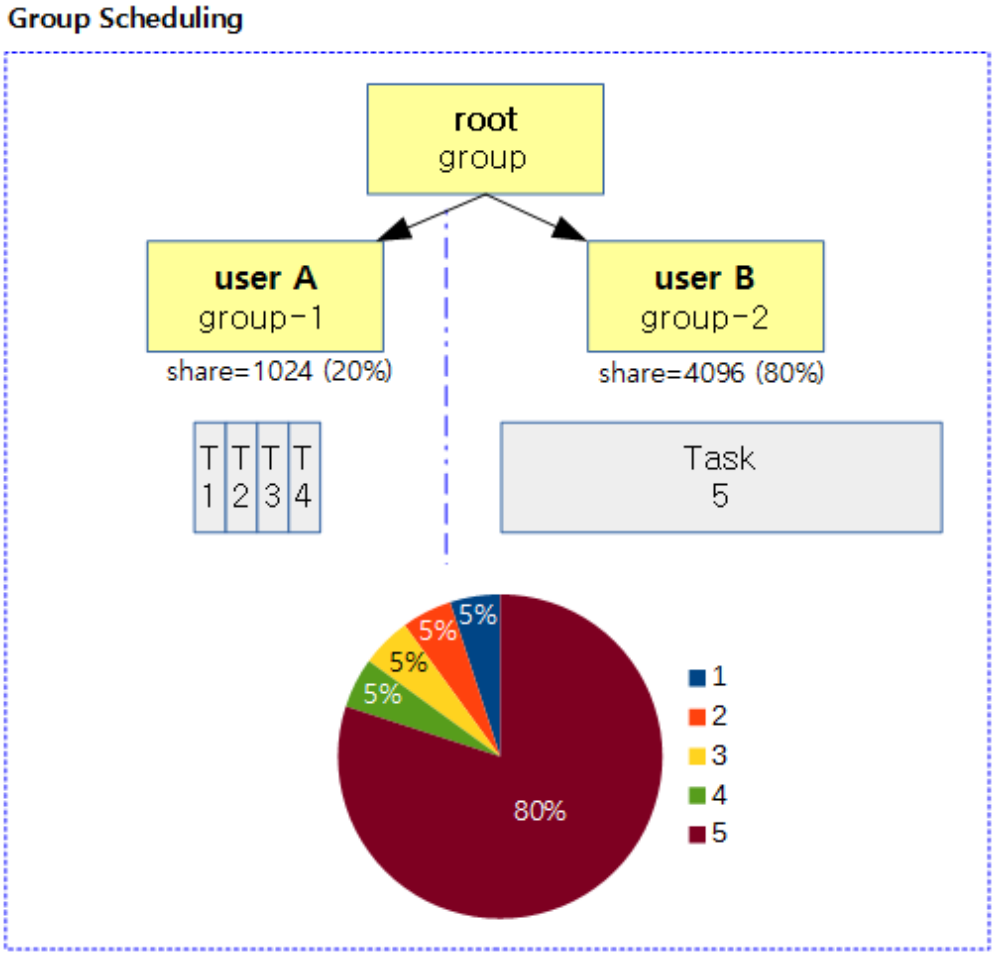
- On the left, if the tasks have the same priority as designed by the CFS scheduler, each task is assigned a runtime fairly.
- On the right, if you apply a cgroup and apply the same priority to each user, each user will be assigned a runtime fairly.
- On the right, if you create a root group and two schedule groups below it, the bottom two groups will divide the utilization by 50% by default.
- When using task groups, each task group has a CFS runqueue equal to the number of CPUs, and the tasks or task groups that belong to that group are considered to be their own schedule entities.
    - On the right, assuming that all five task schedule entities have a weight value of 5, the root group determines the distribution for the schedule entities for the two groups, and each group determines the distribution for the schedule entities for the tasks that belong to the group.
    - On the right, if two more tasks are executed in the root group, the root group performs the distribution for a total of four schedule entities. (2 schedule entities for task groups + 4 schedule entities for tasks)
        - Each task in group-1 is allocated 6.25%.
        - Tasks in group-2 are allocated 25%.
        - Task 6 and Task 7, which are not shown in the figure, are newly added to the root group, each allocated 25%.



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/group-scheduling-1a.png)

The following figure shows the change in the time slice coverage ratio when group scheduling is applied.

- If you create two task groups under the root group and set the share values corresponding to 20% and 80% respectively, you can check the utilization of the tasks under the group as follows.

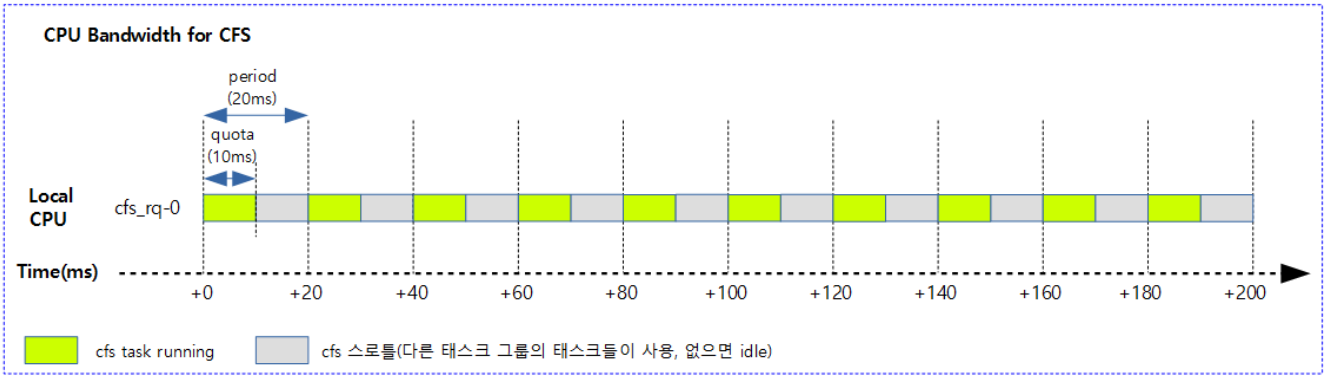


(http://jake.dothome.co.kr/wp-content/uploads/2017/04/group-scheduling-2b.png)

CPU Bandwidth for CFS

Instead of using all of the time slices assigned to the task group, make sure that only a certain percentage of them are used. This non-time-consuming section is called CFS throttling, and it gives up time slices to other task groups. If there are no tasks to concede, idle it.

The following illustration shows half of the CFS tasks in a task group set to be throttled. (Shows an example of a system using 1 core)



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/cfs-bandwidth-1c.png)

CFS Scheduling Domain

If there is a lot of CPU load on the scheduling group (sched_group), it should be loaded balanced through each scheduling domain.

- Through the hierarchical structure, we try to find the core that is appropriate for CPU affinity.

The scheduling domain refers to the device tree and MPIDR registers to change the CPU topology every time the cpu is turned on/off, and the scheduling domain is configured through this.

- Device tree + MPIDR registers -> cpu on/off -> cpu topology configuration changes -> scheduling domain

The steps of CPU topology are as follows:

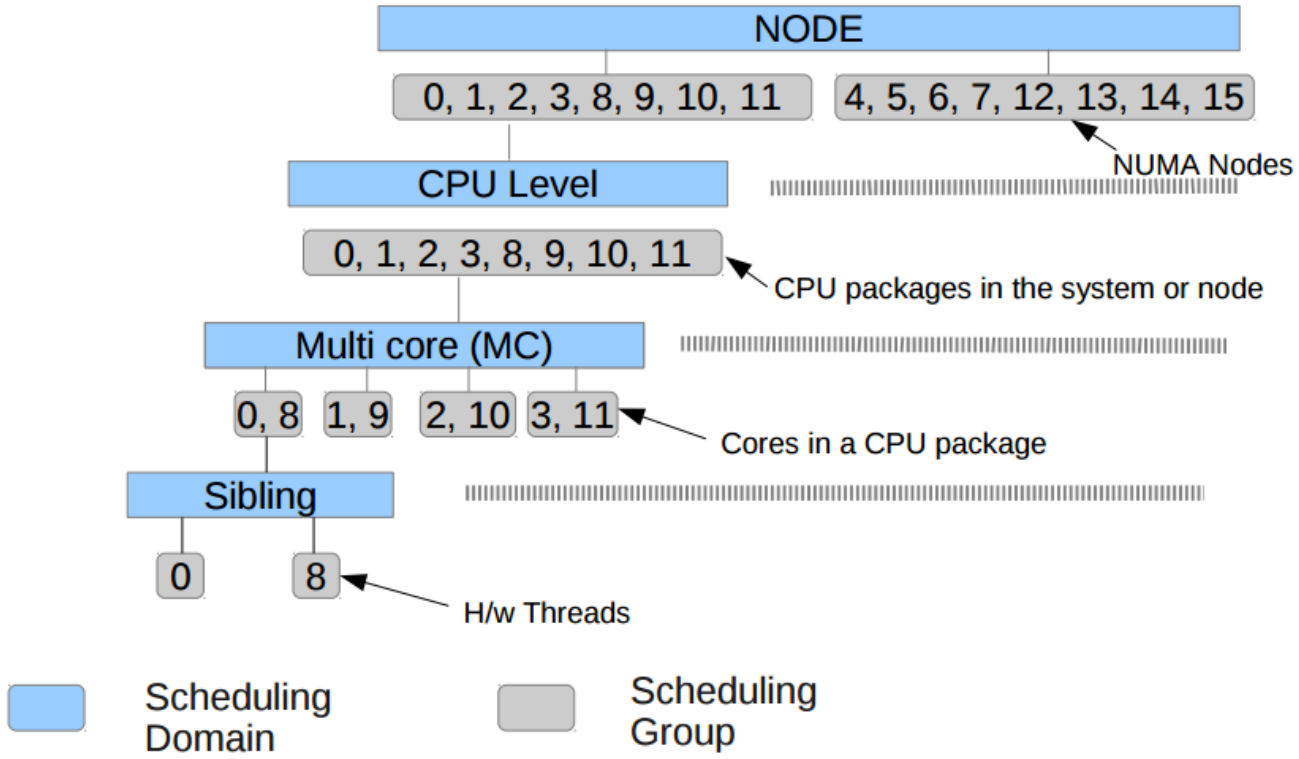
- cluster
- core
- thread

The steps for a scheduling domain are as follows:

- DIE Domains
  - cpu die (package)
- MC Domains
  - Multicore
- SMT Domains
  - Multi-threaded (H/W)

The following figure shows a scheduling domain and a scheduling group at the same time. (See ARM documentation.)

- Note that the scheduling group (sched_group) and group scheduling (task_group) in the figure below are different functions.



(http://jake.dothome.co.kr/wp-content/uploads/2017/04/scheduling-domain.png)

consultation

- Scheduler -1- (Basic) (http://jake.dothome.co.kr/scheduler/) | Qc
- Scheduler -2- (Global Cpu Load) (http://jake.dothome.co.kr/cpu-load) | Qc
- Scheduler -3- (PELT) (http://jake.dothome.co.kr/pelt) | Qc
- Scheduler -4- (Group Scheduling) (http://jake.dothome.co.kr/scheduling-group) | Qc
- Scheduler -5- (Scheduler Core) (http://jake.dothome.co.kr/scheduler-core) | Qc
- Scheduler -6- (CFS Scheduler) (http://jake.dothome.co.kr/cfs) | Qc
- Scheduler -7- (Preemption & Context Switch) (http://jake.dothome.co.kr/preemption) | Qc
- Scheduler -8- (CFS Bandwidth) (http://jake.dothome.co.kr/bandwidth) | Qc
- Scheduler -9- (RT Scheduler) (http://jake.dothome.co.kr/rt-scheduler) | Qc
- Scheduler -10- (Deadline Scheduler) (http://jake.dothome.co.kr/dl-sched) | Qc
- Scheduler -11- (Stop Scheduler) (http://jake.dothome.co.kr/stop-sched) | Qc
- Scheduler -12- (Idle Scheduler) (http://jake.dothome.co.kr/idle-sched) | Qc
- Scheduler -13- (Scheduling Domain 1) (http://jake.dothome.co.kr/sched-domain/) | Qc
- Scheduler -14- (Scheduling Domain 2) (http://jake.dothome.co.kr/sched-domain-2/) | Qc
- Scheduler -15- (Load Balance 1) (http://jake.dothome.co.kr/load-balance-1) | Qc
- Scheduler -16- (Load Balance 2) (http://jake.dothome.co.kr/load-balance-2) | Qc
- Scheduler -17- (Load Balance 3 NUMA) (http://jake.dothome.co.kr/load-balance-3) | Qc
- Scheduler -18- (Load Balance 4 EAS) (http://jake.dothome.co.kr/energy-aware-scheduling) | Qc
- scheduler -19-(initialization) (http://jake.dothome.co.kr/sched_init) | Qc
- PID Management (http://jake.dothome.co.kr/pid) | Qc
- do_fork() (http://jake.dothome.co.kr/do_fork) | Qc

- cpu_startup_entry() (http://jake.dothome.co.kr/cpu_startup_entry/) | Qc
- RunQ Load Average (cpu_load[]) – v4.0 (http://jake.dothome.co.kr/cpu_load/) | Qc
- PELT(Per-Entity Load Tracking) – v4.0 (http://jake.dothome.co.kr/pelt-v4-0/) | Qc

- CPU bandwidth control for CFS | Google Paul Turner, IBM Bharata B Rao, Google Nikhil Rao – download pdf (https://landley.net/kdocs/ols/2010/ols2010-pages-245-254.pdf)
- CFS bandwidth control (https://lwn.net/Articles/428230/) | LWN.net – Korean Translation (https://linuxkernelmsg.wordpress.com/2016/10/07/%EB%B2%88%EC%97%AD-lwn-cfs-bandwidth-control/) | KERNELMSG
- CFS Bandwidth Control (https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt) | kernel.org
- cfs scheduling 1 & cfs group scheduling (1) (http://iloveriver.egloos.com/6095529) | The kite splits the sky.
- Linux 3.2 – CFS CPU bandwidth (english version) (https://www.blaess.fr/christophe/2012/01/07/linux-3-2-cfs-cpu-bandwidth-english-version/) | Christophe Blaess
- integrating the scheduler and cpufreq | Linaro connect – Download PDF (https://s3.amazonaws.com/connect.linaro.org/bkk16/Presentations/Monday/BKK16-104.pdf)
- CFS Source Code Analysis (http://m.blog.naver.com/johnbaik1118/220430293142?navType=pr) | Childlike

- linux kernel Act 1 – CFS (Completely Fair Scheduler) Scheduler (http://egloos.zum.com/tory45/v/5167290) | Tori
- Linux kernel internals study[chapter 3] (http://egloos.zum.com/izreal/v/36669) | Ezreal
- CFS Summary (http://nzcv.egloos.com/6106818) | NZCV
- [Linux] Completely Fair Scheduler (http://studyfoss.egloos.com/5326671) | F/OSS
- CFS Scheduler (http://iloveriver.egloos.com/category/cfs%20%EC%8A%A4%EC%BC%80%EC%A5%B4%EB%9F%AC) | The kite splits the sky.
- Load balance in Linux source level bottom-up analysis (http://enginius.tistory.com/107) | Mad for Simplicity
- CPU bandwidth control for CFS | P. Turner, B. B. Rao, N. Rao – Download pdf (https://landley.net/kdocs/ols/2010/ols2010-pages-245-254.pdf)
- Evaluation of CPU cgroup | Fujitsu – Download PDF (https://events.linuxfoundation.org/images/stories/pdf/lcjp2012_izumi.pdf)
- Task Management | Baek Seungjae – Download PDF (http://embedded.dankook.ac.kr/~baeksj/course/2016_LKI/Chapter_02.pdf)

8 thoughts to “Scheduler -1- (Basic)”

JODA CITY
2019-11-04 11:25 (http://jake.dothome.co.kr/scheduler/#comment-221970)

Thank you so much for helping.

RESPONSE (/SCHEDULER/?REPLYTOCOM=221970#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2019-11-04 21:36 (http://jake.dothome.co.kr/scheduler/#comment-222011)

Thank you for your support.
Have a nice day. ^^

RESPONSE (/SCHEDULER/?REPLYTOCOM=222011#RESPOND)

FISH SEAR
2019-12-02 17:44 (http://jake.dothome.co.kr/scheduler/#comment-224660)

It's what I was looking for, but it's neatly organized. Thank you very much.

RESPONSE (/SCHEDULER/?REPLYTOCOM=224660#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2019-12-03 07:11 (http://jake.dothome.co.kr/scheduler/#comment-224716)

응원 감사합니다. 좋은 하루되세요~

응답 (/SCHEDULER/?REPLYTOCOM=224716#RESPOND)

사과시계
2023-07-30 19:55 (http://jake.dothome.co.kr/scheduler/#comment-308149)

안녕하세요, 커널 글 덕분에 많은 도움이 되고 있습니다.
한가지 질문이 있는데 task state machine에서 TASK_RUNNING을 보면, task가 preemption되는 경우에도 TASK_RUNNIGN state가 유지되는 이유가 있는건가요?
State만 생각해보면, TASK_STOPPED나 다른 state로 transition되어야할 것 같아서요.

응답 (/SCHEDULER/?REPLYTOCOM=308149#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2023-08-08 08:44 (http://jake.dothome.co.kr/scheduler/#comment-308177)

안녕하세요? 휴가를 다녀오느라 게을러져 질문이 있었는지 몰랐었습니다. ^^;

TASK_RUNNING 상태는 현재 돌고 있다는 의미로 설계가 된 것이 아니라,
실행 가능한 상태라는 의미로 설계되었습니다.
따라서 현재 돌고 있거나(curr), preemption 되어 런큐에서 대기중일 때에도
TASK_RUNNING 상태를 유지하고 있습니다.

다른 state를 하나 만들어 전이 되어도 좋을 것 같지만, 처음 설계가 위와 같아서 그대로 유지하고 있다고 판단하고 있습니다.

감사합니다.

응답 (/SCHEDULER/?REPLYTOCOM=308177#RESPOND)

사과시계
2023-08-13 14:23 (http://jake.dothome.co.kr/scheduler/#comment-308191)

헷갈렸던 부분인데 설명 감사합니다!

응답 (/SCHEDULER/?REPLYTOCOM=308191#RESPOND)

문영일 (HTTP://JAKE.DOTHOME.CO.KR)
2023-08-16 09:01 (http://jake.dothome.co.kr/scheduler/#comment-308199)

좋은 하루되세요. ^^

응답 (/SCHEDULER/?REPLYTOCOM=308199#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

❮ Scheduler -3- (PELT-1) (http://jake.dothome.co.kr/pelt/)

Scheduler -7- (Preemption & Context Switch) ❯ (http://jake.dothome.co.kr/preemption/)

문c 블로그 (2015 ~ 2023)