Λειτουργικά Συστήματα

Ο τρόπος υπολογισμού του weight στο scheduler του Linux

Μπαξεβάνος Ανδρέας

Πανεπιστήμιο Μακεδονίας

09/12/2021

Σχέση nice και weight

Έχουμε δει πως κάθε normal διεργασία έχει μία τιμή **nice**, που ανήκει στο εύρος [-20..19].

Επίσης έχουμε δει:

	RT διεργασίες	normal διεργασίες
προτεραιότητα	[0..99]	[100..139]

Οι τιμές nice, αντιστοιχούν στις τιμές priority 100..139.

Οι διεργασίες έχουν και κάποιο **weight**, το οποίο επηρεάζει το ποσοστό χρόνου που θα έχουν στη CPU.

Για να δούμε πως υπολογίζεται το weight και πως σχετίζεται με τη τιμή nice, θα δούμε κατά βήμα πως αλλάζει η τιμή nice ενός προγράμματος.

Αυτό γίνεται μέσα από τη **sys_nice**.

Η sys_nice είναι ένα syscall που επιτρέπει σε διεργασίες να αλλάξουν την προτεραιότητα τους.

(Linux Kernel 5.15.4)

/ kernel / sched / core.c

```
6945  SYSCALL_DEFINE1(nice, int, increment)
6946  {
6947          long nice, retval;
6948
6949          /*
6950           * Setpriority might change our priority at the same moment.
6951           * We don't have to worry. Conceptually one call occurs first
6952           * and we have a single winner.
6953           */
6954          increment = clamp(increment, -NICE_WIDTH, NICE_WIDTH);
6955          nice = task_nice(current) + increment;
6956
6957          nice = clamp_val(nice, MIN_NICE, MAX_NICE);
6958          if (increment < 0 && !can_nice(current, nice))
6959                  return -EPERM;
6960
6961          retval = security_task_setnice(current, nice);
6962          if (retval)
6963                  return retval;
6964
6965          set_user_nice(current, nice);
6966          return 0;
6967  }
```

/ kernel / sched / core.c

```
6945  SYSCALL_DEFINE1(nice, int, increment)
6946  {
6947          long nice, retval;
6948
6949          /*
6950           * Setpriority might change our priority at the same moment.
6951           * We don't have to worry. Conceptually one call occurs first
6952           * and we have a single winner.
6953           */
6954          increment = clamp(increment, -NICE_WIDTH, NICE_WIDTH);
6955          nice = task_nice(current) + increment;
6956
6957          nice = clamp_val(nice, MIN_NICE, MAX_NICE);
6958          if (increment < 0 && !can_nice(current, nice))
6959                  return -EPERM;
6960
6961          retval = security_task_setnice(current, nice);
6962          if (retval)
6963                  return retval;
6964
6965          set_user_nice(current, nice);
6966          return 0;
6967  }
```

/ kernel / sched / core.c

```
6945  SYSCALL_DEFINE1(nice, int, increment)
6946  {
6947          long nice, retval;
6948
6949          /*
6950           * Setpriority might change our priority at the same moment.
6951           * We don't have to worry. Conceptually one call occurs first
6952           * and we have a single winner.
6953           */                                -40          40                                  19          -20
6954          increment = clamp(increment, -NICE_WIDTH, NICE_WIDTH);//NICE_WIDTH είναι (MAX_NICE - MIN_NICE + 1)
6955          nice = task_nice(current) + increment;                                  / include / linux / sched / prio.h
6956
6957          nice = clamp_val(nice, MIN_NICE, MAX_NICE);
6958          if (increment < 0 && !can_nice(current, nice))
6959                  return -EPERM;
6960
6961          retval = security_task_setnice(current, nice);
6962          if (retval)
6963                  return retval;
6964
6965          set_user_nice(current, nice);
6966          return 0;
6967  }
```

/ include / linux / minmax.h

```
89   #define clamp(val, lo, hi) min((typeof(val))max(val, lo), hi)
     //επιστρέφει το val (increment) αφού το κάνει "clamp"
     //στο εύρος lo..hi και type checking (έλεγχος τύπων)
```

/ kernel / sched / core.c

```
6945  SYSCALL_DEFINE1(nice, int, increment)
6946  {
6947          long nice, retval;
6948
6949          /*
6950           * Setpriority might change our priority at the same moment.
6951           * We don't have to worry. Conceptually one call occurs first
6952           * and we have a single winner.
6953           */
6954          increment = clamp(increment, -NICE_WIDTH, NICE_WIDTH);
6955          nice = task_nice(current) + increment; //προσθέτει το nice της διεργασίας με το increment
6956
6957          nice = clamp_val(nice, MIN_NICE, MAX_NICE);
6958          if (increment < 0 && !can_nice(current, nice))
6959                  return -EPERM;
6960
6961          retval = security_task_setnice(current, nice);
6962          if (retval)
6963                  return retval;
6964
6965          set_user_nice(current, nice);
6966          return 0;
6967  }
```

/ include / linux / sched.h

```
1838  static inline int task_nice(const struct task_struct *p)
1839  {
1840          return PRIO_TO_NICE((p)->static_prio);
1841  }
      //μέσα από το PRIO_TO_NICE macro, μετατρέπεται
      //το priority σε nice και επιστρέφεται
```

/ kernel / sched / core.c

```
6945  SYSCALL_DEFINE1(nice, int, increment)
6946  {
6947          long nice, retval;
6948
6949          /*
6950           * Setpriority might change our priority at the same moment.
6951           * We don't have to worry. Conceptually one call occurs first
6952           * and we have a single winner.
6953           */
6954          increment = clamp(increment, -NICE_WIDTH, NICE_WIDTH);
6955          nice = task_nice(current) + increment;
6956
6957          nice = clamp_val(nice, MIN_NICE, MAX_NICE);
6958          if (increment < 0 && !can_nice(current, nice))
6959                  return -EPERM;
6960
6961          retval = security_task_setnice(current, nice);
6962          if (retval)
6963                  return retval;
6964
6965          set_user_nice(current, nice);
6966          return 0;
6967  }
```

/ include / linux / minmax.h

```
137   #define clamp_val(val, lo, hi) clamp_t(typeof(val), val, lo, hi)
      //επιστρέφει μια τιμή "clamped" στο εύρος lo..hi
```

/ kernel / sched / core.c

```
6945  SYSCALL_DEFINE1(nice, int, increment)
6946  {
6947          long nice, retval;
6948
6949          /*
6950           * Setpriority might change our priority at the same moment.
6951           * We don't have to worry. Conceptually one call occurs first
6952           * and we have a single winner.
6953           */
6954          increment = clamp(increment, -NICE_WIDTH, NICE_WIDTH);
6955          nice = task_nice(current) + increment;
6956
6957          nice = clamp_val(nice, MIN_NICE, MAX_NICE);
6958          if (increment < 0 && !can_nice(current, nice))
6959                  return -EPERM;
6960
6961          retval = security_task_setnice(current, nice);
6962          if (retval)
6963                  return retval;
6964
6965          set_user_nice(current, nice);
6966          return 0;
6967  }
```

```
//ελέγχεται αν το increment είναι αρνητικό καθώς και
//μέσα από τη can_nice μαθαίνουμε αν μία διεργασία
//μπορεί να μειώσει το nice value της

//αν το increment είναι αρνητικό και η διεργασία δεν
//επιτρέπεται να μειώσει το nice value της, επιστρέφεται
//η τιμή 1 - Operation not permitted
```

/ kernel / sched / core.c

```
6945  SYSCALL_DEFINE1(nice, int, increment)
6946  {
6947          long nice, retval;
6948
6949          /*
6950           * Setpriority might change our priority at the same moment.
6951           * We don't have to worry. Conceptually one call occurs first
6952           * and we have a single winner.
6953           */
6954          increment = clamp(increment, -NICE_WIDTH, NICE_WIDTH);
6955          nice = task_nice(current) + increment;
6956
6957          nice = clamp_val(nice, MIN_NICE, MAX_NICE);
6958          if (increment < 0 && !can_nice(current, nice))
6959                  return -EPERM;
6960
6961          retval = security_task_setnice(current, nice);
6962          if (retval)
6963                  return retval;
6964
6965          set_user_nice(current, nice);
6966          return 0;
6967  }
```

/ include / linux / security.h

```
1130  static inline int security_task_setnice(struct task_struct *p, int nice)
1131  {
1132          return cap_task_setnice(p, nice);
1133  }
```

/ security / commoncap.c

```
1228  int cap_task_setnice(struct task_struct *p, int nice)
1229  {
1230          return cap_safe_nice(p);
1231  }
```

```
//ελέγχεται αν είναι επιτρεπτό να θέσουμε το nice value
//που έχει υπολογισθεί, στη συγκεκριμένη διεργασία. αν
//είναι επιτρεπτό, επιστρέφεται η τιμή 0
```

/ kernel / sched / core.c

```
6945  SYSCALL_DEFINE1(nice, int, increment)
6946  {
6947          long nice, retval;
6948
6949          /*
6950           * Setpriority might change our priority at the same moment.
6951           * We don't have to worry. Conceptually one call occurs first
6952           * and we have a single winner.
6953           */
6954          increment = clamp(increment, -NICE_WIDTH, NICE_WIDTH);
6955          nice = task_nice(current) + increment;
6956
6957          nice = clamp_val(nice, MIN_NICE, MAX_NICE);
6958          if (increment < 0 && !can_nice(current, nice))
6959                  return -EPERM;
6960
6961          retval = security_task_setnice(current, nice);
6962          if (retval)             //αν δεν είναι επιτρεπτό να κάνουμε set το nice value
6963                  return retval; //στη συγκεκριμένη διεργασία, μπαίνει μέσα στην if
6964                                 //και επιστρέφεται η τιμή του retval
6965          set_user_nice(current, nice);
6966          return 0;
6967  }
```

/ kernel / sched / core.c

```
6945  SYSCALL_DEFINE1(nice, int, increment)
6946  {
6947          long nice, retval;
6948
6949          /*
6950           * Setpriority might change our priority at the same moment.
6951           * We don't have to worry. Conceptually one call occurs first
6952           * and we have a single winner.
6953           */
6954          increment = clamp(increment, -NICE_WIDTH, NICE_WIDTH);
6955          nice = task_nice(current) + increment;
6956
6957          nice = clamp_val(nice, MIN_NICE, MAX_NICE);
6958          if (increment < 0 && !can_nice(current, nice))
6959                  return -EPERM;
6960
6961          retval = security_task_setnice(current, nice);
6962          if (retval)
6963                  return retval;
6964
6965        ▶ set_user_nice(current, nice);//καλείται η set_user_nice
6966          return 0;
6967  }
```

/ kernel / sched / core.c

```
6868  void set_user_nice(struct task_struct *p, long nice)
6869  {
6870          bool queued, running;
6871          int old_prio;
6872          struct rq_flags rf;
6873          struct rq *rq;
6874
6875          if (task_nice(p) == nice || nice < MIN_NICE || nice > MAX_NICE)
6876                  return;
6881          rq = task_rq_lock(p, &rf);
6882          update_rq_clock(rq);
6883
6890          if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
6891                  p->static_prio = NICE_TO_PRIO(nice);
6892                  goto out_unlock;
6893          }
6894          queued = task_on_rq_queued(p);
6895          running = task_current(rq, p);
6896          if (queued)
6897                  dequeue_task(rq, p, DEQUEUE_SAVE | DEQUEUE_NOCLOCK);
6898          if (running)
6899                  put_prev_task(rq, p);
6900
6901          p->static_prio = NICE_TO_PRIO(nice);
6902          set_load_weight(p, true);
6903          old_prio = p->prio;
6904          p->prio = effective_prio(p);
6905
6906          if (queued)
6907                  enqueue_task(rq, p, ENQUEUE_RESTORE | ENQUEUE_NOCLOCK);
6908          if (running)
6909                  set_next_task(rq, p);
6910
6915          p->sched_class->prio_changed(rq, p, old_prio);
6916
6917  out_unlock:
6918          task_rq_unlock(rq, p, &rf);
6919  }
6920  EXPORT_SYMBOL(set_user_nice);
```

/ kernel / sched / core.c

```
6868  void set_user_nice(struct task_struct *p, long nice)
6869  {
6870          bool queued, running;
6871          int old_prio;
6872          struct rq_flags rf;
6873          struct rq *rq;
6874
6875          if (task_nice(p) == nice || nice < MIN_NICE || nice > MAX_NICE) //ελέγχει αν το nice που θέλουμε να θέσουμε
6876                  return;                                                 είναι ίδιο με υπάρχον nice
6881          rq = task_rq_lock(p, &rf);                                      είναι μικρότερο του MIN_NICE
6882          update_rq_clock(rq);                                            είναι μεγαλύτερο του MAX_NICE
6883
6890          if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
6891                  p->static_prio = NICE_TO_PRIO(nice);
6892                  goto out_unlock;
6893          }
6894          queued = task_on_rq_queued(p);
6895          running = task_current(rq, p);
6896          if (queued)
6897                  dequeue_task(rq, p, DEQUEUE_SAVE | DEQUEUE_NOCLOCK);
6898          if (running)
6899                  put_prev_task(rq, p);
6900
6901          p->static_prio = NICE_TO_PRIO(nice);
6902          set_load_weight(p, true);
6903          old_prio = p->prio;
6904          p->prio = effective_prio(p);
6905
6906          if (queued)
6907                  enqueue_task(rq, p, ENQUEUE_RESTORE | ENQUEUE_NOCLOCK);
6908          if (running)
6909                  set_next_task(rq, p);
6910
6915          p->sched_class->prio_changed(rq, p, old_prio);
6916
6917  out_unlock:
6918          task_rq_unlock(rq, p, &rf);
6919  }
6920  EXPORT_SYMBOL(set_user_nice);
```

/ include / linux / sched.h

```
1838  static inline int task_nice(const struct task_struct *p)
1839  {
1840          return PRIO_TO_NICE((p)->static_prio);
1841  }
```

//μέσα από το PRIO_TO_NICE macro, μετατρέπεται
//το priority σε nice και επιστρέφεται

/ kernel / sched / core.c

```
6868  void set_user_nice(struct task_struct *p, long nice)
6869  {
6870          bool queued, running;
6871          int old_prio;
6872          struct rq_flags rf;
6873          struct rq *rq;
6874
6875          if (task_nice(p) == nice || nice < MIN_NICE || nice > MAX_NICE)
6876                  return;
6881          rq = task_rq_lock(p, &rf);
6882          update_rq_clock(rq);
6883
6890          if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
6891                  p->static_prio = NICE_TO_PRIO(nice); //το niceval μετατρέπεται σε priority value
6892                  goto out_unlock;                     //και το θέτουμε στο static_prio της διεργασίας
6893          }
6894          queued = task_on_rq_queued(p);
6895          running = task_current(rq, p);
6896          if (queued)
6897                  dequeue_task(rq, p, DEQUEUE_SAVE | DEQUEUE_NOCLOCK);
6898          if (running)
6899                  put_prev_task(rq, p);
6900
6901          p->static_prio = NICE_TO_PRIO(nice);
6902          set_load_weight(p, true);
6903          old_prio = p->prio;
6904          p->prio = effective_prio(p);
6905
6906          if (queued)
6907                  enqueue_task(rq, p, ENQUEUE_RESTORE | ENQUEUE_NOCLOCK);
6908          if (running)
6909                  set_next_task(rq, p);
6910
6915          p->sched_class->prio_changed(rq, p, old_prio);
6916
6917  out_unlock:
6918          task_rq_unlock(rq, p, &rf);
6919  }
6920  EXPORT_SYMBOL(set_user_nice);
```

λόγω χρόνου, δεν θα εξηγηθούν τα policies διεργασιών, αλλά αν
ενδιαφέρεστε να μάθετε μπορείτε να δείτε τα παρακάτω:

https://red.ht/3Eh7abj

Λειτουργικά Συστήματα - Αρχές Σχεδίασης σελ.436+

/ kernel / sched / core.c

```
6868  void set_user_nice(struct task_struct *p, long nice)
6869  {
6870          bool queued, running;
6871          int old_prio;
6872          struct rq_flags rf;
6873          struct rq *rq;
6874
6875          if (task_nice(p) == nice || nice < MIN_NICE || nice > MAX_NICE)
6876                  return;
6881          rq = task_rq_lock(p, &rf);
6882          update_rq_clock(rq);
6883
6890          if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
6891                  p->static_prio = NICE_TO_PRIO(nice);
6892                  goto out_unlock;
6893          }
6894          queued = task_on_rq_queued(p);
6895          running = task_current(rq, p);
6896          if (queued)
6897                  dequeue_task(rq, p, DEQUEUE_SAVE | DEQUEUE_NOCLOCK);
6898          if (running)
6899                  put_prev_task(rq, p);
6900
6901          p->static_prio = NICE_TO_PRIO(nice);
6902          set_load_weight(p, true);
6903          old_prio = p->prio;
6904          p->prio = effective_prio(p);
6905
6906          if (queued)
6907                  enqueue_task(rq, p, ENQUEUE_RESTORE | ENQUEUE_NOCLOCK);
6908          if (running)
6909                  set_next_task(rq, p);
6910
6915          p->sched_class->prio_changed(rq, p, old_prio);
6916
6917  out_unlock:
6918          task_rq_unlock(rq, p, &rf);
6919  }
6920  EXPORT_SYMBOL(set_user_nice);
```

/ kernel / sched / sched.h

```
2033  static inline int task_on_rq_queued(struct task_struct *p)
2034  {
2035          return p->on_rq == TASK_ON_RQ_QUEUED;
2036  }
```

task->on_rq έχει τρεις καταστάσεις:
0 η διεργασία δεν είναι στη runqueue (rq)
1 (TASK_ON_RQ_QUEUED) - η διεργασία είναι στη runqueue
2 (TASK_ON_RQ_MIGRATING) - η διεργασία βρίσκεται στη runqueue αλλά σε διαδικασία "μετανάστευσης" σε άλλη runqueue

/ kernel / sched / core.c

```
6868  void set_user_nice(struct task_struct *p, long nice)
6869  {
6870          bool queued, running;
6871          int old_prio;
6872          struct rq_flags rf;
6873          struct rq *rq;
6874
6875          if (task_nice(p) == nice || nice < MIN_NICE || nice > MAX_NICE)
6876                  return;
6881          rq = task_rq_lock(p, &rf);
6882          update_rq_clock(rq);
6883
6890          if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
6891                  p->static_prio = NICE_TO_PRIO(nice);
6892                  goto out_unlock;
6893          }
6894          queued = task_on_rq_queued(p);
6895          running = task_current(rq, p);
6896          if (queued)
6897                  dequeue_task(rq, p, DEQUEUE_SAVE | DEQUEUE_NOCLOCK);
6898          if (running)
6899                  put_prev_task(rq, p);
6900
6901          p->static_prio = NICE_TO_PRIO(nice);
6902          set_load_weight(p, true);
6903          old_prio = p->prio;
6904          p->prio = effective_prio(p);
6905
6906          if (queued)
6907                  enqueue_task(rq, p, ENQUEUE_RESTORE | ENQUEUE_NOCLOCK);
6908          if (running)
6909                  set_next_task(rq, p);
6910
6915          p->sched_class->prio_changed(rq, p, old_prio);
6916
6917  out_unlock:
6918          task_rq_unlock(rq, p, &rf);
6919  }
6920  EXPORT_SYMBOL(set_user_nice);
```

/ kernel / sched / sched.h

```
2019  static inline int task_current(struct rq *rq, struct task_struct *p)
2020  {
2021          return rq->curr == p;
2022  }
```

//ελέγχει αν το task που εκτελείται τώρα είναι το task του οποίου
//θέλουμε να αλλάξουμε το nice. Αν είναι επιστρέφει τη τιμή 1.

/ kernel / sched / core.c

```
6868  void set_user_nice(struct task_struct *p, long nice)
6869  {
6870          bool queued, running;
6871          int old_prio;
6872          struct rq_flags rf;
6873          struct rq *rq;
6874
6875          if (task_nice(p) == nice || nice < MIN_NICE || nice > MAX_NICE)
6876                  return;
6881          rq = task_rq_lock(p, &rf);
6882          update_rq_clock(rq);
6883
6890          if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
6891                  p->static_prio = NICE_TO_PRIO(nice);
6892                  goto out_unlock;
6893          }
6894          queued = task_on_rq_queued(p);
6895          running = task_current(rq, p);
6896          if (queued)                                          //εάν το task βρίσκεται σε runqueue
6897                  dequeue_task(rq, p, DEQUEUE_SAVE | DEQUEUE_NOCLOCK); //βγάλτο από τη runqueue
6898          if (running)
6899                  put_prev_task(rq, p);
6900
6901          p->static_prio = NICE_TO_PRIO(nice);
6902          set_load_weight(p, true);
6903          old_prio = p->prio;
6904          p->prio = effective_prio(p);
6905
6906          if (queued)
6907                  enqueue_task(rq, p, ENQUEUE_RESTORE | ENQUEUE_NOCLOCK);
6908          if (running)
6909                  set_next_task(rq, p);
6910
6915          p->sched_class->prio_changed(rq, p, old_prio);
6916
6917  out_unlock:
6918          task_rq_unlock(rq, p, &rf);
6919  }
6920  EXPORT_SYMBOL(set_user_nice);
```

/ kernel / sched / core.c

```
6868  void set_user_nice(struct task_struct *p, long nice)
6869  {
6870          bool queued, running;
6871          int old_prio;
6872          struct rq_flags rf;
6873          struct rq *rq;
6874
6875          if (task_nice(p) == nice || nice < MIN_NICE || nice > MAX_NICE)
6876                  return;
6881          rq = task_rq_lock(p, &rf);
6882          update_rq_clock(rq);
6883
6890          if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
6891                  p->static_prio = NICE_TO_PRIO(nice);
6892                  goto out_unlock;
6893          }
6894          queued = task_on_rq_queued(p);
6895          running = task_current(rq, p);
6896          if (queued)
6897                  dequeue_task(rq, p, DEQUEUE_SAVE | DEQUEUE_NOCLOCK);
6898          if (running)                    //εάν το task εκτελείται
6899                  put_prev_task(rq, p);  //βγάλτο από τη CPU
6900
6901          p->static_prio = NICE_TO_PRIO(nice);
6902          set_load_weight(p, true);
6903          old_prio = p->prio;
6904          p->prio = effective_prio(p);
6905
6906          if (queued)
6907                  enqueue_task(rq, p, ENQUEUE_RESTORE | ENQUEUE_NOCLOCK);
6908          if (running)
6909                  set_next_task(rq, p);
6910
6915          p->sched_class->prio_changed(rq, p, old_prio);
6916
6917  out_unlock:
6918          task_rq_unlock(rq, p, &rf);
6919  }
6920  EXPORT_SYMBOL(set_user_nice);
```

/ kernel / sched / core.c

```
6868  void set_user_nice(struct task_struct *p, long nice)
6869  {
6870          bool queued, running;
6871          int old_prio;
6872          struct rq_flags rf;
6873          struct rq *rq;
6874
6875          if (task_nice(p) == nice || nice < MIN_NICE || nice > MAX_NICE)
6876                  return;
6881          rq = task_rq_lock(p, &rf);
6882          update_rq_clock(rq);
6883
6890          if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
6891                  p->static_prio = NICE_TO_PRIO(nice);
6892                  goto out_unlock;
6893          }
6894          queued = task_on_rq_queued(p);
6895          running = task_current(rq, p);
6896          if (queued)
6897                  dequeue_task(rq, p, DEQUEUE_SAVE | DEQUEUE_NOCLOCK);
6898          if (running)
6899                  put_prev_task(rq, p);
6900
6901          p->static_prio = NICE_TO_PRIO(nice)//το niceval μετατρέπεται σε priority value
6902          set_load_weight(p, true);          //και το θέτουμε στο static_prio της διεργασίας
6903          old_prio = p->prio;
6904          p->prio = effective_prio(p);
6905
6906          if (queued)
6907                  enqueue_task(rq, p, ENQUEUE_RESTORE | ENQUEUE_NOCLOCK);
6908          if (running)
6909                  set_next_task(rq, p);
6910
6915          p->sched_class->prio_changed(rq, p, old_prio);
6916
6917  out_unlock:
6918          task_rq_unlock(rq, p, &rf);
6919  }
6920  EXPORT_SYMBOL(set_user_nice);
```

/ kernel / sched / core.c

```
6868  void set_user_nice(struct task_struct *p, long nice)
6869  {
6870          bool queued, running;
6871          int old_prio;
6872          struct rq_flags rf;
6873          struct rq *rq;
6874
6875          if (task_nice(p) == nice || nice < MIN_NICE || nice > MAX_NICE)
6876                  return;
6881          rq = task_rq_lock(p, &rf);
6882          update_rq_clock(rq);
6883
6890          if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
6891                  p->static_prio = NICE_TO_PRIO(nice);
6892                  goto out_unlock;
6893          }
6894          queued = task_on_rq_queued(p);
6895          running = task_current(rq, p);
6896          if (queued)
6897                  dequeue_task(rq, p, DEQUEUE_SAVE | DEQUEUE_NOCLOCK);
6898          if (running)
6899                  put_prev_task(rq, p);
6900
6901          p->static_prio = NICE_TO_PRIO(nice);
6902          set_load_weight(p, true);//τελικά φτάνουμε σε αυτό το σημείο όπου καλείται η set_load_weight
6903          old_prio = p->prio;
6904          p->prio = effective_prio(p);
6905
6906          if (queued)
6907                  enqueue_task(rq, p, ENQUEUE_RESTORE | ENQUEUE_NOCLOCK);
6908          if (running)
6909                  set_next_task(rq, p);
6910
6915          p->sched_class->prio_changed(rq, p, old_prio);
6916
6917  out_unlock:
6918          task_rq_unlock(rq, p, &rf);
6919  }
6920  EXPORT_SYMBOL(set_user_nice);
```

set_load_weight

Μέσα από τη set_load_weight υπολογίζεται και το weight της διεργασίας.

Το weight της κάθε διεργασίας είναι **περίπου** ίσο με **1024 / (1.25) ^ (nice)**

set_load_weight

Το weight μπορεί επίσης να υπολογισθεί με βάση τον ακόλουθο πίνακα:

nice value	weight
-20	1024 * 1.25^20
[-19, -1]	~ 1024 * 1.25^-nice
0	1024
[1, 18]	~ 1024 * 1.25^-nice
19	1024 * 1.25^-19

Γιατί είναι όμως το weight περίπου ίσο με **1024 / (1.25) ^ (nice);**

set_load_weight

Η απάντηση είναι, γιατί με αυτό το τρόπο σχεδιάστηκε ο Completely Fair Scheduler.

Συγκεκριμένα ο CFS σχεδιάστηκε με το σκεπτικό:

nice value αυξάνεται κατά 1 -> ~10% λιγότερη CPU

nice value μειώνεται κατά 1 -> ~10% περισσότερη CPU

Για να πετύχουμε αυτό το σκεπτικό, χρησιμοποιείται πολλαπλασιασμός με το 1.25.

set_load_weight

Αυτό το σκεπτικό υλοποιείται στον sched_prio_to_weight[40] πίνακα

/ kernel / sched / core.c

```
10814   const int sched_prio_to_weight[40] = {
10815    /* -20 */     88761,     71755,     56483,     46273,     36291,
10816    /* -15 */     29154,     23254,     18705,     14949,     11916,
10817    /* -10 */      9548,      7620,      6100,      4904,      3906,
10818    /*  -5 */      3121,      2501,      1991,      1586,      1277,
10819    /*   0 */      1024,       820,       655,       526,       423,
10820    /*   5 */       335,       272,       215,       172,       137,
10821    /*  10 */       110,        87,        70,        56,        45,
10822    /*  15 */        36,        29,        23,        18,        15,
10823   };
```

Παρατηρούμε πως όσο το nice μειώνεται, το weight αυξάνεται και επομένως αυξάνεται το ποσοστό χρόνου της CPU που θα δοθεί στη διεργασία. Γιατί;

set_load_weight

Είχαμε πει πως τα νέα κβάντα μίας διεργασίας υπολογίζονται με βάση το τύπο: **TL * (K/M)**

Όπου K = 1024 / 1.25 ^ (nice) και

Όπου M = $\sum_{i=1}^{N}(K)$

Το K *είναι* το weight, και το M *είναι* το weight της rq.

Επομένως ο χρόνος της CPU που παίρνει κάθε διεργασία είναι **TL*(weight(task)/weight(rq))**

Παράδειγμα 1

Για να δούμε καλύτερα την επίδραση των παραπάνω, ας υποθέσουμε πως έχουμε 2 διεργασίες pA και pB όπου:

	nice value	weight
A	0	1024
B	0	1024

Το weight προκύπτει με βάση το πίνακα της διαφάνειας 24.

Αν υποθέσουμε πως δεν τρέχουν άλλες διεργασίες τότε στη rq (run queue) της CPU υπάρχουν 2 διεργασίες.

Το weight της rq = 1024 + 1024 = 2048

CPUpercpX = weight(pX) / weight(rq)

CPUpercpA = 1024 / 2048 = 50%

Αντίστοιχα και το CPUpercpB θα είναι = 50%.

Παράδειγμα 1

Στη συνέχεια κάνουμε τις ίδιες υποθέσεις, με τη διαφορά πως το nice value της A θα είναι 1.

	nice value	weight
A	1	820
B	0	1024

Το weight της rq θα είναι: 820 + 1024 = 1844

Επομένως το ποσοστό χρόνου στη CPU που θα πάρει η A είναι:

CPUpercpA = 820 / 1844 = ~45%

CPUpercpB = 1024 / 1844 = ~55%

Επομένως παρατηρούμε πως αυξάνοντας τη τιμή του nice της A κατά 1 πλέον το ποσοστό χρόνου της CPU που παίρνει είναι ~10% λιγότερο απ’ ότι αν είχε nice 0.

Παράδειγμα 2

Υποθέτουμε πως έχουμε 2 διεργασίες pA και pB όπου:

	nice value	weight
A	-1	1277
B	0	1024

Αν υποθέσουμε πως δεν τρέχουν άλλες διεργασίες τότε στο rq (run queue) της CPU υπάρχουν 2 διεργασίες.

Το weight της rq = 1277 + 1024 = 2301

CPUpercpA = 1277 / 2301 = ~55%

Αντίστοιχα το CPUpercpB θα είναι ~45%.

Παράδειγμα 2

Στη συνέχεια κάνουμε τις ίδιες υποθέσεις, με τη διαφορά πως το nice value της B θα είναι 1.

	nice value	weight
A	-1	1277
B	1	820

Το weight της rq θα είναι: 1277 + 820 = 2097

Επομένως το ποσοστό χρόνου στη CPU που θα πάρει η A είναι:

CPUpercpA = 1277 / 2097 ~ 60%

CPUpercpB = 820 / 2097 ~ 40%

Επομένως παρατηρούμε πως αυξάνοντας τη τιμή του nice της B κατά 1 πλέον το ποσοστό χρόνου της CPU που παίρνει είναι ~10% λιγότερο απ’ ότι αν είχε nice 0.