Ordonnancement de processus

Noyaux Multi-cœurs et Virtualisation

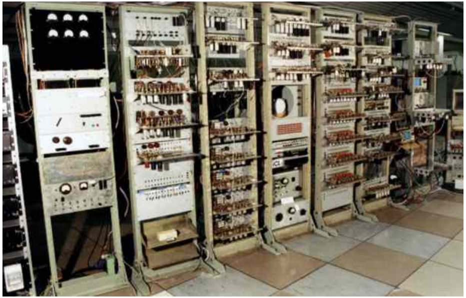
Redha GOUICEM

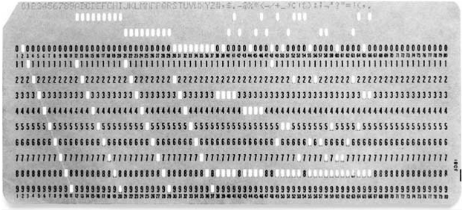
Sorbonne Université

2020

Histoire

Manchester Mark 1 (1949), premier ordinateur électronique à mémoire
Programmes stockés sur **cartes perforées** placées “à la main”





Modèle économique → Location de temps machine à des entreprises

Partage de ressources

1 processeur, N tâches → plusieurs traitements possibles:

- **par lots** (*batch*): chaque tâche a un accès exclusif au processeur jusqu'à sa terminaison

Partage de ressources

1 processeur, N tâches → plusieurs traitements possibles:

- **par lots** (*batch*): chaque tâche a un accès exclusif au processeur jusqu'à sa terminaison

- **multi-programmation**: lorsqu'une tâche est bloquée, une autre peut s'exécuter

Partage de ressources

1 processeur, N tâches → plusieurs traitements possibles:

- **par lots** (*batch*): chaque tâche a un accès exclusif au processeur jusqu'à sa terminaison

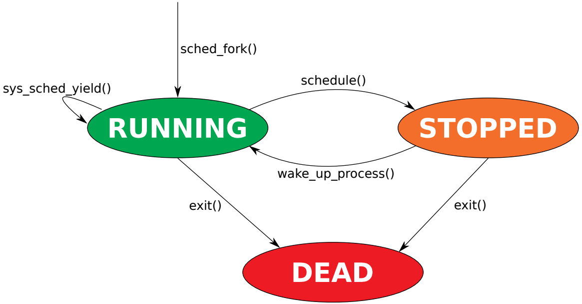
- **multi-programmation**: lorsqu'une tâche est bloquée, une autre peut s'exécuter

- **temps partagé** (*time sharing*): les tâches utilisent alternativement le processeur pendant leur exécution

Une tâche (processus/thread) peut être dans l'état:

RUNNING, STOPPED, DEAD

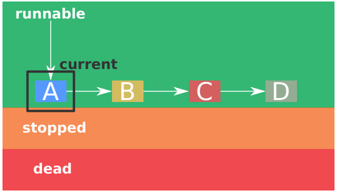
```c
struct task_struct {
        long state;
        pid_t pid;
};

/* Choix prochaine tache */
void schedule(void);
/* Réveil d'une tache */
int wake_up_process(struct task_struct *p);
/* Appel systeme yield() */
long sys_sched_yield(void);
/* Fork/clone d'une tache */
void sched_fork(struct task_struct *p);
/* Terminaison d'une tache */
void exit(struct task_struct *p);
/* Handler du tick horloge */
void tick(struct task_struct *p);

/* Tache courante */
struct task_struct *current;
```



Ordonnanceur round-robin (Linux 1.2)

Les tâches **RUNNING** sont rangées dans une FIFO `runnable`

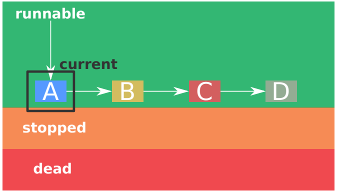
```c
struct list_head runnable;

struct task_struct {
  int quantum;
  struct list_head list;
};

void schedule(void)
{
  struct task_struct *next;
  struct task_struct *prev = current;

  /* Si STOPPED, pop() */
  if (!(prev->state & TASK_RUNNING))
    list_del(&prev->list);

  /* Si une tache RUNNING, choisir */
  if (!list_empty(&runnable))
    next = list_first_entry(&runnable,
                            struct task_struct,
                            list);
  /* Sinon, devenir idle */
  else
    next = &idle_task;

  context_switch(prev, next);
}
```

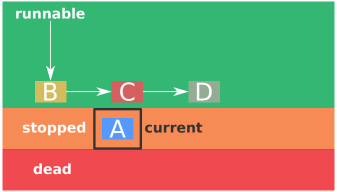


Ordonnanceur round-robin (Linux 1.2)

Les tâches **RUNNING** sont rangées dans une FIFO `runnable`

```c
struct list_head runnable;

struct task_struct {
  int quantum;
  struct list_head list;
};

void schedule(void)
{
  struct task_struct *next;
  struct task_struct *prev = current;

  /* Si STOPPED, pop() */
  if (!(prev->state & TASK_RUNNING))
    list_del(&prev->list);

  /* Si une tache RUNNING, choisir */
  if (!list_empty(&runnable))
    next = list_first_entry(&runnable,
                            struct task_struct,
                            list);
  /* Sinon, devenir idle */
  else
    next = &idle_task;

  context_switch(prev, next);
}
```

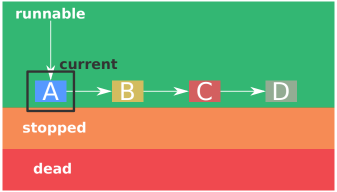


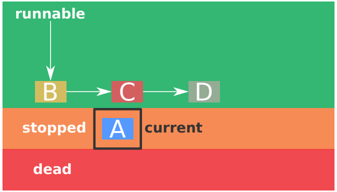


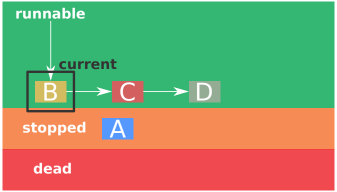
Ordonnanceur round-robin (Linux 1.2)

Les tâches **RUNNING** sont rangées dans une FIFO `runnable`

```c
struct list_head runnable;

struct task_struct {
  int quantum;
  struct list_head list;
};

void schedule(void)
{
  struct task_struct *next;
  struct task_struct *prev = current;

  /* Si STOPPED, pop() */
  if (!(prev->state & TASK_RUNNING))
    list_del(&prev->list);

  /* Si une tache RUNNING, choisir */
  if (!list_empty(&runnable))
    next = list_first_entry(&runnable,
                            struct task_struct,
                            list);
  /* Sinon, devenir idle */
  else
    next = &idle_task;

  context_switch(prev, next);
}
```

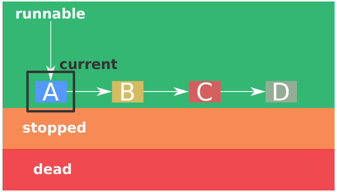






Ordonnanceur round-robin (Linux 1.2)

```c
void tick(void)
{
  current->quantum--;

  /* Si quantum epuise, preemption */
  if (!current->quantum) {
    list_del(&current->list);
    current->quantum = QUANTUM;
    list_add_tail(&current->list,
                  &runnable);
    schedule();
  }
}
```

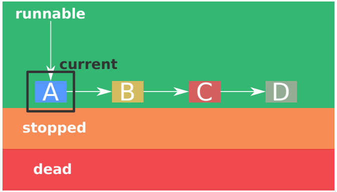


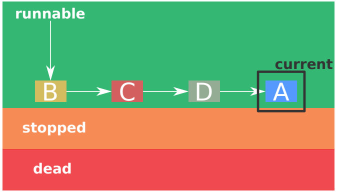

```c
void sys_sched_yield(void)
{
  list_del(&current->list);
  current->quantum = QUANTUM;
  list_add_tail(&current->list,
                &runnable);
  schedule();
}
```

Ordonnanceur round-robin (Linux 1.2)

```c
void tick(void)
{
  current->quantum--;

  /* Si quantum epuise, preemption */
  if (!current->quantum) {
    list_del(&current->list);
    current->quantum = QUANTUM;
    list_add_tail(&current->list,
                  &runnable);
    schedule();
  }
}
```

```c
void sys_sched_yield(void)
{
  list_del(&current->list);
  current->quantum = QUANTUM;
  list_add_tail(&current->list,
                &runnable);
  schedule();
}
```

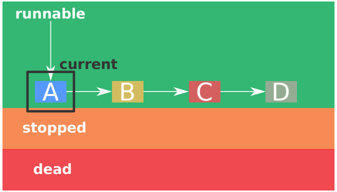


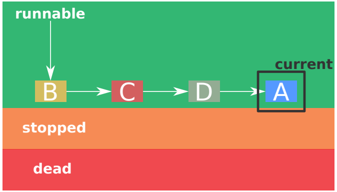


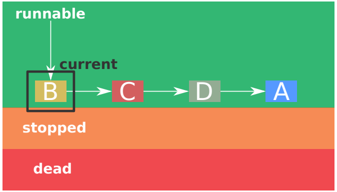
Ordonnanceur round-robin (Linux 1.2)

```
void tick(void)
{
  current->quantum--;

  /* Si quantum epuise, preemption */
  if (!current->quantum) {
    list_del(&current->list);
    current->quantum = QUANTUM;
    list_add_tail(&current->list,
                  &runnable);
    schedule();
  }
}
```

```
void sys_sched_yield(void)
{
  list_del(&current->list);
  current->quantum = QUANTUM;
  list_add_tail(&current->list,
                &runnable);
  schedule();
}
```

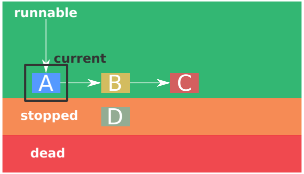






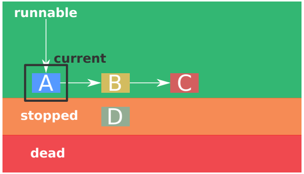
Ordonnanceur round-robin (Linux 1.2)

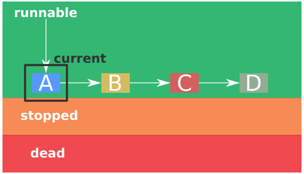
```
int wake_up_process(struct task_struct *p)
{
  p->state = TASK_RUNNING;
  current->quantum = QUANTUM;
  list_add_tail(&p->list,
                &runnable);
}

void sched_fork(struct task_struct *p)
{
  p->state = TASK_RUNNING;
  current->quantum = QUANTUM;
  list_add_tail(&p->list,
                &runnable);
}


void exit(struct task_struct *p)
{
  p->state = TASK_DEAD;
  list_del(&p->list);

  if (current == p)
    schedule();
}
```



Ordonnanceur round-robin (Linux 1.2)

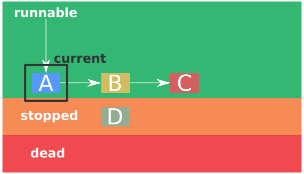
```c
int wake_up_process(struct task_struct *p)
{
  p->state = TASK_RUNNING;
  current->quantum = QUANTUM;
  list_add_tail(&p->list,
                &runnable);
}

void sched_fork(struct task_struct *p)
{
  p->state = TASK_RUNNING;
  current->quantum = QUANTUM;
  list_add_tail(&p->list,
                &runnable);
}


void exit(struct task_struct *p)
{
  p->state = TASK_DEAD;
  list_del(&p->list);

  if (current == p)
    schedule();
}
```

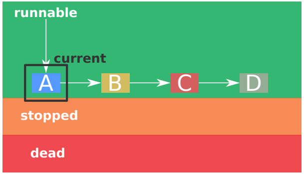


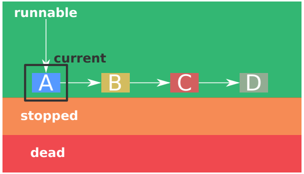


Ordonnanceur round-robin (Linux 1.2)

```c
int wake_up_process(struct task_struct *p)
{
  p->state = TASK_RUNNING;
  current->quantum = QUANTUM;
  list_add_tail(&p->list,
                &runnable);
}

void sched_fork(struct task_struct *p)
{
  p->state = TASK_RUNNING;
  current->quantum = QUANTUM;
  list_add_tail(&p->list,
                &runnable);
}


void exit(struct task_struct *p)
{
  p->state = TASK_DEAD;
  list_del(&p->list);

  if (current == p)
    schedule();
}
```

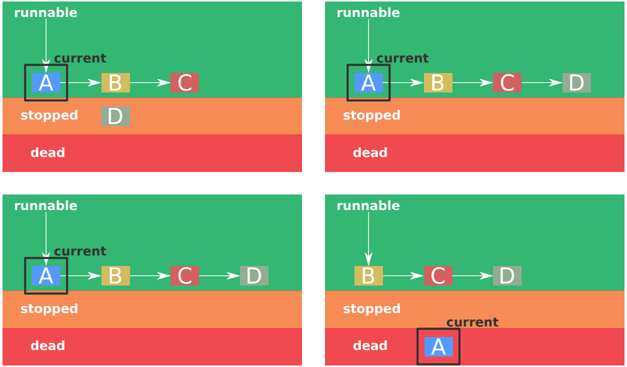






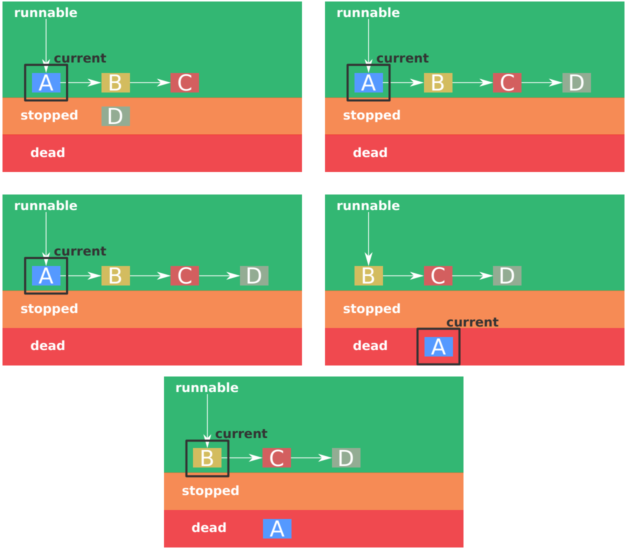
Ordonnanceur round-robin (Linux 1.2)

```
int wake_up_process(struct task_struct *p)
{
  p->state = TASK_RUNNING;
  current->quantum = QUANTUM;
  list_add_tail(&p->list,
                &runnable);
}

void sched_fork(struct task_struct *p)
{
  p->state = TASK_RUNNING;
  current->quantum = QUANTUM;
  list_add_tail(&p->list,
                &runnable);
}


void exit(struct task_struct *p)
{
  p->state = TASK_DEAD;
  list_del(&p->list);

  if (current == p)
    schedule();
}
```



Ordonnanceur round-robin (Linux 1.2)

```c
int wake_up_process(struct task_struct *p)
{
  p->state = TASK_RUNNING;
  current->quantum = QUANTUM;
  list_add_tail(&p->list,
                &runnable);
}

void sched_fork(struct task_struct *p)
{
  p->state = TASK_RUNNING;
  current->quantum = QUANTUM;
  list_add_tail(&p->list,
                &runnable);
}


void exit(struct task_struct *p)
{
  p->state = TASK_DEAD;
  list_del(&p->list);

  if (current == p)
    schedule();
}
```



Ordonnanceur round-robin (Linux 1.2)

Avantages:

+ Complexité: $O(1)$
+ Simplicité

Inconvénients:

– *Inéquité*: tâches interactives désavantagées (cf. C)

– *Interactivité*: latence sensible à la charge (cf. C)

Ordonnanceur round-robin (Linux 1.2)

Avantages:

+ Complexité: $O(1)$
+ Simplicité

Inconvénients:

– *Inéquité*: tâches interactives désavantagées (cf. C)

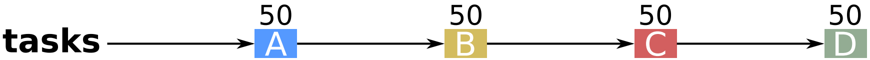
– *Interactivité*: latence sensible à la charge (cf. C)

Solution possible

Instauration d'un système de priorité entre les tâches

Ordonnanceur $O(N)$ (Linux 2.4)

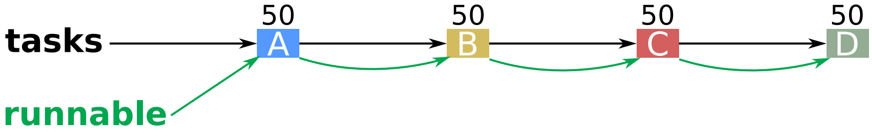
Toutes les tâches sont rangées dans une liste chaînée `tasks`



Ordonnanceur $O(N)$ (Linux 2.4)

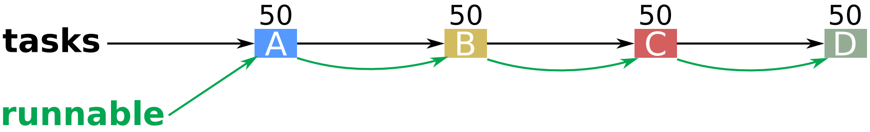
Toutes les tâches sont rangées dans une liste chaînée `tasks`

Les tâches **RUNNING** sont rangées dans une liste chaînée `runnable`



Ordonnanceur $O(N)$ (Linux 2.4)

Toutes les tâches sont rangées dans une liste chaînée `tasks`
Les tâches **RUNNING** sont rangées dans une liste chaînée `runnable`

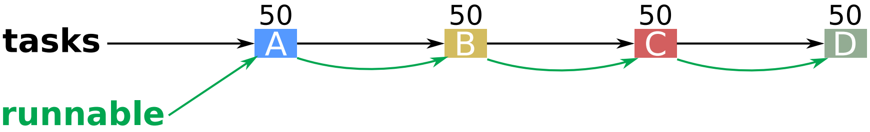



```
struct list_head runnable;
struct list_head tasks;

struct task_struct {
  int nice;
  int timeslice;
  struct list_head task_list;
  struct list_head run_list;
};
```

Ordonnanceur $O(N)$ (Linux 2.4)

Toutes les tâches sont rangées dans une liste chaînée `tasks`

Les tâches **RUNNING** sont rangées dans une liste chaînée `runnable`



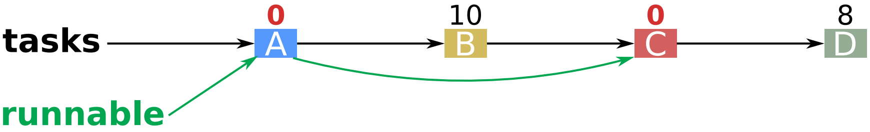

```
struct list_head runnable;
struct list_head tasks;

struct task_struct {
  int nice;
  int timeslice;
  struct list_head task_list;
  struct list_head run_list;
};

void schedule(void)
{
  struct task_struct *next, *prev = &current;
  struct task_struct *tmp;

  /* Si STOPPED, retirer de runnable */
  if (!(prev->state & TASK_RUNNING))
    list_del(&prev->run_list);
```

```
retry:
  next = &idle_task;
  /* Si RUNNING dispo */
  if (!list_empty(&runnable)) {
    /* Recherche meilleure goodness */
    list_for_each_entry(tmp, &runnable, run_list)
      if (goodness(tmp) > goodness(next))
        next = tmp;
    /* Si toute timeslice epuisee, recalcul */
    if (!next->timeslice) {
      list_for_each_entry(tmp, &tasks, task_list)
        tmp->timeslice = tmp->nice + tmp->timeslice >> 2;
      goto retry;
    }
  }

  context_switch(prev, next);
}
```

Ordonnanceur $O(N)$ (Linux 2.4)

Toutes les tâches sont rangées dans une liste chaînée `tasks`

Les tâches **RUNNING** sont rangées dans une liste chaînée `runnable`



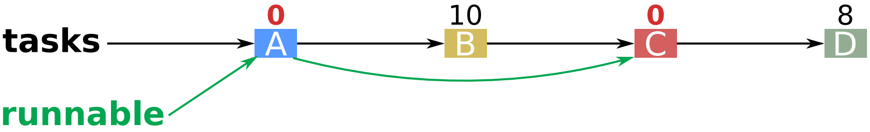

```c
struct list_head runnable;
struct list_head tasks;

struct task_struct {
  int nice;
  int timeslice;
  struct list_head task_list;
  struct list_head run_list;
};

void schedule(void)
{
  struct task_struct *next, *prev = &current;
  struct task_struct *tmp;

  /* Si STOPPED, retirer de runnable */
  if (!(prev->state & TASK_RUNNING))
    list_del(&prev->run_list);
```

```c
retry:
  next = &idle_task;
  /* Si RUNNING dispo */
  if (!list_empty(&runnable)) {
    /* Recherche meilleure goodness */
    list_for_each_entry(tmp, &runnable, run_list)
      if (goodness(tmp) > goodness(next))
        next = tmp;
    /* Si toute timeslice epuisee, recalcul */
    if (!next->timeslice) {
      list_for_each_entry(tmp, &tasks, task_list)
        tmp->timeslice = tmp->nice + tmp->timeslice >> 2;
      goto retry;
    }
  }

  context_switch(prev, next);
}
```

Ordonnanceur $O(N)$ (Linux 2.4)

Toutes les tâches sont rangées dans une liste chaînée `tasks`

Les tâches **RUNNING** sont rangées dans une liste chaînée `runnable`



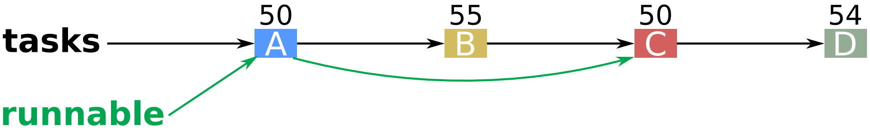

```c
struct list_head runnable;
struct list_head tasks;

struct task_struct {
  int nice;
  int timeslice;
  struct list_head task_list;
  struct list_head run_list;
};

void schedule(void)
{
  struct task_struct *next, *prev = &current;
  struct task_struct *tmp;

  /* Si STOPPED, retirer de runnable */
  if (!(prev->state & TASK_RUNNING))
    list_del(&prev->run_list);
```

```c
retry:
  next = &idle_task;
  /* Si RUNNING dispo */
  if (!list_empty(&runnable)) {
    /* Recherche meilleure goodness */
    list_for_each_entry(tmp, &runnable, run_list)
      if (goodness(tmp) > goodness(next))
        next = tmp;
    /* Si toute timeslice epuisee, recalcul */
    if (!next->timeslice) {
      list_for_each_entry(tmp, &tasks, task_list)
        tmp->timeslice = tmp->nice + tmp->timeslice >> 2;
      goto retry;
    }
  }

  context_switch(prev, next);
}
```

Ordonnanceur $O(N)$ (Linux 2.4)

Toutes les tâches sont rangées dans une liste chaînée `tasks`

Les tâches **RUNNING** sont rangées dans une liste chaînée `runnable`



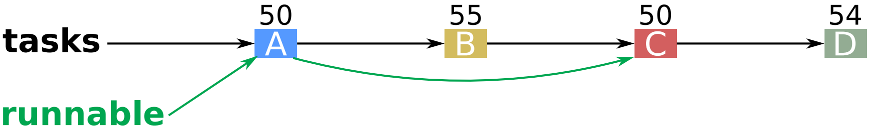

```c
struct list_head runnable;
struct list_head tasks;

struct task_struct {
  int nice;
  int timeslice;
  struct list_head task_list;
  struct list_head run_list;
};

void schedule(void)
{
  struct task_struct *next, *prev = &current;
  struct task_struct *tmp;

  /* Si STOPPED, retirer de runnable */
  if (!(prev->state & TASK_RUNNING))
    list_del(&prev->run_list);
```

```c
retry:
  next = &idle_task;
  /* Si RUNNING dispo */
  if (!list_empty(&runnable)) {
    /* Recherche meilleure goodness */
    list_for_each_entry(tmp, &runnable, run_list)
      if (goodness(tmp) > goodness(next))
        next = tmp;
    /* Si toute timeslice epuisee, recalcul */
    if (!next->timeslice) {
      list_for_each_entry(tmp, &tasks, task_list)
        tmp->timeslice = tmp->nice + tmp->timeslice >> 2;
      goto retry;
    }
  }

  context_switch(prev, next);
}
```

Ordonnanceur $O(N)$ (Linux 2.4)

Toutes les tâches sont rangées dans une liste chaînée `tasks`

Les tâches **RUNNING** sont rangées dans une liste chaînée `runnable`



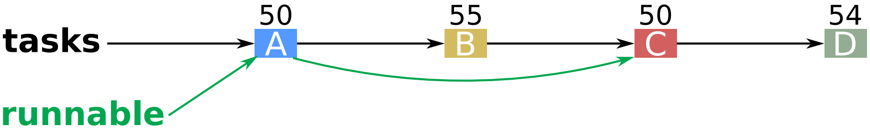

```c
struct list_head runnable;
struct list_head tasks;

struct task_struct {
  int nice;
  int timeslice;
  struct list_head task_list;
  struct list_head run_list;
};

void schedule(void)
{
  struct task_struct *next, *prev = &current;
  struct task_struct *tmp;

  /* Si STOPPED, retirer de runnable */
  if (!(prev->state & TASK_RUNNING))
    list_del(&prev->run_list);
```

```c
retry:
  next = &idle_task;
  /* Si RUNNING dispo */
  if (!list_empty(&runnable)) {
    /* Recherche meilleure goodness */
    list_for_each_entry(tmp, &runnable, run_list)
      if (goodness(tmp) > goodness(next))
        next = tmp;
    /* Si toute timeslice epuisee, recalcul */
    if (!next->timeslice) {
      list_for_each_entry(tmp, &tasks, task_list)
        tmp->timeslice = tmp->nice + tmp->timeslice >> 2;
      goto retry;
    }
  }

  context_switch(prev, next);
}
```

Ordonnanceur $O(N)$ (Linux 2.4)

Toutes les tâches sont rangées dans une liste chaînée `tasks`

Les tâches **RUNNING** sont rangées dans une liste chaînée `runnable`




```
struct list_head runnable;
struct list_head tasks;

struct task_struct {
  int nice;
  int timeslice;
  struct list_head task_list;
  struct list_head run_list;
};

void schedule(void)
{
  struct task_struct *next, *prev = &current;
  struct task_struct *tmp;

  /* Si STOPPED, retirer de runnable */
  if (!(prev->state & TASK_RUNNING))
    list_del(&prev->run_list);

retry:
  next = &idle_task;
  /* Si RUNNING dispo */
  if (!list_empty(&runnable)) {
    /* Recherche meilleure goodness */
    list_for_each_entry(tmp, &runnable, run_list)
      if (goodness(tmp) > goodness(next))
        next = tmp;
    /* Si toute timeslice epuisee, recalcul */
    if (!next->timeslice) {
      list_for_each_entry(tmp, &tasks, task_list)
        tmp->timeslice = tmp->nice + tmp->timeslice >> 2;
      goto retry;
    }
  }

  context_switch(prev, next);
}
```

Ordonnanceur $O(N)$ (Linux 2.4)

Avantages:

+ *Équité*: basée sur *nice*
+ *Interactivité*: basée sur *nice*

Inconvénients:

– *Complexité*: $O(N)$

Ordonnanceur $O(N)$ (Linux 2.4)

Avantages:

+ *Équité*: basée sur *nice*
+ *Interactivité*: basée sur *nice*

Inconvénients:

– *Complexité*: $O(N)$

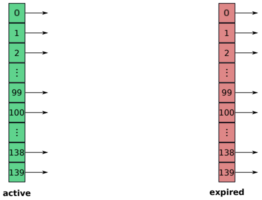
Solution possible

Changer de structure de données pour avoir les priorités **et** une meilleure complexité

Ordonnanceur $O(1)$ (Linux 2.6)

Les tâches **RUNNING** sont rangées dans une `struct rq`

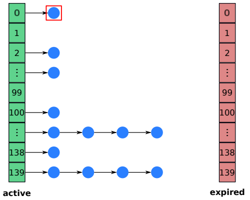
```
struct task_struct {
  struct list_head list;
  int timeslice;
  int prio;
};

struct prio_array {
  struct list_head array[140];
  unsigned int nr_tasks;
};

struct rq {
  struct prio_array *active;
  struct prio_array *expired;
};
```



Ordonnanceur $O(1)$ (Linux 2.6)

Les tâches **RUNNING** sont rangées dans une `struct rq`

```
struct task_struct {
  struct list_head list;
  int timeslice;
  int prio;
};

struct prio_array {
  struct list_head array[140];
  unsigned int nr_tasks;
};

struct rq {
  struct prio_array *active;
  struct prio_array *expired;
};
```

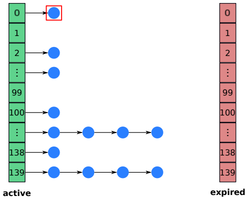



```
void tick(void)
{
  /* timeslice restante diminue */
  current->timeslice--;

  /* si timeslice epuisee, devenir expired */
  if (!current->timeslice) {
    update_prio(current);
    list_del(current->list);
    rq->active.nr_tasks--;
    list_add_tail(&current->list,
                  &rq->expired[current->prio]);
    rq->expired.nr_tasks++;
    update_timeslice(current);

    schedule();
  }
```

Ordonnanceur $O(1)$ (Linux 2.6)

Les tâches **RUNNING** sont rangées dans une `struct rq`

```c
struct task_struct {
  struct list_head list;
  int timeslice;
  int prio;
};

struct prio_array {
  struct list_head array[140];
  unsigned int nr_tasks;
};

struct rq {
  struct prio_array *active;
  struct prio_array *expired;
};
```

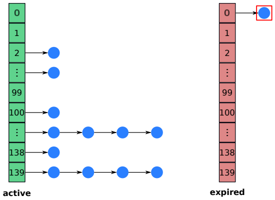



```c
void tick(void)
{
  /* timeslice restante diminue */
  current->timeslice--;

  /* si timeslice epuisee, devenir expired */
  if (!current->timeslice) {
    update_prio(current);
    list_del(current->list);
    rq->active.nr_tasks--;
    list_add_tail(&current->list,
                  &rq->expired[current->prio]);
    rq->expired.nr_tasks++;
    update_timeslice(current);

    schedule();
  }
```

Ordonnanceur $O(1)$ (Linux 2.6)

Les tâches **RUNNING** sont rangées dans une `struct rq`

```
struct task_struct {
  struct list_head list;
  int timeslice;
  int prio;
};

struct prio_array {
  struct list_head array[140];
  unsigned int nr_tasks;
};

struct rq {
  struct prio_array *active;
  struct prio_array *expired;
};
```

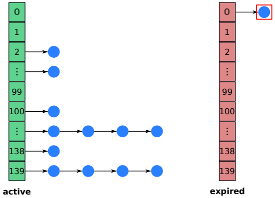



```
void tick(void)
{
  /* timeslice restante diminue */
  current->timeslice--;

  /* si timeslice epuisee, devenir expired */
  if (!current->timeslice) {
    update_prio(current);
    list_del(current->list);
    rq->active.nr_tasks--;
    list_add_tail(&current->list,
                  &rq->expired[current->prio]);
    rq->expired.nr_tasks++;
    update_timeslice(current);

    schedule();
  }
```

Ordonnanceur $O(1)$ (Linux 2.6)

Les tâches **RUNNING** sont rangées dans une `struct rq`

```c
struct task_struct {
  struct list_head list;
  int timeslice;
  int prio;
};

struct prio_array {
  struct list_head array[140];
  unsigned int nr_tasks;
};

struct rq {
  struct prio_array *active;
  struct prio_array *expired;
};
```

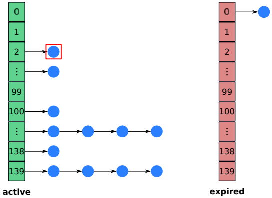



```c
void schedule(void)
{
  int i;
  struct task_struct *next = &idle_task,
                     *prev = current;

  /* si STOPPED, retirer de prio_array */
  if (!(prev->state & TASK_RUNNING)) {
    list_del(prev->list);
    rq->active.nr_tasks--;
  }

  /* Permute active/expired si active vide */
  if (!rq->active.nr_tasks)
        swap_pointers(rq->active, rq->expired);

  /* Choisir la tache la plus prioritaire */
  for (i = 0; i < 140; i++) {
    if (!list_empty(rq->active.array[i])) {
      next = list_first_entry(rq->active.array[i],
                              struct task_struct, list);
      break;
    }
  }

  context_switch(prev, next);
}
```

Ordonnanceur $O(1)$ (Linux 2.6)

Les tâches **RUNNING** sont rangées dans une `struct rq`

```c
struct task_struct {
  struct list_head list;
  int timeslice;
  int prio;
};

struct prio_array {
  struct list_head array[140];
  unsigned int nr_tasks;
};

struct rq {
  struct prio_array *active;
  struct prio_array *expired;
};
```

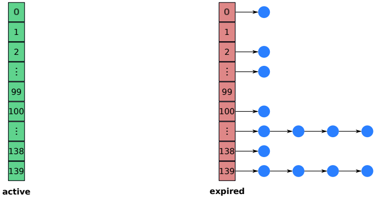



```c
void schedule(void)
{
  int i;
  struct task_struct *next = &idle_task,
                     *prev = current;

  /* si STOPPED, retirer de prio_array */
  if (!(prev->state & TASK_RUNNING)) {
    list_del(prev->list);
    rq->active.nr_tasks--;
  }

  /* Permute active/expired si active vide */
  if (!rq->active.nr_tasks)
        swap_pointers(rq->active, rq->expired);

  /* Choisir la tache la plus prioritaire */
  for (i = 0; i < 140; i++) {
    if (!list_empty(rq->active.array[i])) {
      next = list_first_entry(rq->active.array[i],
                              struct task_struct, list);
      break;
    }
  }

  context_switch(prev, next);
}
```

Ordonnanceur $O(1)$ (Linux 2.6)

Les tâches **RUNNING** sont rangées dans une `struct rq`

```c
struct task_struct {
  struct list_head list;
  int timeslice;
  int prio;
};

struct prio_array {
  struct list_head array[140];
  unsigned int nr_tasks;
};

struct rq {
  struct prio_array *active;
  struct prio_array *expired;
};
```

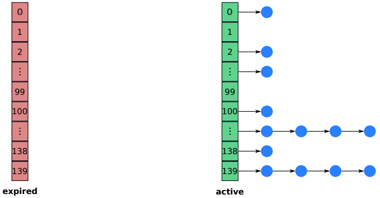



```c
void schedule(void)
{
  int i;
  struct task_struct *next = &idle_task,
                     *prev = current;

  /* si STOPPED, retirer de prio_array */
  if (!(prev->state & TASK_RUNNING)) {
    list_del(prev->list);
    rq->active.nr_tasks--;
  }

  /* Permute active/expired si active vide */
  if (!rq->active.nr_tasks)
        swap_pointers(rq->active, rq->expired);

  /* Choisir la tache la plus prioritaire */
  for (i = 0; i < 140; i++) {
    if (!list_empty(rq->active.array[i])) {
      next = list_first_entry(rq->active.array[i],
                              struct task_struct, list);
      break;
    }
  }

  context_switch(prev, next);
}
```

Ordonnanceur $O(1)$ (Linux 2.6)

Les tâches **RUNNING** sont rangées dans une `struct rq`

```c
struct task_struct {
  struct list_head list;
  int timeslice;
  int prio;
};

struct prio_array {
  struct list_head array[140];
  unsigned int nr_tasks;
};

struct rq {
  struct prio_array *active;
  struct prio_array *expired;
};
```




```c
void schedule(void)
{
  int i;
  struct task_struct *next = &idle_task,
                     *prev = current;

  /* si STOPPED, retirer de prio_array */
  if (!(prev->state & TASK_RUNNING)) {
    list_del(prev->list);
    rq->active.nr_tasks--;
  }

  /* Permute active/expired si active vide */
  if (!rq->active.nr_tasks)
        swap_pointers(rq->active, rq->expired);

  /* Choisir la tache la plus prioritaire */
  for (i = 0; i < 140; i++) {
    if (!list_empty(rq->active.array[i])) {
      next = list_first_entry(rq->active.array[i],
                              struct task_struct, list);
      break;
    }
  }

  context_switch(prev, next);
}
```

Ordonnanceur $O(1)$ (Linux 2.6)

Avantages:

+ *Complexité*: $O(1)$
+ *Interactivité*: basée sur la priorité

Inconvénients:

– *Code*: heuristiques du calcul de la priorité complexes
  $\Rightarrow$ potentielles erreurs

Ordonnanceur $O(1)$ (Linux 2.6)

Avantages:

+ *Complexité*: $O(1)$
+ *Interactivité*: basée sur la priorité

Inconvénients:

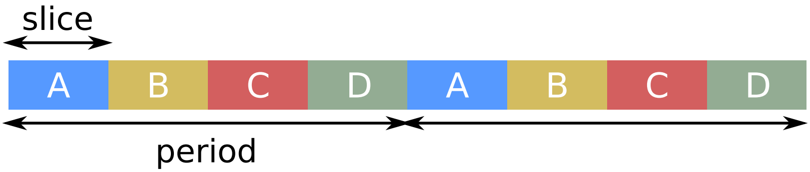
– *Code*: heuristiques du calcul de la priorité complexes
  $\Rightarrow$ potentielles erreurs

Solution choisie

Concevoir un nouvel algorithme d'ordonnancement exempté de cette complexité

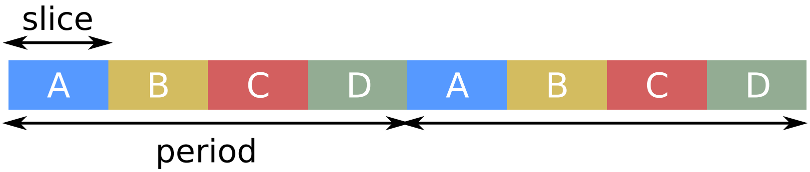
Completely Fair Scheduler (depuis Linux 2.6.23)

Objectif: Répartir le temps CPU équitablement entre les tâches



Completely Fair Scheduler (depuis Linux 2.6.23)

Objectif: Répartir le temps CPU équitablement entre les tâches

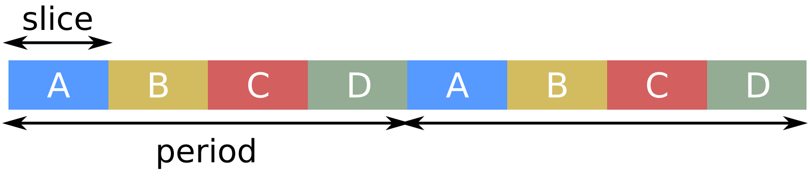


$$period = \begin{cases} |tasks| \times 0.75\ ms & \text{si } |tasks| > 8 \\ 6\ ms & \text{sinon} \end{cases} \tag{1}$$

$$slice = \frac{period}{|tasks|} \tag{2}$$

Completely Fair Scheduler (depuis Linux 2.6.23)

Objectif: Répartir le temps CPU équitablement entre les tâches



$$period = \begin{cases} |tasks| \times 0.75\ ms & \text{si } |tasks| > 8 \\ 6\ ms & \text{sinon} \end{cases} \qquad (1)$$

$$slice = \frac{period}{|tasks|} \qquad (2)$$

Petit point structure de données !

Arbre rouge-noir

Arbre binaire de recherche coloré auto-équilibré:

```
enum color {BLACK, RED};

struct node {
        int key;
        enum color color;
        struct node *left;
        struct node *right;
};
```

Propriétés:

1. Un noeud est **rouge** ou **noir**
2. La racine est **noire**
3. Les feuilles sont **noires** et valent `NULL` (**x**)
4. Si N est **rouge**, N→left et N→right sont **noirs**
5. Tout chemin de la racine à une feuille contient le même nombre de noeuds **noirs**

Arbre rouge-noir

Arbre binaire de recherche coloré auto-équilibré:

C

```
enum color {BLACK, RED};

struct node {
        int key;
        enum color color;
        struct node *left;
        struct node *right;
};
```

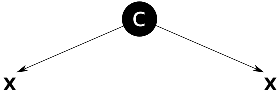
Propriétés:

1. Un noeud est **rouge** ou **noir**
2. La racine est **noire**
3. Les feuilles sont **noires** et valent `NULL` (**x**)
4. Si N est **rouge**, N→left et N→right sont **noirs**
5. Tout chemin de la racine à une feuille contient le même nombre de noeuds **noirs**

Arbre rouge-noir

Arbre binaire de recherche coloré auto-équilibré:

```
enum color {BLACK, RED};

struct node {
        int key;
        enum color color;
        struct node *left;
        struct node *right;
};
```



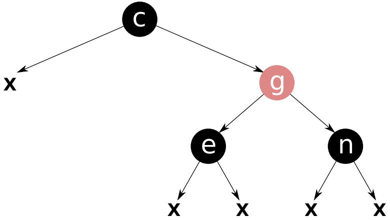
Propriétés:

1. Un noeud est **rouge** ou **noir**
2. La racine est **noire**
3. Les feuilles sont **noires** et valent `NULL` (**x**)
4. Si N est **rouge**, N→left et N→right sont **noirs**
5. Tout chemin de la racine à une feuille contient le même nombre de noeuds **noirs**

Arbre rouge-noir

Arbre binaire de recherche coloré auto-équilibré:

```
enum color {BLACK, RED};

struct node {
        int key;
        enum color color;
        struct node *left;
        struct node *right;
};
```



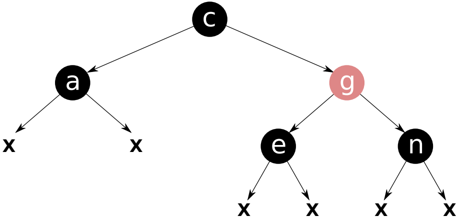
Propriétés:

1. Un noeud est **rouge** ou **noir**
2. La racine est **noire**
3. Les feuilles sont **noires** et valent `NULL` (**x**)
4. Si N est **rouge**, N→left et N→right sont **noirs**
5. Tout chemin de la racine à une feuille contient le même nombre de noeuds **noirs**

Arbre rouge-noir

Arbre binaire de recherche coloré auto-équilibré:

```c
enum color {BLACK, RED};

struct node {
        int key;
        enum color color;
        struct node *left;
        struct node *right;
};
```

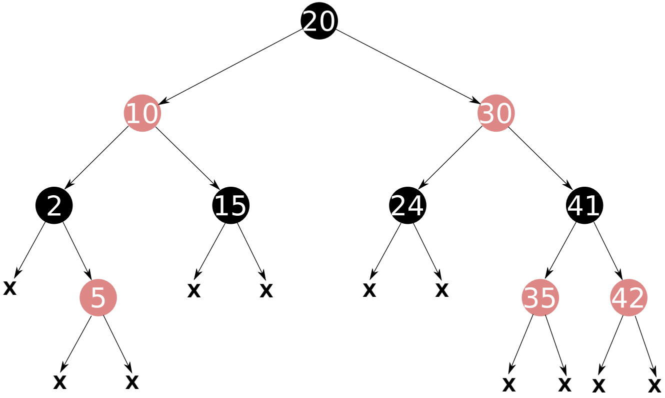


Propriétés:

1. Un noeud est **rouge** ou **noir**
2. La racine est **noire**
3. Les feuilles sont **noires** et valent `NULL` (**x**)
4. Si N est **rouge**, N→left et N→right sont **noirs**
5. Tout chemin de la racine à une feuille contient le même nombre de noeuds **noirs**

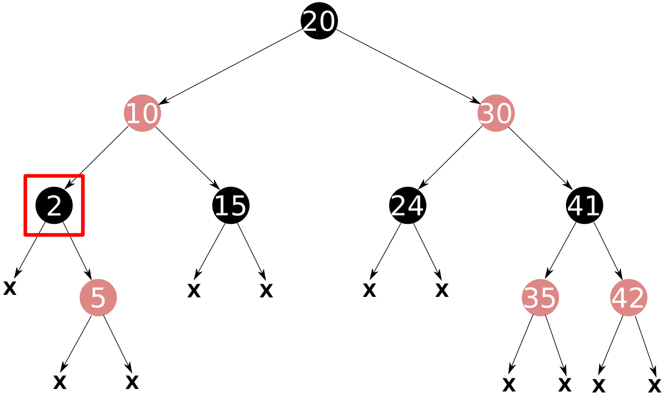
Arbre rouge-noir

1. Suppression
2. Insertion
3. Réparation

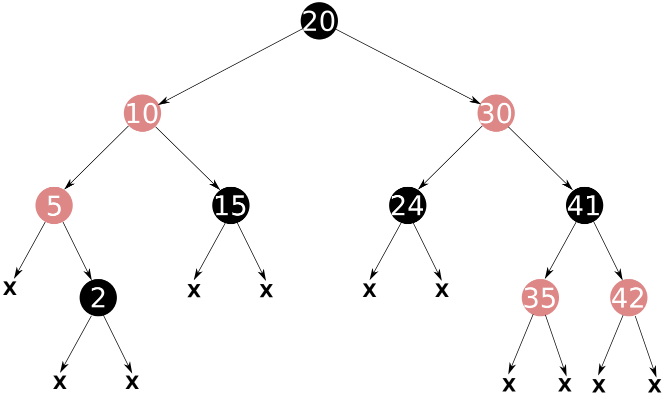


Arbre rouge-noir

1. Suppression
2. Insertion
3. Réparation

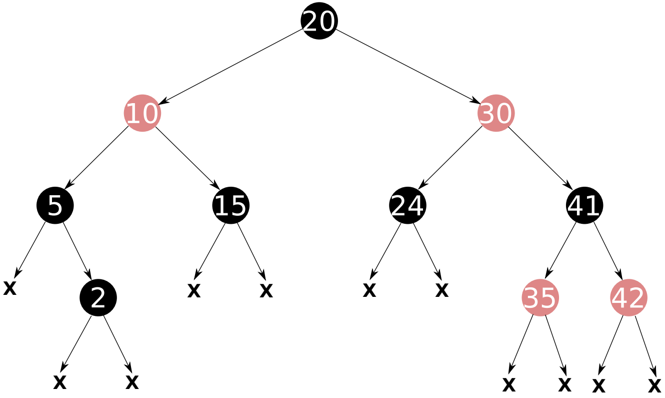


1. Suppression
2. Insertion
3. Réparation



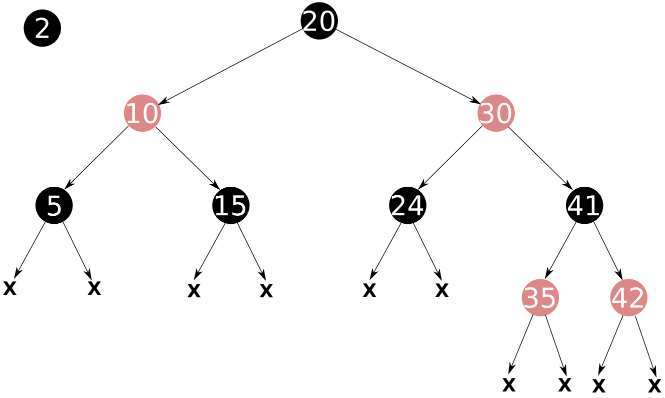
Arbre rouge-noir

1. Suppression
2. Insertion
3. Réparation



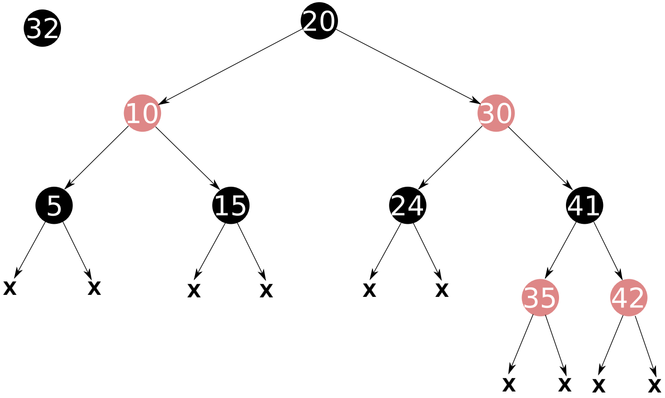
Arbre rouge-noir

1. Suppression
2. Insertion
3. Réparation



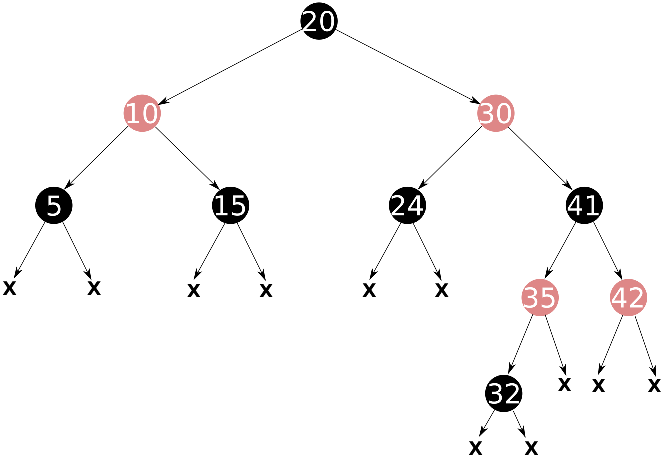
Arbre rouge-noir

1. Suppression
2. Insertion
3. Réparation



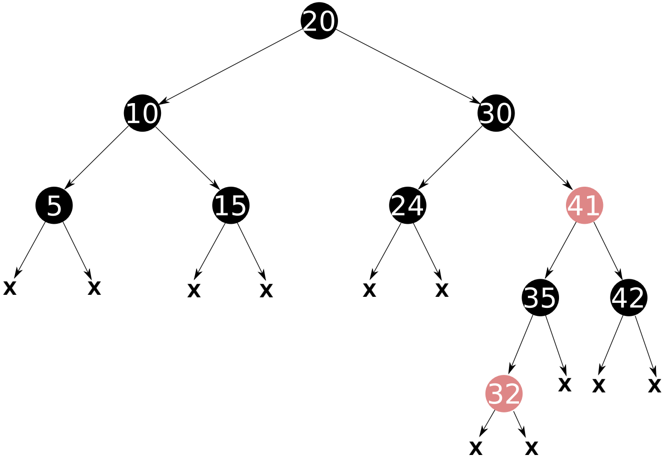
Arbre rouge-noir

1. Suppression
2. Insertion
3. Réparation



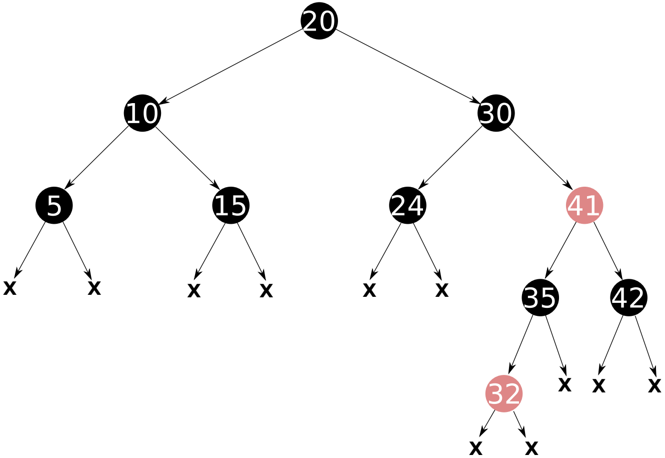
Arbre rouge-noir

1. Suppression
2. Insertion
3. Réparation



Arbre rouge-noir

1. Suppression
2. Insertion
3. Réparation



Complexité

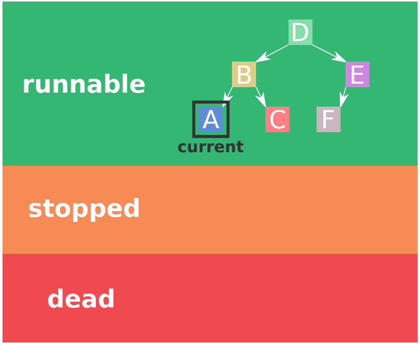
- Recherche: $O(logN)$
- Insertion: $O(logN)$
- Suppression: $O(logN)$

Completely Fair Scheduler (depuis Linux 2.6.23)

Les tâches **RUNNING** sont dans un arbre rouge-noir trié par `vruntime`

```c
struct task_struct {
  unsigned long vruntime;
  u64 exec_start, slice;
  unsigned long load;
  struct rb_node run_node;
};
```

```c
struct rq {
  struct rb_root tasks;
  unsigned long load;
  unsigned int nr_running;
};
```

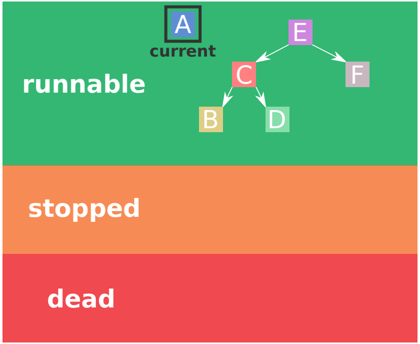



```c
void schedule(void)
{
  struct task_struct *prev = current;
  struct task_struct *next = &idle_task;

  /* Retirer la tache du rb_tree */
  dequeue_task(rq, prev);

  /* Si RUNNING, reinserer dans le rb_tree */
  if (prev->state & TASK_RUNNING)
    enqueue_task(rq, prev);

  /* Si rq vide, devenir idle */
  if (!rq->nr_running)
    goto ctx_switch;

  /* Sinon, choisir la tete de rq */
  next = rb_entry(rb_first(&rq->tasks),
                  struct task_struct, run_node);

ctx_switch:
  context_switch(prev, next);
}
```

Les tâches **RUNNING** sont dans un arbre rouge-noir trié par `vruntime`

```
struct task_struct {
  unsigned long vruntime;
  u64 exec_start, slice;
  unsigned long load;
  struct rb_node run_node;
};
```

```
struct rq {
  struct rb_root tasks;
  unsigned long load;
  unsigned int nr_running;
};
```



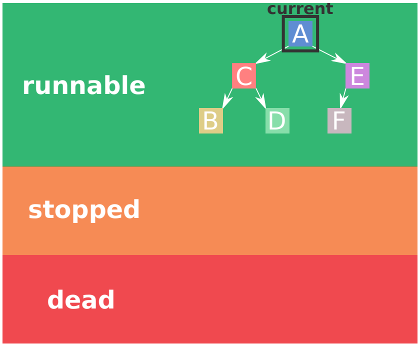

```
void schedule(void)
{
  struct task_struct *prev = current;
  struct task_struct *next = &idle_task;

  /* Retirer la tache du rb_tree */
  dequeue_task(rq, prev);

  /* Si RUNNING, reinserer dans le rb_tree */
  if (prev->state & TASK_RUNNING)
    enqueue_task(rq, prev);

  /* Si rq vide, devenir idle */
  if (!rq->nr_running)
    goto ctx_switch;

  /* Sinon, choisir la tete de rq */
  next = rb_entry(rb_first(&rq->tasks),
                  struct task_struct, run_node);

ctx_switch:
  context_switch(prev, next);
}
```

Completely Fair Scheduler (depuis Linux 2.6.23)

Les tâches **RUNNING** sont dans un arbre rouge-noir trié par `vruntime`

```
struct task_struct {
  unsigned long vruntime;
  u64 exec_start, slice;
  unsigned long load;
  struct rb_node run_node;
};
```

```
struct rq {
  struct rb_root tasks;
  unsigned long load;
  unsigned int nr_running;
};
```



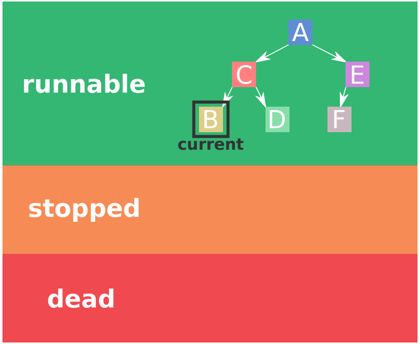

```
void schedule(void)
{
  struct task_struct *prev = current;
  struct task_struct *next = &idle_task;

  /* Retirer la tache du rb_tree */
  dequeue_task(rq, prev);

  /* Si RUNNING, reinserer dans le rb_tree */
  if (prev->state & TASK_RUNNING)
    enqueue_task(rq, prev);

  /* Si rq vide, devenir idle */
  if (!rq->nr_running)
    goto ctx_switch;

  /* Sinon, choisir la tete de rq */
  next = rb_entry(rb_first(&rq->tasks),
                  struct task_struct, run_node);

ctx_switch:
  context_switch(prev, next);
}
```

Completely Fair Scheduler (depuis Linux 2.6.23)

Les tâches **RUNNING** sont dans un arbre rouge-noir trié par `vruntime`

```
struct task_struct {
  unsigned long vruntime;
  u64 exec_start, slice;
  unsigned long load;
  struct rb_node run_node;
};
```

```
struct rq {
  struct rb_root tasks;
  unsigned long load;
  unsigned int nr_running;
};
```




```
void schedule(void)
{
  struct task_struct *prev = current;
  struct task_struct *next = &idle_task;

  /* Retirer la tache du rb_tree */
  dequeue_task(rq, prev);

  /* Si RUNNING, reinserer dans le rb_tree */
  if (prev->state & TASK_RUNNING)
    enqueue_task(rq, prev);

  /* Si rq vide, devenir idle */
  if (!rq->nr_running)
    goto ctx_switch;

  /* Sinon, choisir la tete de rq */
  next = rb_entry(rb_first(&rq->tasks),
                  struct task_struct, run_node);

ctx_switch:
  context_switch(prev, next);
}
```

Completely Fair Scheduler (depuis Linux 2.6.23)

`vruntime` = temps passé sur le CPU pondéré par la valeur de `nice`

Completely Fair Scheduler (depuis Linux 2.6.23)

`vruntime` = temps passé sur le CPU pondéré par la valeur de `nice`
Mis à jour à chaque `tick`

```
void tick(void) {
  u64 now, delta;

  /* Temps depuis dernier tick */
  now = get_clock();
  delta = now - current->exec_start;
  current->exec_start = now;

  /*  Mise a jour vruntime */
  update_vruntime(current, delta);

  /* Si tache n'est plus prioritaire, schedule() */
  if (need_resched(rq, curr))
    schedule();
}
```

Completely Fair Scheduler (depuis Linux 2.6.23)

Avantages:

+ *Complexité*: $O(\log n)$
+ *Équité*: basée sur le `vruntime`
+ *Interactivité*: basée sur le `vruntime`

Inconvénients:

− *Code*: heuristiques devenues complexes avec le temps

− *Taille du code*: $\approx$ 23.000 lignes de code
  $\Rightarrow$ compréhension et maintenance difficiles

Completely Fair Scheduler (depuis Linux 2.6.23)

Avantages:

+ *Complexité*: $O(\log n)$
+ *Équité*: basée sur le `vruntime`
+ *Interactivité*: basée sur le `vruntime`

Inconvénients:

- *Code*: heuristiques devenues complexes avec le temps
- *Taille du code*: $\approx$ 23.000 lignes de code
  $\Rightarrow$ compréhension et maintenance difficiles

Et le multicœur dans tout ça ?

Équilibrage de charge

Depuis $O(1)$, chaque CPU possède sa `struct rq`, ordonnancée localement, protégée par un `spinlock`

La charge entre les CPUs est équilibrée par différents moyens:

- équilibrage périodique
- équilibrage si un CPU devient *idle*
- placement au `fork()/exec()`
- placement au `wake_up()`

Définition de la charge (CFS)

La charge (*load*) d'un processus est une proportion d'utilisation du CPU

$$load = weight \; \frac{time_{runnable}}{slice}$$

avec:

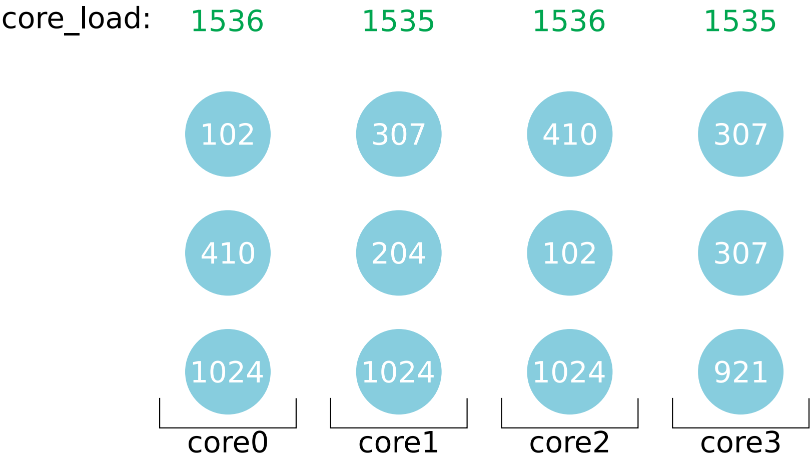
- *weight*: une correspondance de `nice` vers $[0, 1024[$
- $time_{runnable}$: le temps passé en **RUNNING**
- *slice*: le temps alloué au processus

Pour chaque coeur:

$$core\_load = \sum_{p \in rq.tasks} p.load$$

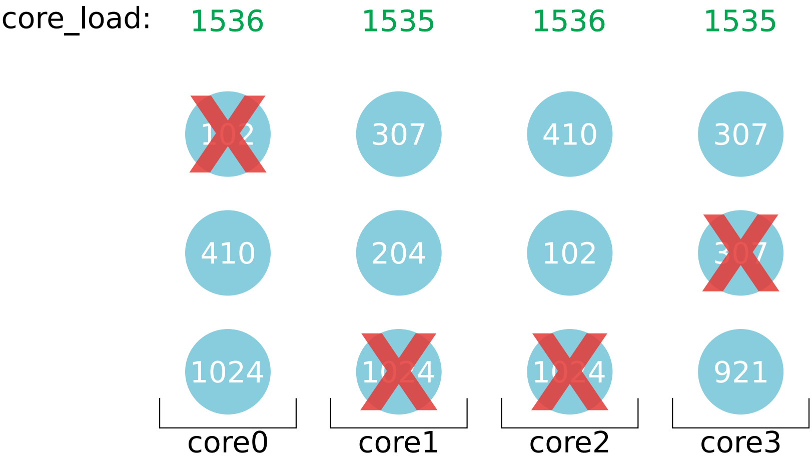
Équilibrage de charge périodique (CFS)

Périodiquement, chaque coeur tente de voler le coeur le plus chargé



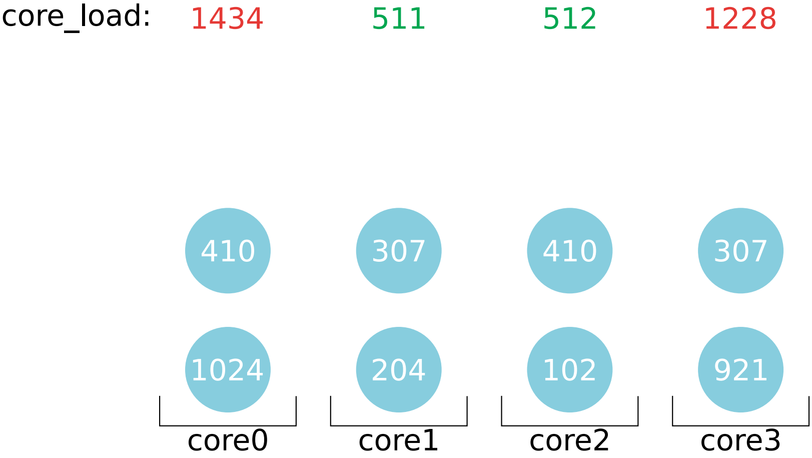
Équilibrage de charge périodique (CFS)

Périodiquement, chaque coeur tente de voler le coeur le plus chargé



Équilibrage de charge périodique (CFS)

Périodiquement, chaque coeur tente de voler le coeur le plus chargé



Équilibrage de charge périodique (CFS)

Périodiquement, chaque coeur tente de voler le coeur le plus chargé

core_load:	1434	511	512	1228
	410	307	410	307
	1024	204	102	921
	core0	core1	core2	core3

Équilibrage de charge périodique (CFS)

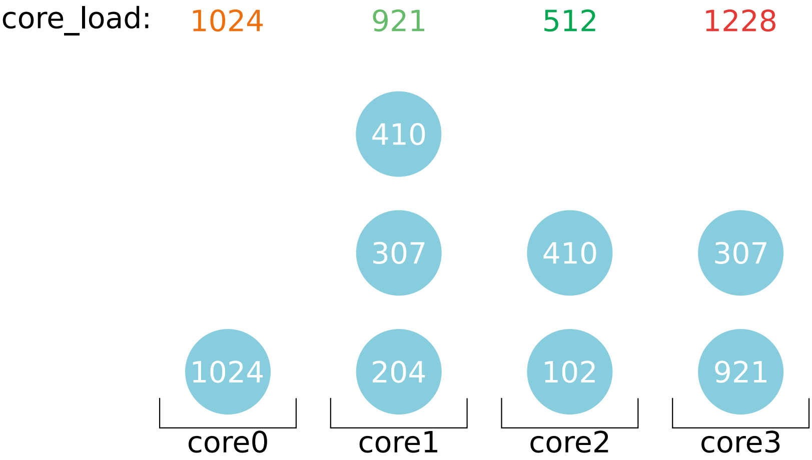
Périodiquement, chaque coeur tente de voler le coeur le plus chargé

core_load:	1434	511	512	1228

	core0	core1	core2	core3
	410	307	410	307
	1024	204	102	921

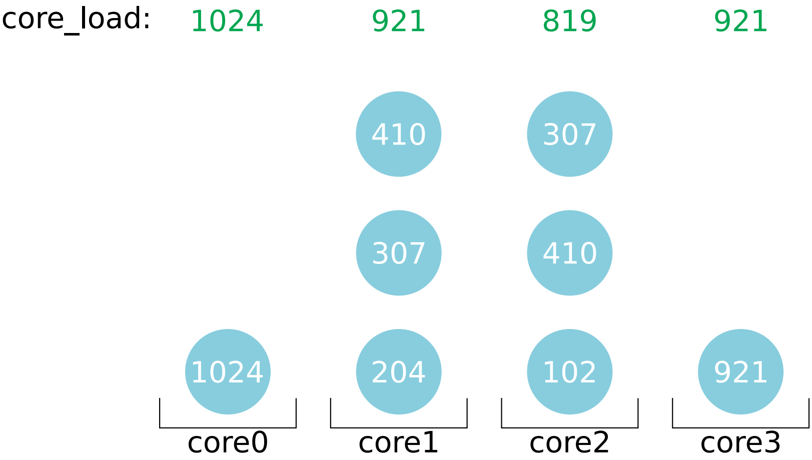
Équilibrage de charge périodique (CFS)

Périodiquement, chaque coeur tente de voler le coeur le plus chargé



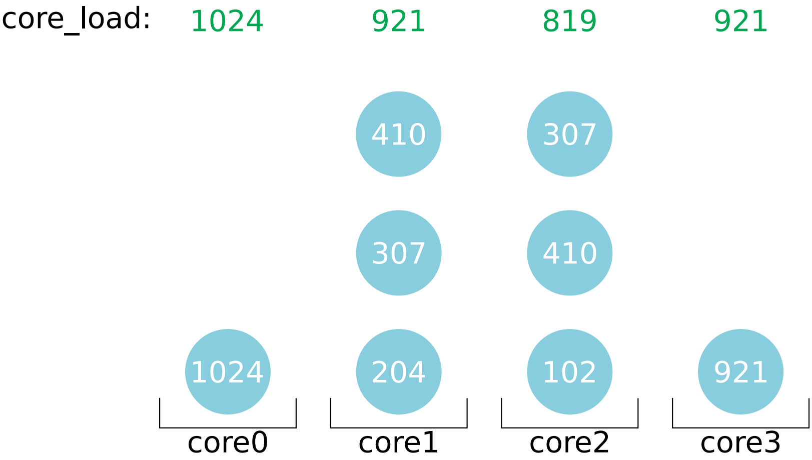
Équilibrage de charge périodique (CFS)

Périodiquement, chaque coeur tente de voler le coeur le plus chargé



Équilibrage de charge périodique (CFS)

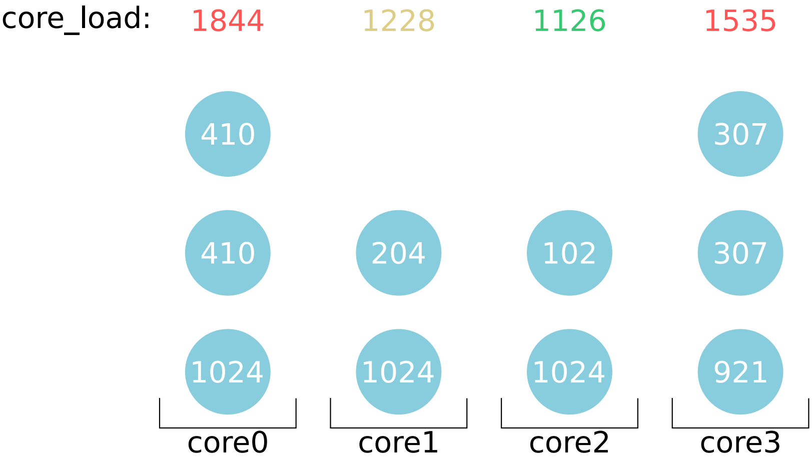
Périodiquement, chaque coeur tente de voler le coeur le plus chargé



La période vaut `NR_CPUS` millisecondes (4 *ms* ici)

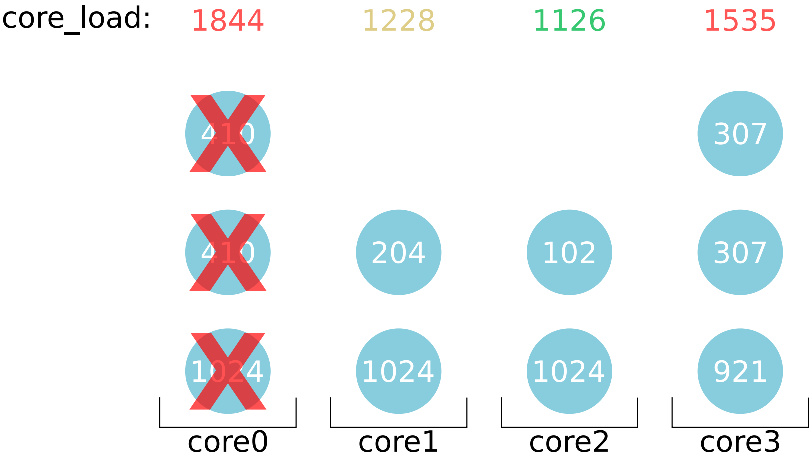
Équilibrage de charge sur coeur *idle* (CFS)

Si un coeur devient *idle*, il tente un équilibrage avec le coeur le plus chargé



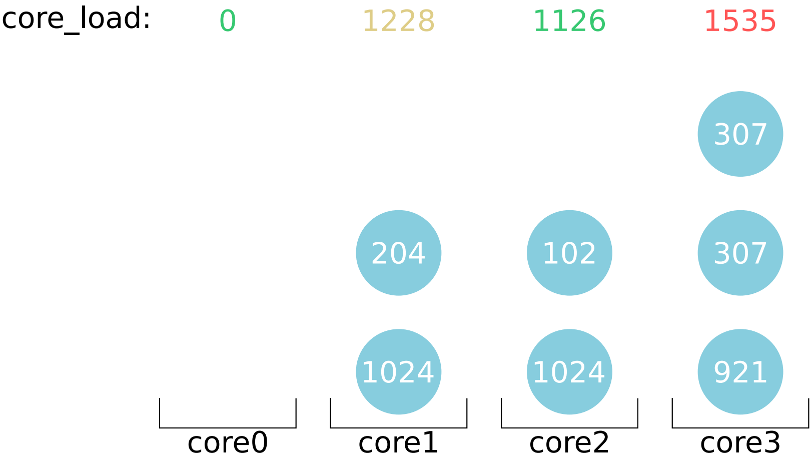
Équilibrage de charge sur coeur *idle* (CFS)

Si un coeur devient *idle*, il tente un équilibrage avec le coeur le plus chargé



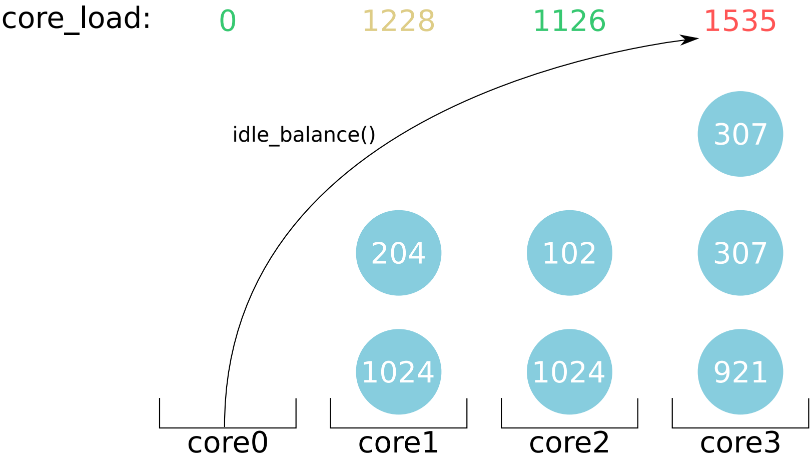
Équilibrage de charge sur coeur *idle* (CFS)

Si un coeur devient *idle*, il tente un équilibrage avec le coeur le plus chargé



Équilibrage de charge sur coeur *idle* (CFS)

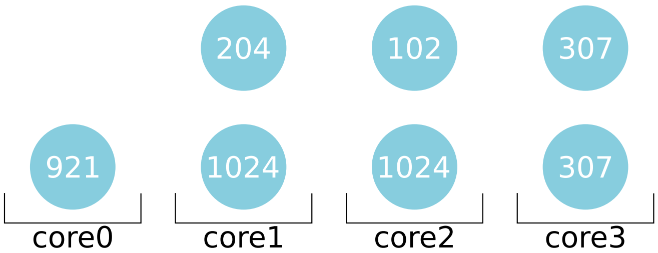
Si un coeur devient *idle*, il tente un équilibrage avec le coeur le plus chargé



Équilibrage de charge sur coeur *idle* (CFS)

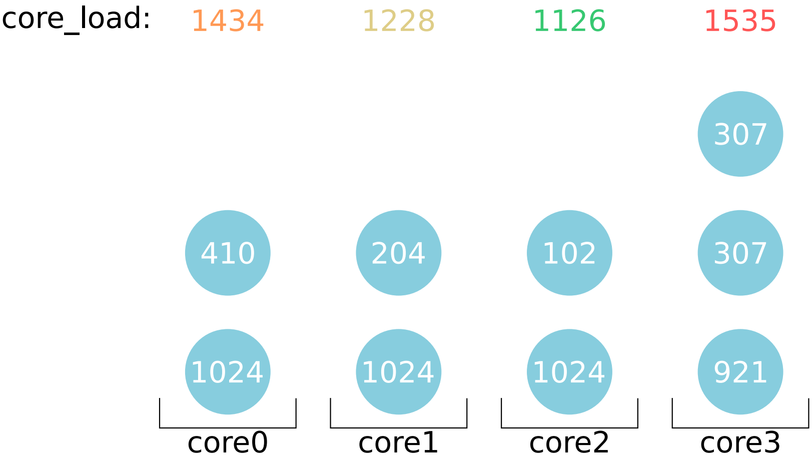
Si un coeur devient *idle*, il tente un équilibrage avec le coeur le plus chargé

core_load: 921 1228 1126 614



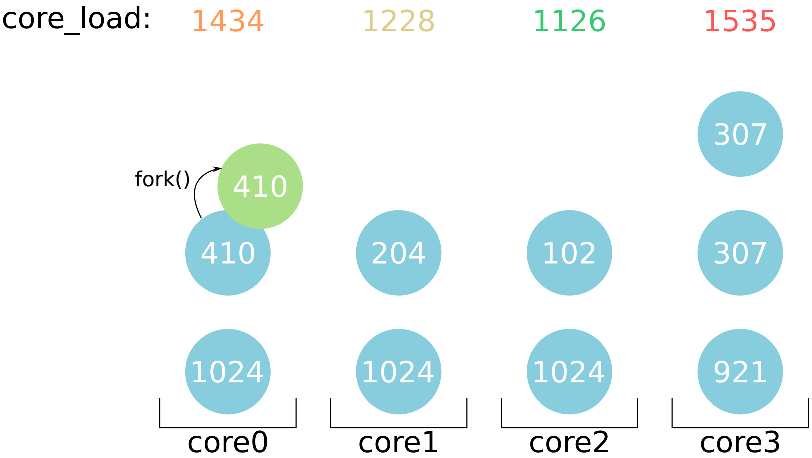
Placement au `fork()/exec()` (CFS)

La tâche est placée sur le coeur le moins chargé



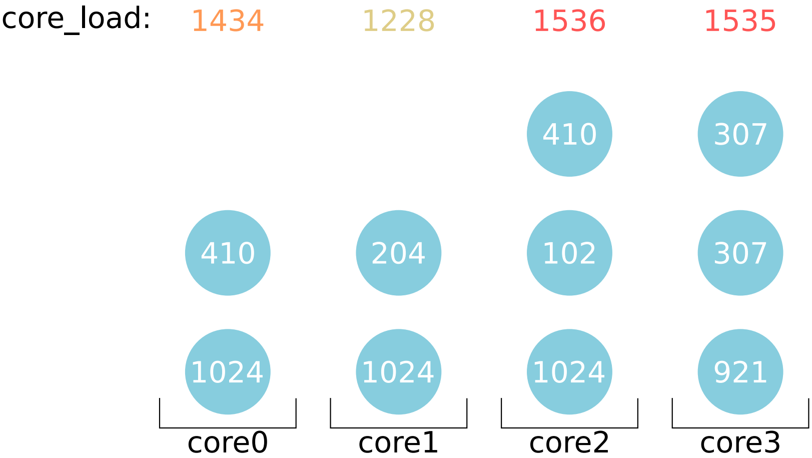
Placement au `fork()/exec()` (CFS)

La tâche est placée sur le coeur le moins chargé



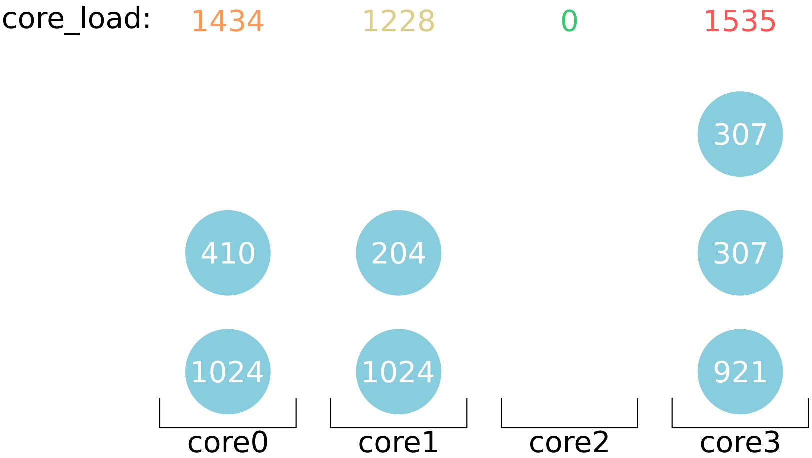
Placement au `fork()`/`exec()` (CFS)

La tâche est placée sur le coeur le moins chargé



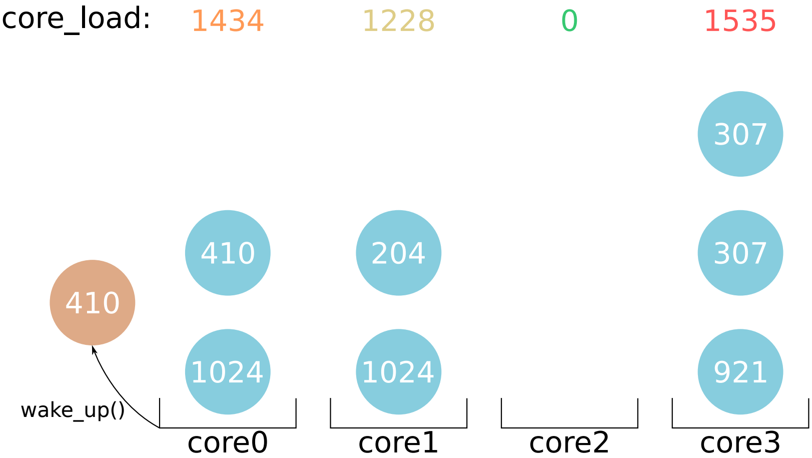
Placement au `wake_up()` (CFS)

La tâche est placée sur un coeur *idle* si possible



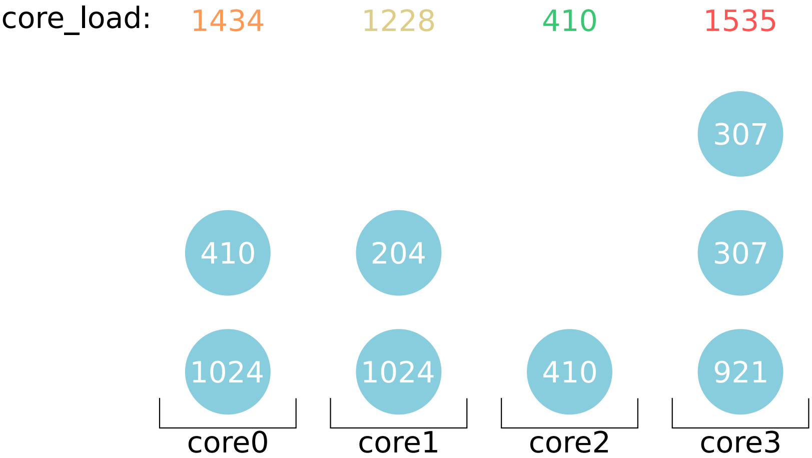
Placement au `wake_up()` (CFS)

La tâche est placée sur un coeur *idle* si possible



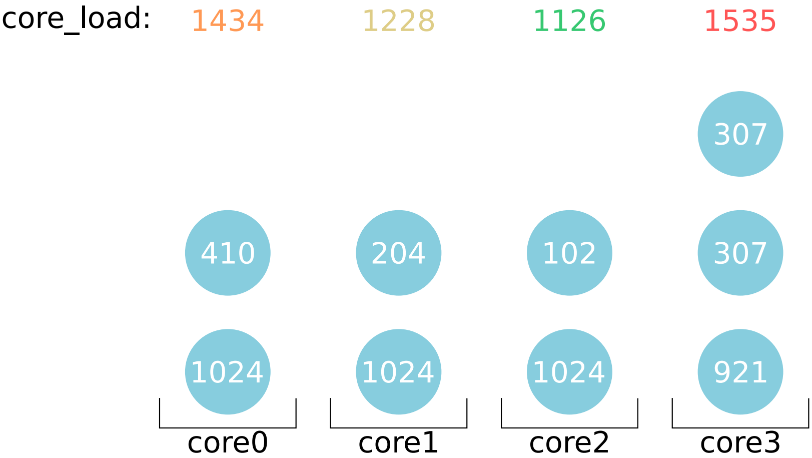
Placement au `wake_up()` (CFS)

La tâche est placée sur un coeur *idle* si possible



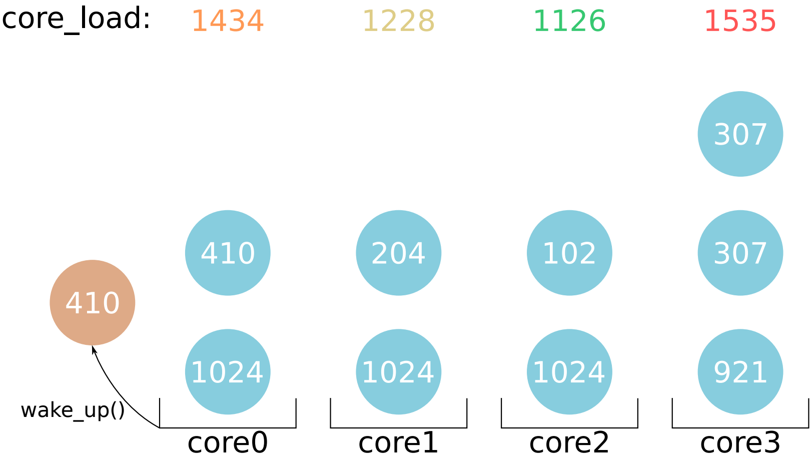
Placement au `wake_up()` (CFS)

La tâche est placée sur un coeur *idle* si possible, ~~localement sinon~~



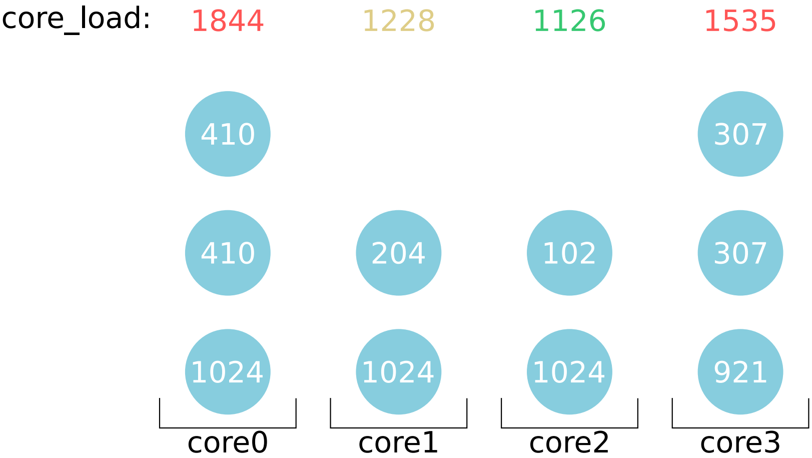
Placement au `wake_up()` (CFS)

La tâche est placée sur un coeur *idle* si possible, ~~localement sinon~~



Placement au `wake_up()` (CFS)

La tâche est placée sur un coeur *idle* si possible, ~~localement sinon~~



Topologie matérielle

Threads hardware (HWT)

Topologie matérielle

Threads hardware (HWT)

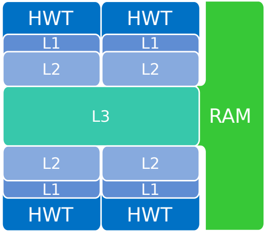
- homogène ? hétérogène ?
- consommation d'énergie

Topologie matérielle

Threads hardware (HWT)

- homogène ? hétérogène ?
- consommation d'énergie

Caches



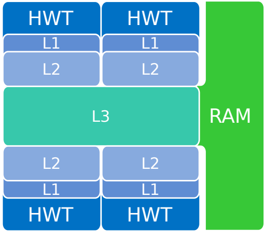
Topologie matérielle

Threads hardware (HWT)

- homogène ? hétérogène ?
- consommation d'énergie

Caches

- localité des données
- contention/thrashing



Topologie matérielle

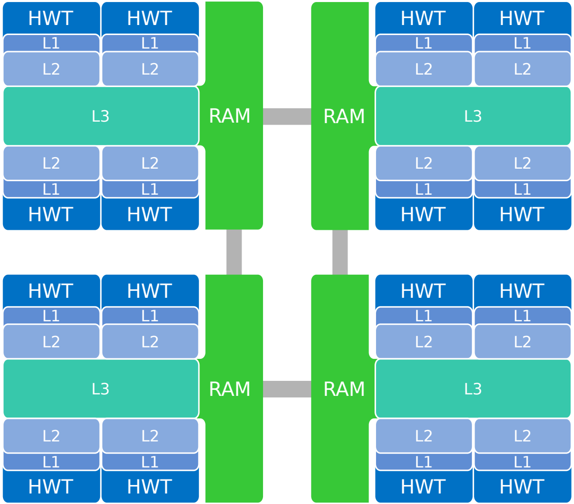
Threads hardware (HWT)

- homogène ? hétérogène ?
- consommation d'énergie

Caches

- localité des données
- contention/thrashing

NUMA



Topologie matérielle

Threads hardware (HWT)

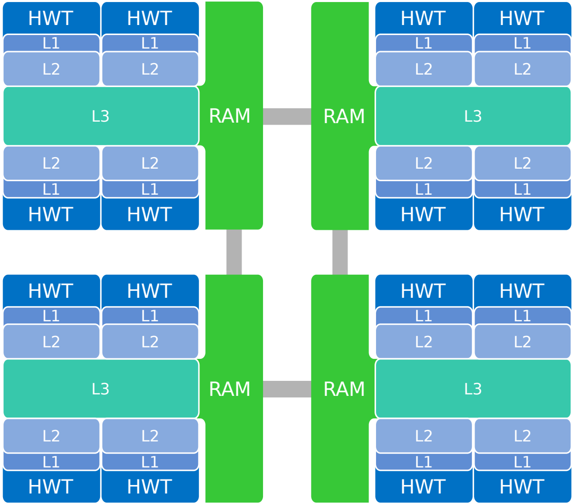
- homogène ? hétérogène ?
- consommation d'énergie

Caches

- localité des données
- contention/thrashing

NUMA

- latence des accès distants
- contention sur l'interconnect



Topologie matérielle

Threads hardware (HWT)

- homogène ? hétérogène ?
- consommation d'énergie

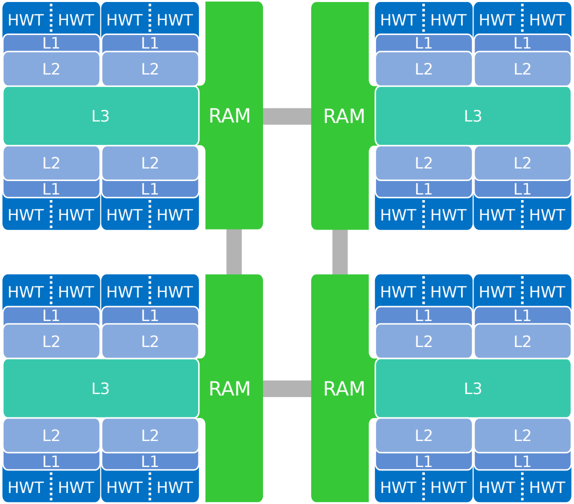
Caches

- localité des données
- contention/thrashing

NUMA

- latence des accès distants
- contention sur l'interconnect

SMT



Topologie matérielle

Threads hardware (HWT)

- homogène ? hétérogène ?
- consommation d'énergie

Caches

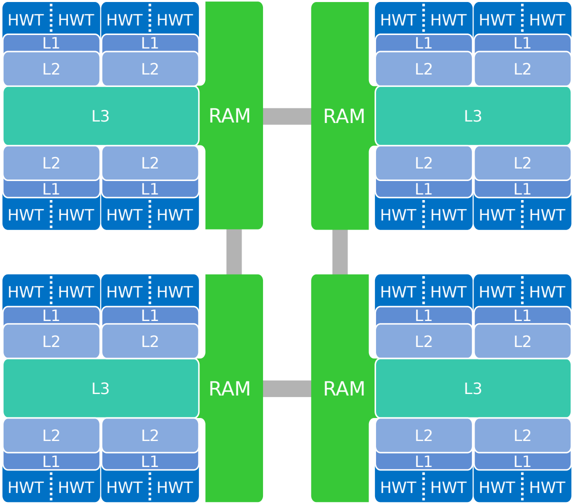
- localité des données
- contention/thrashing

NUMA

- latence des accès distants
- contention sur l'interconnect

SMT

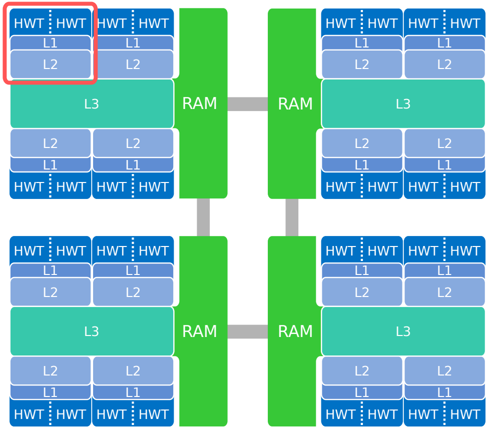
- localité (caches L1/L2)
- partage ALU, FPU, ...



Scheduling domains dans Linux

Chaque CPU construit une hiérarchie de domaines selon la topologie.
Domaines du CPU supérieur gauche:

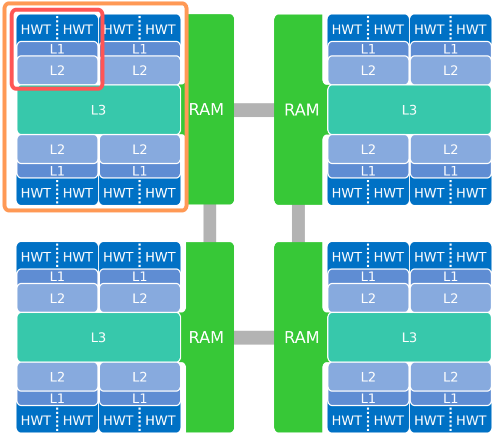
- SMT: partage de ressources de calcul
- LLC: partage d'un cache
- NUMA 1-hop: accès mémoires non uniformes
- NUMA 2-hop: accès mémoires non uniformes



Scheduling domains dans Linux

Chaque CPU construit une hiérarchie de domaines selon la topologie.
Domaines du CPU supérieur gauche:

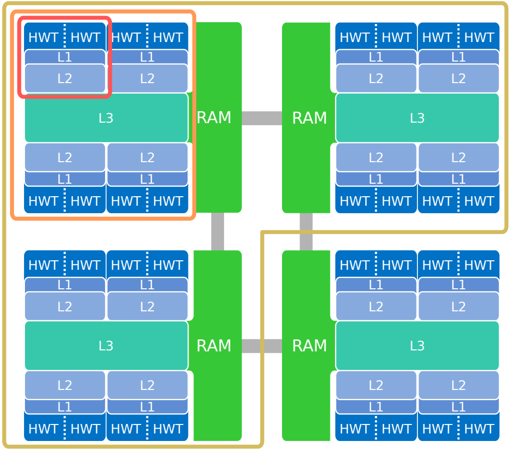
- SMT: partage de ressources de calcul
- LLC: partage d'un cache
- NUMA 1-hop: accès mémoires non uniformes
- NUMA 2-hop: accès mémoires non uniformes



Scheduling domains dans Linux

Chaque CPU construit une hiérarchie de domaines selon la topologie.
Domaines du CPU supérieur gauche:

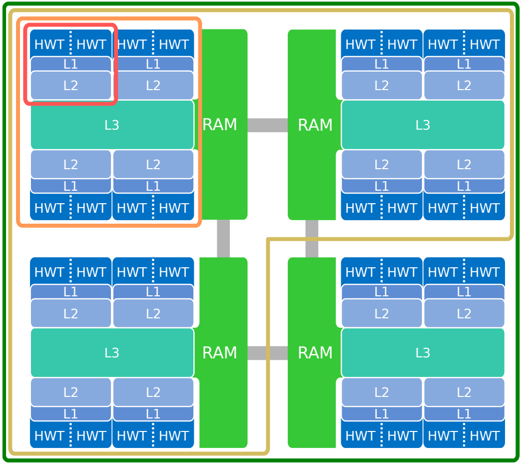
- SMT: partage de ressources de calcul
- LLC: partage d'un cache
- NUMA 1-hop: accès mémoires non uniformes
- NUMA 2-hop: accès mémoires non uniformes



Scheduling domains dans Linux

Chaque CPU construit une hiérarchie de domaines selon la topologie.
Domaines du CPU supérieur gauche:

- SMT: partage de ressources de calcul
- LLC: partage d'un cache
- NUMA 1-hop: accès mémoires non uniformes
- NUMA 2-hop: accès mémoires non uniformes



Scheduling domains dans Linux (2)

Le domaine régit la distance de placement selon l'évènement, ainsi que la période d'équilibrage de charge ($nr\_cpus(domain)\ ms$).

Domaine	fork	exec	wakeup	load balancing
SMT	✔	✔	✔	✔
LLC	✔	✔	✔	✔
NUMA 1-hop	✔	✔	✘	✔
NUMA 2-hop	✘	✘	✘	✔

Sujets connexes

Ordonnancement:

- de VM, conteneurs
- multi-machines

Recherche:

- Ce que j'ai fait en thèse (cf. soutenance)
- Faire une thèse