Compte rendu Ordonnanceurs Nanvix

Erwan Poncin / Maxime Bossant / Maxence Maury

Lien du git:

https://github.com/erwanponcin/AS.git

Ordonnancements par priorité

Code à retrouver dans la branche *priority* du repository git.

```
/* Choose a process to run next. */
next = IDLE;
for (p = FIRST_PROC; p <= LAST_PROC; p++){
   /* Skip non-ready process. */
   if (p->state != PROC_READY)
      continue;
   if ((next == IDLE || p->nice < next->nice || (p->nice == next->nice && p->counter > next->counter)))
      next = p;
   else
      p->counter++;
}
```

Le champ nice d'un processus représente sa priorité user indexé de `-40` à `20` .

Pour choisir le processus nous regardons quel processus du premier au dernier a une priorité `p->nice` la plus élevée. Pour les processus avec une priorité égale nous choisissons en round robin.

Principe de la boucle de yield

On parcours tous les processus READY.

Si le statut du processus choisi (`next`) est IDLE, on choisit le processus en cours (`p`) afin que `next` soit valide.

Si le processus étudié (`p`) a une priorité `nice` inférieure, donc plus prioritaire que celle de `next` , le processus choisi devient le processus `p` .

Si les processus `p` et `next` on la même priorité `nice` , on regarde lequel a le `counter` le plus élevé : celui ayant le `counter` le plus élevé sera le processus choisi.

Si aucune de ces conditions ne sont vérifiées, on incrémente seulement le counter du processus `p` .

Nous aurions aimé aller plus loin notamment en prenant un rapport entre le champ `priority` et le champ `nice` afin de pouvoir gérer l'impact de `nice` sur la priorité système mais par faute de temps, nous avons préférez continuer.

Afin de tester l'efficacité de notre code, nous avons créé un test avec des print, pour pouvoir vérifier que les priorités fortes s'exécutent avant les priorités faibles.

Lottery scheduling

Code à retrouver dans la branche *lottery* du repository git.

Voici le pseudo code pour notre algorithme de scheduling lottery :

```
nb_ticket = 0;
ticket_tab = tab[100]; // 100 tickets maximum, limité par la memoire kernel

for   p in process
   si p.nice < 20 et nb_ticket > 100
   /* ici on veut un nice plus petit que 20 et pas plus de 100 tickets */
      nice = un nb entre 0 et 20
   for i de nb_ticket à (nb_ticket + p.nice) :
      ticket_tab[i] = p.PID;
   end for
   nb_ticket += p.nice;
end for

winner = rand de 0 a nb_ticket

for
   if p.PID == ticket_tab[winner]
      next = p;
end for

switch process
```

Les tickets sont représentés par le nombre de cases contenant le PID du process dans le tableau ticket_tab. par exemple si le processus de PID = 6 a 5 tickets, ticket_tab[0] à ticket_tab[4] seront égal a 6.

- **Attribution des tickets** : Chaque processus prêt se voit attribuer un nombre de tickets, proportionnel à son "nice" ne dépassant pas 20. quand 100 tickets ont été attribués (limite taille du tableau dans le kernel) on donne 0 tickets.
- **Sélection d'un gagnant** : Un processus est choisi au hasard en fonction du nombre total de tickets attribués.
- **Changement de processus** : Le processus gagnant devient le prochain à s'exécuter, et le contrôle est transféré à lui.

  Le dernier test ne passe pas, il y a un core dumped.

Multiple Queues

Code à retrouver sur la branche *multiple_queues* du repository git

Voici le pseudo code pour notre algorithme de scheduling multiple queues :

```
for p in process
      si p→time > 20 ms
         queue1.add(p)
      else
         queue2.add(p)
   end for
   for p in queue1
      priority
   end for
   if queue1 vide
      for p in queue2
```

```
        FIFO
    end for
```

L'idée est d'avoir 2 queues. On place les processus ayant un temps d'exécution total supérieur à 20ms dans la `queue1` . Tous les autres processus sont placés dans le `queue2` .

Une fois les 2 queues remplies, on traite chaque processus présent dans la `queue1` en utilisant priority scheduling. Lorsque la `queue1` est vide, on traite la `queue2` en utilisant FIFO scheduling.