IT-201 TD2-correction

baptiste.coye

February 2020

Implémentation de base

2-a

```
void wait()
{
  /* on s'ajoute au debut de la sleepq */
  current->p_next = sleepq;
  sleepq = current;
  /* on se marque en dodo sur l'evenement */
  current->p_stat = STATUS_BLOCKED;
  /* on rend la main */
  switch();
}

void wakeup()
{
  while (sleepq) {
    /* on enleve de la sleepq */
    struct proc * myproc = sleepq;
    sleepq = myproc->p_next;
    myproc->p_next = NULL;
    /* on marque reveill  */
    myproc->p_stat = STATUS_READY;
    /* on met sur la runq */
    myproc->p_next = runq;
    runq = myproc;
  }
  sleepq = NULL;
}
```

Dans ce cas là le status n'est pas strictement nécessaire, il est équivalent à être sur la Run Queue ou la Sleep Queue. Cependant il est utile pour le switch() pour savoir si on veut s'endormir ou si l'ordonnanceur peut nous rendre la main

si aucuns autre processus ne la demande.

2-b

Une interruption du disque peut techniquement reveiller un processus qui attend une lecture disque. Cela fait qu'un wakeup peut être appelé au milieu d'un wait, par exemple entre les modifications de la SleepQ.
Sur le même principe, un wakeup peut également être appelé au milieu d'un autre wakeup si une interruption matérielle (irq) arrive quand un autre processus fait un wakeup (Exemple : Ecriture dans un tube sur lequel un processus attend en lecture). Pour prévenir cela on protége wait et wakeup des irqs.

```
void wait()
{
  irq_disable();
  current->p_stat = STATUS_BLOCKED;
  /* on s'ajoute au debut de la sleepq */
  current->p_next = sleepq;
  sleepq = current;
  irq_enable();
  /* on rend la main */
  switch();
}

void wakeup()
{
  irq_disable();
  while (sleepq) {
    /* on enleve de la sleepq */
    struct proc * myproc = sleepq;
    sleepq = myproc->p_next;
    myproc->p_next = NULL;
    /* on marque reveill  */
    myproc->p_stat = STATUS_READY;
    /* on met sur la runq */
    myproc->p_next = runq;
    runq = myproc;
  }
  sleepq = NULL;
  irq_enable();
}
```

Notes: On peut remarquer qu'il vaut mieux garder les irqs autour du changement de status car un switch alors que le status est BLOCKED sans être dans la file peut rendre la main sans jamais la reprendre si l'ordonnanceur verifie le status quand il va piocher dans la RunQ.

Pour un exemple d'implémentation réelle, se référer au code de linux qui permet l'attente ou le wakeup.

- wait_for_completion : https://git.kernel.org/pub/scm/linux/kernel/git/wtarreau/linux-2.4.git/tree/kernel/sched.c?h=v2.4.37n768
- try_to_wake_up : https://git.kernel.org/pub/scm/linux/kernel/git/wtarreau/linux-2.4.git/tree/kernel/sched.c?h=v2.4.37n349

2-c

Dans le cas ou un processus veuille s'endormir et qu'avant qu'il rende la main une interruption réveil tous les processus en attente, il va se placer dans la file d'attente puis être reveillé immediatement avant switch et rendre la main alors que ce n'était pas nécessaire. Dans ce cas là il perd du temps inutilement. Cela ne pose aucuns problèmes mais entraîne une perte de performance. Pour éviter que cela se produise, on pourrait changer l'implémentation du switch afin de vérifier que le process rendant la main a bien un STATUS_BLOCKED

2-d

Dans le cas où notre machine était monoprocesseur on ne pouvait avoir qu'un processus dans wait ou wakeup interrompu par un wakeup d'une interruption matérielle. En multiprocesseurs, on a alors plusieurs processus dans wait ou wakeup en même temps (même sans interruptions). Dans ce cas là il faut protéger la SleepQ et la RunQ grâce à des verrous.
On utilise normalement des verrou en attente active (spin_lock dans linux), pas un verrou qui endort le processus qui attend. Il faut tout de même désactiver les interruptions avant de prendre le verrou, de cette façon nous pouvons opérer sur la RunQ et la SleepQ en sécurité puis relacher le verrou avant de réactiver les interruptions.

Réception de paquets réseau

3-a

Il est nécessaire dans la fonction wakeup d'incrémenter la valeur de received_count mais il faut alors désactiver les interruptions pour empêcher d'avoir des accès concurents à cette variable.

```
void wakeup()
{
  irq_disable();

  received_count++;

  while (sleepq) {
    struct proc * myproc = sleepq;
    sleepq = myproc->p_next;
    myproc->p_stat = STATUS_READY;
    myproc->p_next = runq;
    runq = myproc;
  }
  irq_enable();
}
```

Dans le wait, il faut vérifier le received_count avant de dormir. Il est possible d'autoriser les interruptions avant le consume car on est sûr qu'il y aura au moins un paquet disponible à consommer. De plus on gagne en performance car cela évite de bloquer la machine si la fonction consume n'est pas assez rapide.

```
void wait()
{
  while (1) {
    irq_disable();
    if (received_count) {
      /* c'est bon, il y a un paquet pour nous, on marque qu'on le prend */
      received_count--;
      irq_enable();
      /* maintenant on peut le prendre en securit  */
      consume();
      return;
    }
    current->p_stat = STATUS_BLOCKED;
    current->p_next = sleepq;
    sleepq = current;
    irq_enable();
    switch();
    /* quelqu'un nous a reveill  , mais verifions qu'il reste
    bien un paquet pour nous */
  }
}
```

3b- Il suffit de prendre en compte la longueur du message reçu avec length. Reviens à modifier WakeUp en incrémentant received_count de length. Pour

le wait c'est un peu plus complexe, il faut consommer l'ensemble des octets disponibles sinon on dort

```
void wait(int length)
{
  while (length) {
    irq_disable();
    /* tant qu'on peut lire, on lit */
    if (receive_count) {
      int consumed = min(received_count, length);
      /* on lit ce qu'on peut */
      received_count -= consumed;
      irq_enable();
      consume(consumed);
      length -= consumed;
    } else {
      /* pu rien a lire, il faut dormir */
      current->p_stat = STATUS_BLOCKED;
      current->p_next = sleepq;
      sleepq = current;
      irq_enable();
      switch();
      /* on nous a reveill , on s'est peut-etre fait piquer
      notre paquet par un autre process entre temps */
    }
  }
```

3c- Si plusieurs processus se réveil en même temps pour consommer les paquets on risque le cas où un processus récupère l'intégralité de façon solitaire. Ce qui nous aurait fait réveiller l'ensemble des autres processus pour rien. Pour pallier à ce problème on peut ajouter dans la structure Proc une quantité d'octets p_waitlenght qui précisera le nombre d'octet qu'un processus peut récupérer en une fois. Il faut donc légèrement modifier l'implémentation de wait et wakeup pour prendre ce champ en compte.

3d- Dans le cas où un processus fait un poll ou un select alors il pourrait vouloir être réveiller sans consommer de paquets directement. Ainsi on ajoute un Flag dans la structure qui permettra en parcourant la Sleepq de réveiller tous les processus qui ont ce flag même si length ¡ 0. Le problème est que de cette façon on doit parcourir toute la sleepq pour les réveiller, du coup on peut dédoubler la SleepQ et ainsi en avoir une correspondant aux Recv et une aux Poll/Select.

3e- Afin de prendre en compte la connexion on déplace le received_count dans la structure conn et on ajout la connexion dans la structure proc en p_waitconn. On a alors Wait qui regarde le received_count de la conn et s'il doit dormir il met la connexion dans son p_waitconn. Le WakeUp lui parcourt la liste et

ne regarde que les processus correspondant à cette connexion. (Techniquement il faudrait aller chercher non plus au début mais un peu partout dedans mais compliqué à implémenter avec une sleepq simplement chainée).

3f- Encore une fois la technique ici est dédoubler le nombre de sleepq en la mettant dans la connexion. En pratique on a des sleepq pour à peut près tout, un par socket, un par lecture disque, un par tube...

Extensions

Petit Laïus pour comprendre comment peuvent être utilisés les évènements:
En programmation on ne peut pas toujours se baser sur une programmation séquentielle. Une méthode de programmation est de définir le programme par ses réactions aux différents événements pouvant se produire, un événement se traduit fréquemment par un changement de variable. Un bon exemple pour voir cela reste le jeu vidéo, en fonction de la zone (Proximité avec un ennemi, marcher sur un objet etc...) le programme va lancer une réaction correspondante (Activer l'aggressivité de l'ennemi ou ajouter l'objet à l'inventaire).

4a- la structure de l'event doit contenir une file d'attente sur laquelle les processus peuvent se mettre avant de dormir ainsi qu'un verrou pour protéger cette dernière. On peut aussi ajouter la variable ou la fonction de callback qui va servir d'élément déclencheur à l'événement.

4b- Il nous faut une liste d'événements attendu par process au lieu d'un seul. Chaque maillon de la liste contient un maillon pour la fille d'attente correspondante, la condition qu'il attend. Ainsi on transforme wait en wait_any qui va tester chacun des event et consume en fonction de si ils sont réalisés ou pas. Attention ici c'est le wait qui va gérer la sleepq intégralement du coup on enlève pas de la sleepq dans le wakeup.

Si on veut attendre que tous les events se soit produits il suffit d'attendre que le 1er et le 2eme etc... se soit produits.

4c- Dans le but de gérer les priorité on insère en triant la runq et la sleepq. Cependant cela peut poser des problèmes, si un processus est peu prioritaire il pourrait rester en stall tout du long sans jamais être executé, pour pallier à ça on a plusieurs solutions, la première est de définir un temps qu'on consacre aux processus de faible priorité. On peut également décider de remonter les priorités des plus faibles à chaques tours ou ils ne sont pas executés, de cette façon au bout d'un moment ils repassent devant et on reboot à la priorité initiale une fois qu'ils sont passés. Cependant l'insertion dans la liste chainée de façon triée à un coût non négligeable.

L'ordonnanceur de l'OS évite se problème grâce aux TimeSlice, tout le monde à la main à un moment mais pas le même temps (Les priorités les plus faibles vont la garder beaucoup moins longtemps) ainsi on a pas besoin de trier la runq, on suit juste l'ordre d'arriée sur la sleepq.

4d- Dans le cas où on est en sommeil, ça dépend de qu'est ce qui nous y à mit. Certains sommeils ne sont pas interruptibles par les signaux par exemple

les lectures sur le disque car on risquerait de ne pas savoir où reprendre la lecture. Pour un recv certaines connexion ne vont pas recevoir le paquet tout de suite donc on ne peut pas attendre indefiniment avant de traiter le signal. Pour se renseigner sur le comportement du système en cas d'interruption d'un appel système par un signal vous pouvez regarder signal(7).

/section*Exemples plus concrets

5a- Pour résumer:

Ce n'est plus le handler d'interruption qui va faire wakeup mais l'appel système write, le wait quant à lui est fait par l'appel système read. Le write va devoir aussi lancer un wait si received_count dépasse la taille du tube il limite ainsi pour qu'aucun processus ne puisse trop écrire dans le tube.

5b- Cf Cours implémentation d'une semaphore

5c- Techniquement une variable de condition n'est qu'une file d'attente avec cond_signal qui réveille le premier de la file, cond_broadcast qui réveille tout le monde et cond_wait qui endor à la fin de la liste.

De cette façon une semaphore est similaire à une variable de condition où:

- La condition est "compteur" supérieur à 0
- le compteur est modifié en tenant le verrou
- sem_wait fait cond_wait si compteur = 0 puis décrémente le compteur
- sem_post incrémente le compteur puis fait cond_signal