Sakun WITHANAGE-PERERA 2SN R2

Projet Systèmes Concurrents

Objectif: Nous nous présentons face à un problème de type accès lecteur - rédacteur, le but est de s'assurer de la bonne partage des ressources.

Les modifications sont faites sur tube.c principalement dans les fonctions tubeLire et tubeEcrire (mis à part l'initialisation des conditions).

Il y a deux types d'implémentation possible pour chaque tubeLire et tubeEcrire. Cas bloquant ou non bloquant.

I -Mise en place générale

Les section critique dans notre situation est l'accès au tube pour l'écriture et/ou la lecture. Ceci nous amène à introduire une exclusion mutuelle pour s'assurer que l'écrivain ou le lecteur puisse avoir l'accès exclusif aux resources.

Nous introduisons alors dans le struc tube de nouvelles variables :

```
typedef struct _tube {
   uint8_t * donnees;   //< Pointeur sur la zone de données
   int taille;          //< Nombre d'octets présents dans le tube
   int indiceProchain ; //< Position de la prochaine insertion

   int nbEcrivains;
   int nbLecteurs;

   ExclusionMutuelle verrou;

   Condition pasPlein; //pour permettre l'acces à l'ecriture
   Condition pasVide;  // pour permettre l'accès à la lecture

} Tube;
```

- pasPlein signifie qu'on peut encore écrire dans le tube
- pasVide signifie qu'il y a des choses à lire dans le tube
- Le verrou est utilisé pour donner l'accès exclusif

Quelle que soit la situation (bloquant ou non bloquant) nous devons mettre en place l'exclusion mutuelle. On le fait grâce aux fonctions exclusionMutuelleEntrer et exclussionMutuelleSortie initialisé dans la fonction sys_tube.

II - Ecrivain

II.1 - Cas écrivain non bloquant

Ecrivain non bloquant correspond au cas où après avoir reçu l'accès exclusif, effectue une écriture dans le tube (cas normal). Si le tube est plein donc il ne peut rien écrire. Le processus entre dans la boucle et resort imédiatement et la valeur 0 est renvoyé.

On place exclusionMutuelleEntrer qui permet d'avoir l'accès exclusif en obtenant le verrou jusqu'à exclussionMutuelleSortie qui permet de rendre le verrou.

```
size_t tubeEcrire(Fichier * f, void * buffer, size_t nbOctets)
{
   Tube * tube;
   int n = 0;
   int nbOctetsEcrits = 0; // Le nombre d'octets déja écrits
   // Peut-on décemment écrire dans le tube ?
   if ((f == NULL) || (f->iNoeud == NULL) || (f->iNoeud->prive == NULL)) {
      return -EINVAL;
   }
   tube = f->iNoeud->prive;

   exclusionMutuelleEntrer(&(tube->verrou));
   printk_debug(DBG_KERNEL_TUBE, "in\n");
   do {

      n = MIN(nbOctets - nbOctetsEcrits, MANUX_TUBE_CAPACITE - tube->taille);

      // On ne doit pas aller écrire au delà du buffer
      n = MIN(n, (MANUX_TUBE_CAPACITE - tube->indiceProchain));

      memcpy(tube->donnees + tube->indiceProchain, buffer, n);

      tube->indiceProchain = (tube->indiceProchain + n) % MANUX_TUBE_CAPACITE;
      tube->taille += n;

      buffer += n;

      nbOctetsEcrits += n;
   } while (n > 0);

   exclusionMutuelleSortir(&(tube->verrou)); //on rend la main
   printk_debug(DBG_KERNEL_TUBE, "out\n");

   return nbOctetsEcrits;
}
```

II.2 - Cas écrivain bloquant

Ici, le cas bloquant correspond au cas où l'écrivain attends s'il n'y a pas assez de place dans le tube pour écrire et quitte l'écriture lorsqu'il a finit totalement d'écrire. (voir le code)

On va alors commencer par vérifier qu'il y a bien de la place dans le tube (pour chaque boucle dans le do) :

```
//si pas de place on on attends que de la place se libère.
//On préviens les potentiel lecteurs
while (tube->taille == MANUX_TUBE_CAPACITE){
 conditionDiffuser(&(tube->pasVide));
 conditionAttendre(&(tube->pasPlein), &(tube->verrou));
}
```

La condition diffuser permet ici de prévenir les potentiel lecteurs en attente que les données sur le tube attendent d'être lus et libéré pour ne plus bloquer l'écrivain en attente.

```
   nbOctetsEcrits += n;
} while (n > 0 || nbOctetsEcrits < nbOctets);
// condition de sortie est bloquant -> tant qu'on n'a pas tout écis on ne sort pas
```

Une fois la condition de sortie (bloquante) satisfaite, on relâche le verrou d'exclusion mutuelle et on signale que le tube contient des données afin de réveiller les lecteurs en attente :

```
conditionDiffuser(&(tube->pasVide));
exclusionMutuelleSortir(&(tube->verrou));
printk_debug(DBG_KERNEL_TUBE, "out\n");
```

III - Lecteurs

III.1 - Cas lecteurs non bloquant

Similaire au cas des écrivains non bloquants, correspond au cas où après avoir reçu l'accès exclusif, effectue une lecture dans le tube (cas normal). Si le tube est vide, il ne peut rien lire. Le processus entre alors dans la boucle et ressort immédiatement et la valeur 0 est renvoyé.

```c
//NON BLOQUANT
size_t tubeLire(Fichier * f, void * buffer, size_t nbOctets)
{
   Tube * tube;
   int n = 0;
   int nbOctetsLus = 0;
   int indicePremier;
   printk_debug(DBG_KERNEL_TUBE, "in\n");
   if ((f == NULL) || (f->iNoeud == NULL) || (f->iNoeud->prive == NULL)) {
      return -EINVAL;
   }
   tube = f->iNoeud->prive;
   exclusionMutuelleEntrer(&(tube->verrou));
   do {
      // A partir de quel octet peut-on lire ?
      indicePremier = (tube->indiceProchain + MANUX_TUBE_CAPACITE - tube->taille) %MANUX_TUBE_CAPACITE;
      n = MIN(nbOctets - nbOctetsLus, tube->taille);
      n = MIN(n, (MANUX_TUBE_CAPACITE - indicePremier));
      printk_debug(DBG_KERNEL_TUBE,"Je vais lire %d\n", n);
      memcpy(buffer, tube->donnees + indicePremier, n);
      indicePremier = (indicePremier + n) % MANUX_TUBE_CAPACITE;
      tube->taille -= n;
      buffer += n;
      nbOctetsLus += n;
   } while (n > 0);

   printk_debug(DBG_KERNEL_TUBE, "out\n");
   exclusionMutuelleSortir(&(tube->verrou));

   return nbOctetsLus;
}
```

III.2 - Cas lecteur bloquant

Le cas lecteur bloquant correspond au cas où s'il n'y a rien à lire dans le tube, on garde le lecteur en attente le temps qu'un écrivain vienne écrire.

Comme pour la l'écrivain on signale à la sortie pour actualiser l'état du tube.

```c
size_t tubeLire(Fichier * f, void * buffer, size_t nbOctets)
{
   Tube * tube;
   int n = 0;
   int nbOctetsLus = 0;
   int indicePremier;
   printk_debug(DBG_KERNEL_TUBE, "in\n");
   if ((f == NULL) || (f->iNoeud == NULL) || (f->iNoeud->prive == NULL)) {
      return -EINVAL;
   }
   tube = f->iNoeud->prive;
   exclusionMutuelleEntrer(&(tube->verrou));
   do {
      // Quelque chose à lire dans le tube ?
      while ((tube->taille) <= 0){
         conditionAttendre(&(tube->pasVide), &(tube->verrou));
      }
      // A partir de quel octet peut-on lire ?
      indicePremier = (tube->indiceProchain + MANUX_TUBE_CAPACITE - tube->taille) %MANUX_TUBE_CAPACITE;
      n = MIN(nbOctets - nbOctetsLus, tube->taille);
      n = MIN(n, (MANUX_TUBE_CAPACITE - indicePremier));
      printk_debug(DBG_KERNEL_TUBE,"Je vais lire %d\n", n);
      memcpy(buffer, tube->donnees + indicePremier, n);
      indicePremier = (indicePremier + n) % MANUX_TUBE_CAPACITE;
      tube->taille -= n;
      buffer += n;
      nbOctetsLus += n;
   } while (n > 0);

   printk_debug(DBG_KERNEL_TUBE, "out\n");
   conditionSignaler(&(tube->pasPlein));
   exclusionMutuelleSortir(&(tube->verrou));

   return nbOctetsLus;
}
```

Ici, on aurait pu faire le choix de considérer que le cas lecteur bloquant soit il n'y a rien à lire dans le tube et que le lecteur ait lu nbOctets dans le tube avant que la lecture ne se termine. Dans ce cas on aurait eu une condition de fin de boucle while différente (nbOctetsLus < nbOctets).