université de BORDEAUX

Collège Sciences et technologies

Année	2015-2016	**Type**	Corrigé partiel du DS
Master	Informatique		
Code UE	J1INAW11	**Épreuve**	Systèmes d'Exploitation
Date	12/11/2015	**Documents**	Non autorisés
Début	10h15	**Durée**	1h30

1 Question de cours (échauffement)

Cf cours.

2 Gestion de processus

Cf cours.

3 Synchronisation

Question 1 Le système d'impression tel que proposé peuvent aboutir à des incidents car certains accès concurrents risquent d'aboutir à l'écriture de valeurs incohérentes dans certaines variables. Par exemple, les deux premières lignes de la fonction `submit` posent problème : si deux processus testent simultanément la valeur de la variable `available` et qu'elle s'avère suffisante pour accueilir l'une ou l'autre des tâches d'impression, les deux processus vont simultanément décrémenter la variable, ce qui est un non-sens du point de vue sémantique (il n'y a peut-être pas assez de place pour accueillir les deux tâches d'impression) et ce qui peut en outre corrompre la valeur de la variable (la décrémentation n'est pas une opération atomique).

Question 2 On ajoute un mutex et une condition dans la structure `job_t` :

```
struct job_t {
  void *data;      /* adresse des données à imprimer */
  size_t size;     /* taille des données */
  unsigned status; /* vaut INIT ou COMPLETED */
};
```

Et voici les nouvelles variables globales et le nouveau code des deux fonctions :

```
unsigned available = MAX_SPOOL;
unsigned nb_jobs = 0;
mutex_t mutex;
cond_t cond_p; // pour bloquer les processus en attente de soumission
cpnd_t cond_d; // pour bloquer le démon d'impression

void submit(struct job_t *job)
{
  mutex_lock (&mutex);
  while (available < job->size)
    cond_wait (cond_p, mutex);

  available -= job->size;
  nb_jobs++;
  job->status = INIT;
  spool_insert(job);
  cond_signal (cond_d);
  mutex_unlock (&mutex);

  mutex_lock (&job->m);
  while (job->status != COMPLETED)
    cond_wait (&job->c, &job->m);
  mutex_unlock (&job->m);
}
```

```
 void printer_daemon()
{
  struct job_t *job;

  while (1) {
    mutex_lock (&mutex);
    while (nb_jobs == 0)
      cond_wait (&cond_d);

    job = spool_remove(); /* récupère une tâche et place les données
                             dans la mémoire interne de l'imprimante */
    nb_jobs--;
    available += job->size;
    cond_signal (cond_p);
    mutex_unlock (&mutex);

    SLEEP(DUREE_IMPRESSION); /* on simule la durée d'impression */

    mutex_lock (&job->m);
    job->status = COMPLETED;
    cond_signal (&job->c);
    mutex_unlock (&job->m);
  }
}
```

4 Appels système

Si les adresses des objets du noyau étaient transmises au niveau utilisateur pour servir d'identifiant, ces adresses ne pourraient pas être dé-référencées pour accéder directement à la mémoire du noyau, mais par contre il serait fastidieux pour le noyau de vérifier, lors de chaque appel système utilisant un tel identifiant, que cet identifiant n'a pas été modifié et pointe donc sur l'adresse de début d'un objet valide du noyau.

La bonne manière de faire est illustrée par les *descripteurs de fichiers* : le noyau maintient une table qui contient des pointeurs sur les objets alloués. L'identifiant renvoyé en espace utilisateur est l'indice occupé par le pointeur dans le tableau. Ainsi, lorsqu'un identifiant est passé en paramètre d'un appel système, le noyau peut aisément vérifier si un pointeur non-nul se trouve dans la case correspondante du tableau, et donc vérifier si l'identifiant est valide ou pas...