R. Namyst
LaBRI, Bureau 322

le 28/05/2004

Master d'informatique 1ère année — Examen
Programmation des Architectures Parallèles

Durée : 3h — Notes de cours autorisées

1 Communications en espace utilisateur

La plupart des interfaces de communication orientées "performance" (e.g. BIP, SBP ou VIA) court-circuitent le système d'exploitation lors des phases de communication, c'est-à-dire qu'elles n'utilisent pas d'appel système lors des opérations de communication proprement dites.

Question 1 Pour quelle raison cherche-t-on à éviter les appels systèmes ? Expliquez comment ces interfaces parviennent à piloter la carte réseau dans ces conditions ?

Question 2 Lors des opérations de communication, quelles précautions faut-il prendre avec les zones mémoire concernées pour éviter que le système n'évince certaines pages de la mémoire pendant un transfert ? Expliquez. Quelle est l'approche choisie par l'interface VIA de ce point de vue ?

Question 3 Pour utiliser le mécanisme de DMA disponible sur la plupart des cartes réseau, il faut adresser une requête au contrôleur en indiquant les *adresses physiques* des zones mémoire concernées par le transfert. Au niveau de l'application, c'est pourtant des adresses virtuelles qui sont fournies en paramètres des primitives de communication (e.g. `bip_send`).

Selon vous, à quel moment a lieu la conversion des adresses virtuelles en adresses physiques ? Comment faire cela en évitant, "la plupart du temps", le recours à un appel système ?

2 Intégration du multithreading et des communications

On dispose d'une bibliothèque (une sorte de PM2 simplifié) permettant à un ensemble de processus de communiquer à l'aide d'un mécanisme d'appel de procédure à distance "léger" (que nous appelerons simplement RPC).

Comme c'est le cas avec l'environnement PM2, chaque processus de la configuration contient un thread interne "démon" qui scrute le réseau et dont le travail est de répondre aux demandes de RPC. Dans cette version simplifiée, tous les processus exécutent le même programme (binaire identique) et les fonctions n'ont pas besoin d'être exportées pour pouvoir être invoquées à distance : l'appel d'une fonction à distance s'effectue en indiquant directement l'adresse de la fonction à exécuter et le numéro de processus au sein duquel il faut exécuter la fonction. Lorsqu'un thread démon reçoit une demande de RPC, il commence donc par extraire l'adresse de la fonction à exécuter puis l'appelle directement (sans créer de thread supplémentaire). Lorsque la fonction est terminée, le thread démon reprend son activité d'écoute du réseau.

La figure 1 présente le code d'un petit programme illustrant l'utilisation de la bibliothèque. Le processus 0 demande l'exécution de la fonction `f` au sein du processus 1, ce qui a pour effet d'afficher "Hello World !" sur la sortie standard de ce dernier. Les fonctions `pack`/`unpack` permettent de transmettre des paramètres (considérés comme une suite d'octets) à la fonction invoquée.

2.1 RPC basiques

Question 1 Comme exemple simple d'utilisation des RPC, on voudrait disposer d'une primitive `remote_write(unsigned node,` qui permette d'écrire un entier `value` à distance (à l'adresse `addr` au sein du processus `node`). Donnez le code de cette primitive ainsi que le code qu'il faut ajouter à la bibliothèque.

Question 2 Si l'on autorise l'utilisation de primitives de synchronisation (e.g. sémaphores) à l'intérieur des fonctions appelées lors des RPC, que risque-t-il de se passer ? Expliquez.

Question 3 Même question si l'on autorise des situations où la fonction exécutée par un RPC déclenche à son tour un RPC sur un autre nœud (sorte de récursivité des RPC). Expliquez sur un exemple.

```
void f()
{
  char msg[128];

  unpack(msg, 128);

  printf("%s\n", msg);
}
```

```
int main()
{
  // Initialise la bibliothèque
  init();

  if(my_rank() == 0) {
    char s[128];

    strcpy(s, "Hello World!");

    // appeler 'f' sur le processus '1'
    begin_rpc(1, f);
    pack(s, 128);
    end_rpc();

    // stoppe tous les processus
    halt();
  } else
    // attend qu'un processus exécute 'halt'
    wait_end();
}
```

FIG. 1 – Illustration de l'utilisation des RPC

Question 4 En supposant que la couche de communication sous-jacente est capable d'effectuer simultanément un envoi de message et une réception, expliquez comment on peut modifier l'implantation de la bibliothèque pour ne plus rencontrer ce problème dans une situation telle que celle décrite à la question 3.
NB : on pourra utiliser des threads supplémentaires, mais on évitera d'utiliser des threads créés de manière dynamique (i.e. à chaque RPC).

2.2 RPC "multithreadés"

La solution proposée à la question 4 ne résoud pas le problème posé à la question 2. Pour le résoudre, on décide que les fonctions exécutées lors d'un RPC ne le seront plus directement par le thread démon, mais seront en fait exécutées par un thread temporaire lancé exprès pour l'occasion. On supposera cette propriété pour toutes les questions suivantes.

Question 5 Lorsque le thread démon d'un processus vient de créer un thread temporaire pour exécuter une fonction de l'application, il doit attendre avant de scruter à nouveau le réseau à la recherche d'une nouvelle requête de RPC. Que doit-il attendre exactement ? Déduisez-en qu'il est nécessaire d'introduire une nouvelle primitive que les fonctions de l'application (exécutées par des RPC) devront désormais appeler durant leur exécution. Indiquez de quoi sera typiquement composé le code de cette primitive.

Peut-on désormais placer du code de synchronisation n'importe où dans une fonction exécutée par un RPC ?

2.3 RPC avec retour de résultats

On suppose l'existence de primitives `init`, `P` et `V` sur des objets de type `sem_t` :

```
typedef ... sem_t;
void init(sem_t *sem, unsigned value);
void P(sem_t *sem);
void V(sem_t *sem);
```

On souhaite implanter une primitive `begin_sync_rpc(int node, func_addr_t f, int *result)` qui déclenche l'exécution de la fonction `f` sur le nœud `node`, et qui attend le retour d'un résultat de type entier (pour simplifier) qui sera stocké à l'adresse `result`. Le processus appelant doit rester bloqué dans la fonction `end_sync_rpc()` jusqu'au retour du résultat.

Idée : il faut transmettre, en plus des paramètres de la fonction `f`, des données qui vont permettre, à la fin de `f`, de faire un RPC dans le sens inverse pour acheminer le résultat. Pour que ceci reste transparent, on pourra utiliser des fonctions intermédiaires que l'on pourra supposer intégrées à la bibliothèque. En particulier, c'est une bonne idée d'utiliser une fonction intermédiaire qui, avant d'appeler `f`, récupérera les données supplémentaires, etc.

Question 6 Donnez le code des fonctions `begin_sync_rpc` et `end_sync_rpc`, ainsi que celui des fonctions intermédiaires que vous aurez besoin d'introduire.
NB : On supposera l'existence d'un mécanisme permettant d'utiliser des variables globales "par thread" (appelées variables spécifiques dans Pthreads) de manière transparente : indiquez juste en commentaire lorsqu'une variable de votre programme est une variable globale aux threads...

Question 7 En utilisant ce mécanisme de RPC avec retour de résultat, implantez la primitive `int remote_read(unsigned node, int *addr)` qui renvoie le résultat de la lecture sur le processus `node` d'un entier à l'adresse `addr`.

2.4 Envoi de messages

On souhaite implanter un mécanisme d'envoi de messages par-dessus notre mécanisme de RPC. Les messages sont simplement une suite d'octets et sont contigus, et ils sont déposés dans des boîtes aux lettres. On définira donc un type `mailbox_t` auquel seront associées les opérations suivantes :

```
typedef ... mailbox_t;
void mailbox_create(mailbox_t *box);
void mailbox_put(mailbox_t *box, void *msg, unsigned len);
unsigned mailbox_get(mailbox_t *box, void *msg, unsigned max_len);
```

La primitive `mailbox_create` crée une boîte aux lettre localement qui restera toujours localisée sur le processus au sein duquel elle a été créée. Cependant, un objet de type `mailbox_t` doit pouvoir être transféré en paramètre d'un RPC (i.e. `pack(&box, sizeof(mailbox_t))`) et pouvoir être utilisable à distance. Ainsi, un processus ayant crée une boîte aux lettre pourra en communiquer "la référence" à un autre processus.

Cela signifie que les opérations `mailbox_put` (déposer un message dans la boîte) et `mailbox_get` (attendre un message et le récupérer) doivent être implantées au moyen de RPC. Cela signifie également que le type `mailbox_t` renferme simplement des informations permettant de retrouver la boîte aux lettre effective : l'adresse de la structure de données utilisée de manière interne, et le numéro de nœud sur lequel elle se trouve.

Question 8 Proposez une définition du type `mailbox_t` ainsi que des structures de données annexes dont vous aurez besoin.

Donnez le code de la fonction `mailbox_create`.

Question 9 Donnez le code des fonctions `mailbox_put` et `mailbox_get`.