Systèmes d'exploitation centralisés

1re année Informatique et Réseaux

juin 2017

Documents autorisés. Les exercices sont indépendants.

I Système de fichiers (5 points)

Un lien symbolique est un fichier particulier qui *contient* le nom d'un autre fichier. Cette information est rangée dans les blocs de données, comme des données « normales ». Par exemple considérons un répertoire `/toto` (inode I_1) contenant un lien symbolique `titi` (inode I_2) qui pointe vers `tutu`. Dans ce cas, on trouve dans les blocs de données de l'inode I_1 une entrée `titi` $\rightarrow I_2$, comme pour n'importe quel nom de fichier rangé dans ce répertoire. Par ailleurs on trouve dans les blocs de données de I_2 la chaîne "tutu". La résolution du nom `/toto/`*`titi`*`/xxx` passe par `/toto/`*`tutu`*`/xxx`. Noter que la chaîne `tutu` est un chemin d'accès, contenant éventuellement des `/` et dont certains des composants peuvent aussi être des liens symboliques.

1. Comment pourrait-on faire pour savoir si une inode correspond à un lien symbolique ou à un fichier régulier ?
2. Quelle fonction parmi celles présentées en cours suffit-il de modifier pour prendre en compte l'existence de liens symboliques ?
3. Donner le nouvel algorithme de cette fonction, en précisant quelles autres fonctions elle utilise.
4. Il est possible de construire des chemins circulaires, par exemple deux liens symboliques $a \rightarrow b$ et $b \rightarrow a$. Proposer une solution triviale (mais pas nécessairement optimale) pour détecter cette situation.

II Noyau – Coroutines (5 points)

Soit le squelette de code suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include "coroutines.h"

coroutine_t c1,c2,c3;

void code3 (void *unused)
{
    printf("3\n");
    cor_transferer(XXX,XXX);
    printf("6\n");
    cor_transferer(XXX,XXX);
    printf("9\n");
    cor_transferer(XXX,c1);
    printf("code1: fini\n");
}
void code2 (void *unused)
{
    c2 = cor_creer("C3", code3, NULL);
    printf("2\n");
    cor_transferer(XXX,XXX);
    printf("5\n");
    cor_transferer(XXX,XXX);
    printf("8\n");
    cor_transferer(XXX,XXX);
    printf("code2: fini\n");
}
void code1 (void *unused)
{
    printf("1\n");
    cor_transferer(XXX,XXX);
    printf("4\n");
    cor_transferer(XXX,XXX);
    printf("7\n");
    cor_transferer(XXX,XXX);
    printf("code1: fini\n");
}
int main()
{
    c1 = cor_creer("C1", code1, NULL);
    c2 = cor_creer("C2", code2, NULL);
    c3 = cor_creer("C3", NULL, NULL);
    cor_transferer(c1,c1);
    printf ("10 fini\n");
}
```

Remplacer les XXX pour obtenir l'affichage `1 2 3 4 5 6 7 8 9 10 fini`. Ne pas ajouter de nouvelles variables coroutines.

III Noyau – Processus (5 points)

Soit le code suivant qui coordonne un code de calcul et un code d'affichage via une variable buffer (le code de calcul attend que le buffer soit vide pour y mettre le résultat de son calcul ; le code d'affichage attend que le buffer contienne une valeur pour la traiter) :

```
#include <stdio.h>
#include "processus.h"
#include "scheduler.h"

#define N 10
int buffer = 0;

int somme(int n) {
    return (n <= 1) ? 1 : n + somme(n-1);
}

void calcul (void *unused) {
    for (int i = 1; i < N; i++) {
        int r = somme(i);
        while (buffer != 0) { proc_commuter(); }
        buffer = r;
    }
    buffer = -1;
}

void afficheur (void *unused) {
    int val;
    do {
        while (buffer == 0) { proc_commuter(); }
        val = buffer;
        buffer = 0;
        if (val != -1) printf ("%d\n", val);
    } while (val != -1);
}
int main() {
    processus_t p1, p2;
    sched_set_scheduler(&sched_aleatoire);
    proc_init();
    p1 = proc_activer("P1", calcul, NULL);
    p2 = proc_activer("P2", afficheur, NULL);
    proc_suspendre();
}
```

1. Qu'affiche ce programme ?
2. Se termine-t-il et si oui, comment ?
3. Pourquoi la solution proposée est-elle inutilement consommatrice de ressources ?
4. Proposer une version à base de continuer/suspendre qui ne présente pas cet inconvénient.

IV Mémoire virtuelle (5 points)

1. Le « translation look-aside buffer » (TLB) est-il aussi indispensable avec une table des pages multi-niveaux ou avec une table des pages inversées ?
2. On souhaite enrichir le fonctionnement de la mémoire virtuelle implantée en TP pour qu'un processus puisse s'approprier une page de la mémoire centrale, en empêchant son éjection. Dans ce cas, un processus non propriétaire aura le droit de lire, mais pas de modifier, le contenu de la page. Ainsi, pour une page « appropriée », le processus propriétaire peut faire lectures et modifications comme il veut, et les autres processus ne peuvent que lire la mémoire.

   On ajoute l'interface :

   ```
   void MV_grab(void *adresse);
   void MV_ungrab(void *adresse);
   ```

   La procédure `MV_grab` permet au processus courant de s'approprier la page contenant l'adresse spécifiée, et ce jusqu'à ce qu'il appelle `MV_ungrab`.

   (a) Pour chaque page, il est nécessaire de savoir si elle est appropriée et par qui. Dans quelle(s) structure(s) de données existante(s) va-t-on ranger cette information ?

   (b) Donner l'algorithme de `MV_grab`. Penser que quand un processus s'approprie une page, le contenu courant de la page n'est pas forcément celui de ce processus.

   (c) Modifier le gestionnaire du signal SIGSEGV (`handler_sigsegv`) pour prendre en compte le cas où la page est appropriée par un processus. Penser à distinguer le cas où le processus courant est l'appropriateur et celui où c'est un autre processus.

   Note : ne pas se préoccuper du contrôle d'erreur, en supposant que les processus se comportent bien. Par exemple, un processus ne doit pas appeler `MV_ungrab` sans avoir appeler `MV_grab`, ou ne doit pas appeler `MV_grab` plusieurs fois consécutivement pour la même page.