Systèmes centralisés

1h45, documents autorisés

juin 2013

Les exercices sont indépendants. Une réponse non justifiée est sans intérêt.

1 Coroutines (5 points)

On souhaite réaliser un distributeur (dispatcher) d'événement, tel que, selon l'événement, le contrôle soit transféré à un contexte (une coroutine) spécifique. La coroutine spécifique est nommé le traitant de l'événement. Le gestionnaire est déclenché magiquement (en fait, c'est une interruption matérielle qui le déclenche) et son algorithme est :

```
/* les événements sont identifiés de 0 à NBEVENT - 1 */
#define NBEVENT 20
/* Le tableau des coroutines à invoquer pour un événement donné. */
/* Ce tableau a été initialisé au démarrage avec des coroutines utilisateurs. */
coroutine_t event[NBEVENT];

/* le code du dispatcher */
void dispatcher()
   idev = obtenir_le_numéro_d_événement_à_distribuer(); // magique
   dest = event[idev];  // la coroutine correspondant à cet événement
   transférer le contrôle à la coroutine dest
   nettoyer et réarmer l'interruption matérielle // magique
```

Pour fonctionner correctement, la coroutine qui traite un événement doit finalement retourner le contrôle au dispatcher. Pour cela une fonction `return_to_dispatcher(/* sans paramètre */)` doit être définie et un traitant d'événement s'engage à l'appeler.

Questions

1. Donner le code précis du dispatcher (en ignorant les parties magiques)
2. Donner le code de `return_to_dispatcher`
3. Comment pourrait-on détecter qu'un traitant a oublié d'invoquer `return_to_dispatcher` ? Expliquer alors la mise en œuvre pour corriger automatiquement un tel oubli.

2 Processus (6 points)

Soit le code suivant :

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

processus_t p2, pmain;

void bar (void *unused)
{   int i;
    proc_suspendre();
    for (i = 0; i < 4; i++) {  /* ATTENTION : départ à 0 */
        printf("Bar: %d\n", i);
        if (i % 2 == 0)
          proc_continuer(p2);
        else
          proc_continuer(pmain);
        proc_suspendre();
    }
}
void foo (void *param)
{   int i;
    processus_t p5 = (processus_t)param;
    for (i = 1; i < 4; i++) {  /* ATTENTION : départ à 1 */
        printf("Foo: %d\n", i);
        if (i % 2 == 0)
          proc_continuer(pmain);
        else
          proc_continuer(p5);
        proc_suspendre();
    }
}
int main()
{   int i;
    processus_t p1;
    /* sched_set_scheduler(&sched_aleatoire); */
    proc_init();
    pmain = proc_self();
    p1 = proc_activer("P1", bar, NULL);
    p2 = proc_activer("P2", foo, p1);
    for (i = 0; 1; i++) {  // infinite loop
        printf("Main: %d\n", i);
        proc_suspendre();
        if (i % 2 == 0)
          proc_continuer(p1);
        else
          proc_continuer(p2);
    }
}
```

Questions

1. Sous l'hypothèse de l'utilisation de l'ordonnanceur par défaut, qui est FIFO sans préemption, qu'affiche ce programme ?
2. Ce programme s'arrête-t-il ? Si oui, dans quel état est alors chacun des processus ?
3. Ce programme marche-t-il pareil si on remplace l'ordonnanceur par défaut par l'ordonnanceur aléatoire (toujours sans préemption) ?
4. Le modifier pour qu'il fonctionne strictement comme avec l'ordonnanceur FIFO.

3 Mémoire virtuelle (4 points)

Le *working set* (ensemble de travail) d'un processus est l'ensemble des pages dont ce processus a besoin pour s'exécuter pendant son quantum de temps alloué par l'ordonnanceur sans subir de défaut de page. À chaque quantum alloué à ce processus, le *working set* peut être différent, mais, sous l'hypothèse de régularité, il est probable qu'il soit identique au quantum précédent.

1. Comment le système peut-il déterminer le *working set* d'un processus ?
2. Un phénomène d'écroulement apparaît quand les processus en cours utilisent un nombre de pages virtuelles supérieur au nombre de pages physiques. Quels sont les symptômes et pourquoi parle-t-on d'*écroulement* ?
3. Pour éviter cet écroulement, le système de gestion de mémoire virtuelle évalue périodiquement la condition $\Sigma_p WS_p \leq M$ vérifiant si la somme des working sets des processus en cours reste inférieure ou égale à la capacité mémoire totale disponible (M pages). Que peut faire le système si cette condition devient fausse ?

4 Système de fichiers (5 points)

On parle de lien dur quand plusieurs *chemins d'accès* (noms de fichier) correspondent à la même *inode*. Cela signifie qu'il existe, dans le même répertoire ou dans des répertoires différents, plusieurs associations "chaînes de caractères" $\rightarrow$ *même numéro d'inode*.

1. De tels liens durs sont-ils envisageables dans la description du système de fichier tel que vu en cours ?
2. On considère un fichier déjà existant de 52 Ko, avec des blocs de 4 Ko et des numéros de blocs sur 32 bits (4 octets). On ajoute un lien dur vers ce fichier. Combien de blocs au minimum cela nécessite-t-il ? Combien de blocs au maximum ?
3. Pour réaliser de tel lien dur, on veut rajouter une fonction `link(ancien-nom, nouveau-nom)` qui crée un nouveau lien nommé `nouveau-nom` qui pointe sur la même inode que `ancien-nom` (`nouveau-nom` et `ancien-nom` sont des chemins d'accès arbitraires). Donner l'algorithme de `link` en s'appuyant sur les fonctions déjà décrites.
4. Un répertoire n'est jamais qu'un fichier comme un autre. Pourquoi interdit-on les liens durs vers un répertoire ?
5. On a vu que la libération d'une inode (et donc la libération des blocs de données) se fait quand son compteur de référence passe à 0. L'existence de liens durs (= d'une inode ayant plusieurs noms) pose-t-elle un problème pour cette libération ? Si oui, que faut-il changer ?