Systèmes d'exploitation centralisés

1re année Informatique et Réseaux

8 juin 2016

Documents autorisés. Les exercices sont indépendants.

I Système de fichiers (3 points)

1. Sous unix, un même fichier peut avoir plusieurs noms (liens durs). Comment cela se fait-il ?
2. Est-il possible que le nombre de blocs utilisés pour ranger un fichier soit différent de (taille-du-fichier / taille-d-un-bloc), arrondi à l'entier supérieur ? Le nombre de blocs peut-il être plus grand, plus petit, les deux ?
3. Donner un exemple où l'allocation de blocs contiguë peut être utilisée en pratique. Pourquoi est-ce alors plus intéressant que l'allocation dispersée obtenue par un système de fichiers type UFS ?

II Mémoire virtuelle (2 points)

1. Dans quel(s) cas le contenu d'une page de la mémoire virtuelle d'un processus peut-il n'exister ni en mémoire physique ni en swap (fichier d'échange) ?
2. Le noyau étudié en TP implante une mémoire virtuelle paginée, avec va-et-vient (swapping) mais sans translation d'adresse (dans notre noyau, l'adresse virtuelle est égale à l'adresse en mémoire centrale). En quoi la translation d'adresse serait un complément intéressant au va-et-vient pour réduire le nombre de pages écrites ?

III Noyau – Coroutines (5 points)

Soit le code suivant :

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

#define N 4
coroutine_t c[4];
coroutine_t cmain;

void code1 (void *unused)
{
    int i = 0;
    for (i = 0; i < N; i++) {
        printf("0\n");
        cor_transferer(c[0], c[i]);
    }
    cor_transferer(c[0], cmain);
    printf("code1: fini\n");
}

void code2 (void *p)
{
    int v = *(int *)p;
    printf("%d\n", v);
    cor_transferer(c[v], c[0]);
    printf("%d : fini\n", v);
}

int main()
{
    int i;
    c[0] = cor_creer("C1", code1, NULL);
    for (i = 1; i < N; i++) {
        int *p = malloc(sizeof(int));
        *p = i;
        c[i] = cor_creer("C2", code2, p);
    }
    cmain = cor_creer("Main", NULL, NULL);
    cor_transferer(cmain,c[0]);
    printf ("Tout fini\n");
}
```

1. À quoi sert la variable `p` dans la boucle de création des coroutines ? Pourquoi ne pas passer `&i` ?
2. Détailler l'exécution du programme et en déduire ce qui s'affiche.

IV Noyau – Processus (5 points)

Soit le code suivant :

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

processus_t proc;
processus_t pmain;

void foo (void *p)
{
    int dep = *(int *)p;
    int i;
    proc_continuer(pmain);
    proc = proc_self();
    proc_suspendre();
    for (i = dep; i < 10; i += 2) {
        printf("%s: %i\n", proc_nom(proc_self()), i);
        proc_continuer(proc);
        proc = proc_self();
        proc_suspendre();
    }
}

int main()
{
    processus_t p1, p2;
    int dep1, dep2;
    /* sched_set_scheduler(&sched_aleatoire); */
    proc_init();
    dep1 = 0;
    p1 = proc_activer("P1", foo, &dep1);
    pmain = proc_self();
    proc_suspendre();
    dep2 = 1;
    p2 = proc_activer("P2", foo, &dep2);
    proc_suspendre();
    proc_continuer(p1);
    proc_suspendre();
}
```

1. Expliquer ce qu'affiche ce programme.
2. Comment se termine ce programme ?
3. Donner un code qui similaire en n'utilisant que des `proc_commuter` (= le même comportement, le même enchaînement des actions, sans changer la structure du code fourni : unique procédure `foo`, même code de boucle et d'affichage...), sous l'hypothèse de l'ordonnanceur FIFO.
4. Sans connaître l'ordonnanceur (par exemple l'ordonnanceur aléatoire), comment pourrait-on obtenir la même exécution uniquement avec des `proc_commuter` ?

V Stratégie d'ordonnancement (5 points)

On souhaite enrichir l'ordonnanceur dynamique vu en TP pour obtenir les priorités adaptatives présentées en cours :

- Plusieurs files (comme l'ordonnanceur dynamique) ;
- Quand un processus consomme tout son quantum, il est préempté et déplacé dans un niveau moins prioritaire ;
- Quand un processus libère le CPU avant la fin de son quantum, il reste au même niveau ;
- Quand un processus se bloque sur une entrée/sortie, il sera placé (quand il sera débloqué à la fin de l'entrée-sortie) à un niveau plus prioritaire.

L'interface de l'ordonnanceur est (cf TP) :

```
/* Ajoute p dans les prêts.
 * cause indique la raison pour lequel le processus devient prêt :
 * FULL_QUANTUM : préempté sur consommation de son quantum
 * PARTIAL_QUANTUM : libère le CPU avant la fin de son quantum
 * IO_FINISHED : processus débloqué car l'E/S qu'il attendait est finie
 */
enum Cause { FULL_QUANTUM, PARTIAL_QUANTUM, IO_FINISHED };
void sched_prio_ajouter_pret (processus_t qui, enum Cause cause);

/* Extrait le processus en tête des prêts. (selon la politique d'ordonnancement). */
processus_t sched_prio_choisir_elu (void);
```

Question

1. Quel est l'intérêt d'un tel ordonnanceur ?
2. Indiquer algorithmiquement les changements à apporter au scheduler dynamique fait en TP.