Systèmes d'exploitation centralisés

1re année Informatique et Réseaux

18 juin 2014

Documents autorisés.

I Principes généraux (6 points)

Barème : 1 point par question

1. Un ensemble I de processus exécute des programmes qui engendrent beaucoup de lectures et d'écritures dans des fichiers (compilations par exemple) et un autre ensemble C de processus exécute des programmes qui engendrent des calculs longs. L'ordonnanceur a-t-il intérêt à donner une plus forte priorité aux processus de l'ensemble I ou de l'ensemble C ? Justifier la réponse.
2. Donner et expliquer une stratégie d'allocation du processeur aux processus prêts assurant la répartition équitable du temps processeur entre les processus exécutables.
3. Sur un système biprocesseur, est-il concevable qu'un processus donné puisse s'exécuter sur l'un puis l'autre des deux processeurs alternativement ?
4. Dans un système de fichiers le nombre de fichiers est-il illimité ? Justifier la réponse.
5. Pourquoi le parcours d'un chemin d'accès à un fichier, par exemple /users/1AI/gr1/dupont/prog.c nécessite-t-il l'ouverture et la lecture de plusieurs fichiers ? Préciser lesquels.
6. Sous Unix, il n'existe pas de primitive permettant de détruire un fichier (mais uniquement une primitive permettant d'enlever un nom associé à une inode). Expliquer pourquoi et comment le noyau assure néanmoins la destruction des fichiers.

II Noyau – Coroutines (4 points)

Soit le code suivant :

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

coroutine_t c1, c2, c3, c4;

void code1 (void *unused)
{
    printf(" c");
    cor_transferer(c1,c2);
    printf("nais");
    cor_transferer(c4,c3);
    printf("re");
    cor_transferer(c1,c2);
    printf("se.\n");
}

void code2 (void *unused)
{
    printf("on");
    cor_transferer(c4,c1);
    printf("ce.\n");
}

void code3 (void *unused)
{
    printf(" la ");
    cor_transferer(c1,c4);
    printf("p");
    cor_transferer(c4,c2);
    printf("de.\n");

}

int main()
{
    c1 = cor_creer("C1", code1, NULL);
    c2 = cor_creer("C2", code2, NULL);
    c3 = cor_creer("C3", code3, NULL);
    c4 = cor_creer("C4", NULL, NULL);
    printf("Je");
    cor_transferer(c4,c1);
    printf("le\n");
}
```

1. Détailler l'exécution du programme et en déduire ce qui s'affiche.
2. Modifier les appels à `cor_transferer` pour que la phrase ait un sens (interdit de changer ou déplacer les `printf` !)

III Mémoire virtuelle (6 points)

Barème : 2 points par question

1. Quel est le rôle d'une unité de gestion mémoire (MMU) dans un système de mémoire virtuelle paginée ? En particulier, préciser dans quel cas l'unité de gestion mémoire engendre une interruption « défaut de page ».
   (note : la recopie littérale des transparents sera mal reçue)
2. On a pu observer le phénomène suivant sur un système offrant une mémoire virtuelle paginée : le taux d'utilisation du processeur et du disque ont augmenté de manière linéaire, au fur et à mesure que de nouveaux processus étaient lancés, jusqu'à un nombre $K-1$ de processus en cours d'exécution. Puis, lorsque le Kieme processus a été lancé, le taux d'utilisation du processeur a chuté, pour devenir presque nul (les processus ne progressent ou ne répondent pratiquement plus), tandis que le disque fonctionnait en permanence. Donner une explication plausible de ce qui est advenu, et proposer un remède.
3. On suppose que l'on dispose d'un mécanisme de mémoire virtuelle, avec chargement de pages à la demande, dont la politique de remplacement (choix de pages victimes) est LRU. Le programme suivant initialise à zéro un tableau A, que l'on suppose totalement absent de la mémoire centrale au départ de l'exécution. On suppose qu'un entier est représenté sur 4 octets, que la taille d'une page est 4 Ko, que la taille de la mémoire physique est de 1 Mo (soit 256 pages), et que les matrices sont rangées par lignes (c'est-à-dire que A[i][j] et A[i][j+1] sont rangés à des adresses successives).

```
// une ligne fait 1024 entiers, soit exactement 4 Ko ou une page
#define DIM 1024
void main() {
    int A [DIM][DIM];
    for (int i=0 ; i < DIM; i++) {
        for (int j=0 ; j < DIM; j++) {
            A[i][j] = 0;
        }
    }
}
```

   (a) Combien de défauts de page engendre l'exécution de ce programme ?
   (b) Si l'on remplace, dans ce programme, "`A[i][j]`" par "`A[j][i]`", combien de défauts de page sont alors engendrés ?

IV Systèmes de fichiers (4 points)

Un fichier épars (*sparse file*) est un fichier pour lequel seuls les blocs contenant effectivement des données sont alloués : les blocs vides (ne contenant que des 0) ne sont pas alloués. Pour autant, l'utilisateur peut parfaitement lire l'intégralité du fichier, le système traduisant l'absence de bloc en un bloc vide.

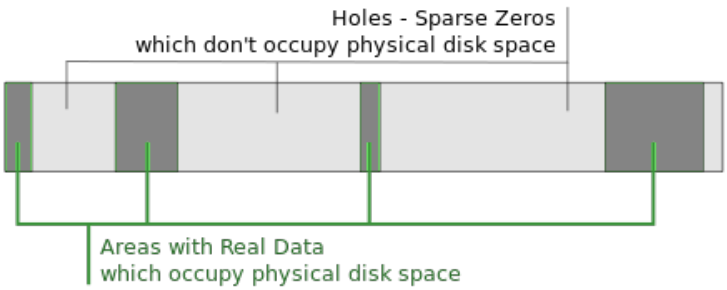

FIGURE 1 – Fichier épars

1. Quel est l'intérêt de cette technique ?
2. La taille (longueur) d'un fichier et son occupation sur disque ne sont plus nécessairement identiques. Quel est l'inconvénient ?
3. Soit le programme suivant :

```
#include <unistd.h>
#include <fcntl.h>
int main()
{
    int fd = open("toto", O_WRONLY|O_TRUNC|O_CREAT, 0644);
    write(fd, "coucou", 6);
    lseek(fd, 1024000, SEEK_SET);
    write(fd, "toto", 4);
    close(fd);
    return 0;
}
```

   (a) Quel est la longueur (en octets) du fichier ?

   (b) En supposant des blocs de 1Ko, combien de blocs de données ce fichier utilise-t-il s'il n'y a pas de support pour les fichiers épars (ignorer les blocs d'indirection) ?

   (c) En supposant des blocs de 1Ko, combien de blocs de données ce fichier utilise-t-il s'il y a un support pour les fichiers épars (ignorer les blocs d'indirection) ?

4. Expliquer quels modifications on peut faire à la couche inode (rangement des données de type UFS) pour supporter les fichiers épars.