Systèmes d'exploitation

Examen - Correction

2023-06-21

25 points

Exercice 1 : questions générales

1. Quels sont les deux modes de fonctionnement d'un processeur ? Qu'est-ce qui les différencie ? Par quel(s) programme(s) chaque mode est utilisé ? Pourquoi ? *1 point*

Un processeur sur un ordinateur a deux modes différents : le mode utilisateur et le mode noyau.

Le système d'exploitation fonctionne en mode noyau, ainsi il a un accès complet au matériel et peut exécuter toute instruction du processeur. Le reste des applications fonctionne en mode utilisateur, seul un sous-ensemble d'instructions de la machine est disponible.

Les applications utilisateurs s'exécutent en mode utilisateur pour des raisons de sécurité. On souhaite empêcher que ces applications interfèrent accidentellement ou intentionnellement avec le fonctionnement du système ou des autres applications.

2. Classez les types de mémoires qu'on trouve dans un ordinateur du plus rapide au moins rapide. *0,5 point*

Registres (processeur) > Caches (processeur, mémoire vive, réseau, etc.) > Mémoire vive (RAM) > Disques (SSD puis HDD).

3. Qu'est-ce que le pointeur d'instruction (*program counter*) ? *0,5 point*

Le pointeur d'instruction est un registre du processeur qui contient l'adresse mémoire de la prochaine instruction à extraire pour un programme en cours d'exécution. Une fois l'instruction extraite, le pointeur d'instruction est mis à jour pour pointer vers l'instruction suivante.

4. On considère un fichier `especes.txt` contenant une liste de noms d'espèces animales. Chaque nom d'espèce se trouve sur sa propre ligne. Écrivez une commande en shell Bash permettant d'enregistrer dans un fichier `top10.txt` les 10 premiers éléments de la liste lorsqu'elle est triée par ordre alphabétique. *1 point*

```
sort especes.txt | head > top10.txt
```

ou

```
cat especes.txt | sort | head > top10.txt
```

5. On souhaite implémenter un serveur HTTP. Celui-ci doit pouvoir traiter plusieurs requêtes de façon concurrente. Décrivez dans les grandes lignes une implémentation possible faisant appel à des mécanismes du système d'exploitation. *0,5 point*

On pourrait utiliser des threads pour mettre en œuvre une approche producteur-consommateur. Un thread producteur place les requêtes HTTP entrantes dans une file d'attente. Des threads consommateurs extraient les requêtes de cette file d'attente et les traitent. Le producteur et les consommateurs se coordonnent grâce à un sémaphore.

6. L'erreur de segmentation (*segmentation fault*) est une erreur assez courante pour les programmes en C. Donnez un exemple d'instructions en C susceptibles de causer une erreur de segmentation. Expliquez ensuite pourquoi une telle erreur se produit. *1 point*

```
int *p; // pointeur non-initialisé
*p = 9;
```

Chaque processus de l'espace utilisateur possède son propre espace d'adressage virtuel et (sauf autorisation explicite) ne peut pas accéder à la mémoire d'autres processus. Ici, le pointeur `p` n'est pas initialisé, ainsi il pointe vers une adresse mémoire arbitraire, probablement située hors de l'espace d'adressage. Lors de l'affectation de l'entier 9, le programme tente d'accéder à une adresse mémoire non-existante (hors de l'espace d'adressage) ou à laquelle il n'a pas le droit d'accéder. En réponse, le système d'exploitation bloque l'opération et met fin au processus (par défaut).

7. En quoi le système d'exploitation a-t-il besoin d'un support du matériel (*hardware*) pour basculer entre les processus dans une approche non-coopérative (préemptive) ? *1 point*

Dans une approche multitâche non-coopérative, pendant la séquence de démarrage, le système d'exploitation lance une minuterie. La minuterie génère une interruption toutes les quelques millisecondes. Quand une interruption se produit, le processus en cours d'exécution est arrêté, l'état du programme est sauvegardé et un gestionnaire d'interruption préconfiguré dans le système d'exploitation s'exécute. Ainsi, une interruption de minuterie donne au noyau la possibilité de s'exécuter à nouveau sur le processeur pour éventuellement exécuter un autre processus utilisateur.

8. Expliquez la différence entre un lien physique (*hard link*) et un lien symbolique (*soft link*) dans un système de fichiers Unix. *0,5 point*

Un lien physique est un fichier normal, et la référence du fichier renvoit à l'inode du fichier pour lequel le lien physique a été créé. Un lien symbolique est un type particulier de fichier qui contient le nom (ou chemin) du fichier original. Il ne pointe pas vers l'inode de ce dernier.

Exercice 2 : ordonnancement

Les politiques d'ordonnancement peuvent facilement être représentées sur un schéma. Par exemple, supposons que l'on exécute la tâche `A` pendant 10 unités de temps, puis que l'on exécute `B` pendant 10 unités également. Notre schéma de cette politique d'ordonnancement aurait l'apparence suivante :

```
    |
CPU | AAAAAAAAAABBBBBBBBBB
    | --------------------
    0          10        20
```

Dans cet exercice, vous devrez montrer votre compréhension des politiques d'ordonnancement en réalisant des schémas similaires.

1. (a) Dessinez un schéma pour la politique d'ordonnancement *Shortest job first (SJF)* avec trois tâches, `A`, `B` et `C`, avec des temps d'exécution respectifs de 15, 10 et 5 unités de temps. `A` et `B` arrivent dans le système à l'instant `0`, `C` arrive à l'instant `4`. *1 point*

Assurez-vous de compléter l'axe des abcisses de manière appropriée.

```
    |
CPU | BBBBBBBBBBCCCCCAAAAAAAAAAAAAAA
    | ------------------------------------------------------------
    0          10   15             30
```

1. (b) Quel est le temps de traitement moyen *(turnaround time)* et le temps de réponse moyen étant donné cet ordonnancement *SJF* des tâches ? *1 point*

temps de traitement moyen $= (10 + (15 - 4) + 30)/3 = 17$

temps de reponse moyen $= (0 + (10 - 4) + 15)/3 = 7$

2. (a) Dessinez un schéma pour la politique d'ordonnancement *Round robin (RR)* avec trois tâches, `A`, `B` et `C`, avec des temps d'exécution de 6 unités de temps, en supposant des tranches de temps de taille 2 unités de temps. Les tâches arrivent toutes dans le système à l'instant `0`. *1 point*

Assurez-vous de compléter l'axe des abcisses de manière appropriée.

```
    |
CPU | AABBCCAABBCCAABBCC
    | ------------------------------------------------------------
    0       6    12    18
```

2. (b) Quel est le temps de traitement moyen *(turnaround time)* et le temps de réponse moyen étant donné cet ordonnancement *RR* des tâches ? *1 point*

temps de traitement moyen $= (14 + 16 + 18)/3 = 16$

temps de reponse moyen $= (0 + 2 + 4)/3 = 2$

Exercice 3 : virtualisation de la mémoire

Dans un système utilisant des registres *base* et *bounds* pour virtualiser un petit espace d'adressage, on a la trace de références mémoire suivante :

```
Virtual Address Trace
    VA  0:  0x000002EE (decimal: 750)  -->  VALID: 0x000036B0 (decimal: 14000)
    VA  1:  0x0000021C (decimal: 540)  -->  VALID: 0x000035DE (decimal: 13790)
    VA  2:  0x000003A8 (decimal: 936)  -->  SEGMENTATION VIOLATION
```

Que peut-on en déduire sur la valeur du registre *base* ? Quelle estimation de la valeur du registre *bounds* peut-on donner ? *2 points*

$base = 14000 - 750 = 13790 - 540 = 13250$

Pour rappel la valeur du registre bounds est égale à la taille de l'espace d'adressage. 750 est une adresse valide, 936 ne l'est pas, ainsi $750 < bounds <= 936$.

Exercice 4 : pagination et TLB

1. Quelles sont les étapes pour accéder à une adresse mémoire avec une gestion de la mémoire par pagination (sans TLB) ? Combien de références mémoires sont nécessaires pour accéder à une adresse mémoire donnée avec cette approche ? *1 point*

1. Extraire le numéro de page de l'adresse virtuelle.
2. Récupérer l'entrée correspondante dans la page table.
3. L'entrée permet ensuite d'obtenir le numéro de page frame.
4. A partir du numéro de page frame et de l'offset (présent dans l'adresse virtuelle), on peut récupérer les données présentes à l'adresse physique.

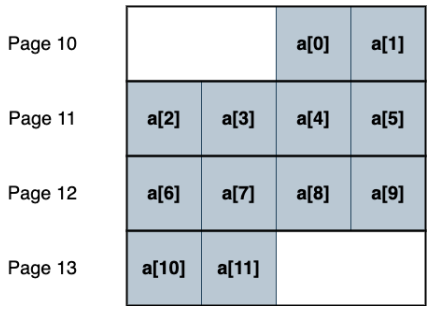
Ainsi, pour chaque référence mémoire, la pagination oblige le système d'exploitation à effectuer une référence mémoire supplémentaire pour récupérer l'entrée correspondante dans la page table.

2. On considère à présent un système avec un TLB ne comportant qu'une seule entrée.

Étant donné le programme suivant :

```
int product = 0;
for (int i=0; i<12; i++) {
    product = product * a[i];
}
```

et la disposition des données dans les pages mémoire suivante :



Combien de TLB "*hits*" et "*misses*" se produisent lors de l'exécution du programme ? De quelle propriété du programme le TLB tire profit pour améliorer les performances ? *1,5 point*

4 misses (`a[0]`, `a[2]`, `a[6]`, `a[10]`) et 8 hits. Dans ce cas, le TLB améliore les performances grâce à la localité spatiale.

Exercice 5 : implémentation d'un verrou

On considère la primitive `FetchAndStoreOne()`. Elle exécute une unique instruction atomique et est définie de la façon suivante :

```
int FetchAndStoreOne(int *ptr) {
    int old = *ptr; // récupère l'ancienne valeur dans 'ptr'
    *ptr = 1;       // donne à 'ptr' la valeur 1
    return old;     // renvoie l'ancienne valeur
}
```

Vous devez définir les fonctions `lock_init()`, `lock()` et `unlock()` et une structure `lock_t` pour implémenter un *spin lock* en utilisant `FetchAndStoreOne()`. *3 points*

```
typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    // 0 indique que le verrou est disponible,
    // 1 indique qu'il est détenu
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (FetchAndStoreOne(&lock->flag) == 1)
        ;
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Exercice 6 : problème du à la concurrence

On considère le code suivant qui additionne deux vecteurs et le fait d'une manière sûre pour le multithread.

```
void vector_add(vector *v1, vector *v2) {
    mutex_lock(v1->lock);
    mutex_lock(v2->lock);
    for (i = 0; i < v1->size; i++) {
        v1[i] = v1[i] + v2[i];
    }
    mutex_unlock(v1->lock);
    mutex_unlock(v2->lock);
}
```

On vous dit alors que deux threads différents, 1 et 2, s'exécutent simultanément et appellent ce code de la façon suivante :

```
// Thread 1:                          // Thread 2:
vector_add(&vectorA, &vectorB);       vector_add(&vectorB, &vectorA);
```

Quel problème peut se produire ? Expliquez pourquoi (vous pouvez faire un dessin). Comment peut-on résoudre ce problème ? *1,5 point*

On risque un interblocage (*deadlock*) :

- Thread 1 : acquisition de `vectorA->lock`
- Thread 2 : acquisition de `vectorB->lock`
- Thread 1 : tentative d'acquisition de `vectorB->lock` mais celui-ci n'est pas libre, attente…
- Thread 2 : tentative d'acquisition de `vectorA->lock` mais celui-ci n'est pas libre, attente…

Une façon de résoudre ce problème est d'ajouter un verrou supplémentaire qu'il faut acquérir avant de pouvoir tenter d'acquérir les verrous des vecteurs.

Exercice 7 : un autre problème du à la concurrence

On considère à présent le programme suivant.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct __sysinfo {
    char *status
} sysinfo_t;

sysinfo_t *s;

void *init(void *arg) {
    s = (sysinfo_t *) malloc(sizeof(sysinfo_t));
    s->status = "READY";
    return NULL;
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, init, NULL);

    printf("system status: %s\n", s->status);

    pthread_join(t, NULL);
    free(s);
    return 0;
}
```

Ce programme n'est pas correct. Quel problème peut se produire lors de l'exécution ? Comment rendre ce programme correct ? Vous pouvez annoter le code pour vos explications si nécessaire. *2 points*

La variable globale `s` n'a pas nécessairement été initialisée lors de l'appel de `printf("system status: %s\n",s->status)`, cela dépend de l'ordonnancement des threads du programme. On risque alors de déréférencer un pointeur NULL ce qui créera une erreur de segmentation.

Pour rendre ce programme correct, il faut faire en sorte que le thread principal attende que le thread qui exécute la fonction `init` ait finit d'initialiser la variable `s`

Cela peut-être fait grâce à une *condition variable* qui permet au thread "init" d'indiquer au thread principal que l'initialisation est terminée.

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
```

```c
int mtInit = 0;

void *init(void *arg) {
    s = (sysinfo_t *) malloc(sizeof(sysinfo_t));
    s->status = "READY";

    // signale que l'initialisation est terminée
    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);

    return NULL;
}

int main() {
    // ...

    // attendre l'initialisation de s
    pthread_mutex_lock(&mtLock);
    while(mtInit == 0)
        pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);

    printf("system status: %s\n", s->status);

    // ...
}
```

Exercice 8 : système de fichiers et journalisation

Considérons le *very simple file system* que nous avons étudié en classe. Il possède un seul super-bloc, un seul tableau de bits (*bitmap*) de données (DB), un seul tableau de bits (*bitmap*) d'inodes (IB), une série de blocs d'inode (I) et une série de blocs de données (D) ; tous les blocs ont une taille de 4 KB (4096 octets).

1. On suppose que chaque inode possède 6 pointeurs directs et 2 pointeurs indirects. Les adresses de disques sont sur 8 octets. Quelle est la taille maximale d'un fichier dans le système de fichiers ? *1 point*

Indiquez les étapes du calcul, il n'est pas nécessaire de donner le résultat final.

$Pointeurs\ par\ bloc\ de\ pointeurs = PB = \frac{4096}{8}$

$Taille\ max = (nombre\ pointeurs) * taille\ bloc$

$Taille\ max = (pointeurs\ directs + blocs\ de\ pointeurs \times PB) * taille\ bloc$

$Taille\ max = (6 + 2 \times \frac{4096}{8}) * 4096$

2. Décrivez les opérations qui prennent place dans le système de fichiers lorsqu'on ouvre le fichier `/home/image.txt` avec l'appel système `open("/home/image.txt", ...)`. *1 point*

1. Lecture du bloc qui contient l'inode racine (généralement l'inode numéro 2).
2. Regarder à l'intérieur de l'inode pour trouver des pointeurs vers les blocs de données.
3. Lecture des blocs de données du répertoire `/`, pour trouver le numéro d'inode de l'entrée `home`.
4. Regarder à l'intérieur de l'inode pour trouver des pointeurs vers les blocs de données. Lecture des blocs de données du répertoire `home`, pour trouver le numéro d'inode de l'entrée `image.txt`.
5. Allocation d'un descripteur de fichier référençant l'inode de `image.txt`.

3. On suppose que le système de fichiers n'est pas journalisé. On souhaite mettre à jour le bloc de données `D2` dans les blocs de données du fichier déjà existant `/home/image.txt`. Donnez un exemple dans lequel le système de fichiers se retrouve dans un état incohérent suite à une défaillance lors de la mise à jour des structures de données. *1 point*

Exemple : Seul l'inode mis à jour est écrit sur le disque. Il pointe vers le bloc de données D2 qui n'a pas été mis à jour. Nous lirons des données obsolètes sur le disque.

4. On suppose à présent que le système de fichiers est journalisé, avec une journalisation physique. On souhaite mettre à jour le bloc de données `D2` dans les blocs de données du fichier `/home/image.txt`. Quel protocole doit-on suivre pour écrire ces nouvelles données tout en garantissant que le système ne se retrouvera pas dans un état incohérent en cas de panne ? *1 point*

1. Écriture dans le journal : écriture du contenu de la transaction dans le journal (TxB, I[v2], B[v2], D2[v2])
2. Commit de la transaction : écriture du bloc de validation de la transaction (TxE).
3. Checkpoint : écriture des métadonnées et des mises à jour des données à leur emplacement final.
4. Libération