Systèmes d'exploitation

Examen - Correction

2023-08-29

25 points

Exercice 1 : questions générales

1. Quelles sont les deux fonctions principales d'un système d'exploitation ? *1 point*

1. Fournir aux programmeurs d'applications (et aux applications) un ensemble abstrait et unifié de ressources au lieu des ressources matérielles désordonnées.
2. Gérer les ressources matérielles et les partager entre les applications.

2. Quels sont les types de multiplexage possibles ? Quel type de multiplexage peut être utilisé pour partager l'utilisation de la mémoire ? Du processeur ? *1 point*

Il existe deux types de multiplexages. Avec le multiplexage temporel, les différents utilisateurs ou programmes utilisent une ressource à tour de rôle. À l'opposé, avec le multiplexage spatial, au lieu que les utilisateurs se relaient, chacun obtient une partie de la ressource. La mémoire peut être partagée avec le multiplexage spatial, le processeur avec le multiplexage temporel.

3. Dans quel mode de fonctionnement du processeur le shell s'exécute-t-il ? *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.

Le shell ne fait pas partie du noyau et s'exécute en mode utilisateur.

4. À quoi sert un appel système dans un système d'exploitation ? Donnez un exemple d'appel système et expliquez son fonctionnement. *1 point*

Un appel système est la manière dont un programme informatique demande un service au système d'exploitation sur lequel il est exécuté. Les services incluent notamment l'accès au matériel (disque, réseau) ou la création d'un nouveau processus.

Avec un système Unix, l'ouverture d'un fichier se fait en utilisant l'appel système open.

5. Qu'est-ce qu'une commutation de contexte (*context switch*) ? Donnez deux raisons pour lesquelles une commutation de contexte peut se produire ? *1 point*

Une commutation de contexte est un mécanisme qui :

1. sauvegarde l'état du processus en cours (valeurs des registres notamment), dans une structure du noyau;
2. restaure l'état du processus à exécuter ensuite;
3. transfère le contrôle du processeur au processus suivant.

Raisons d'une commutation de contexte :

- Le processus se termine.
- Le processus effectue une opération d'E/S lente et le système d'exploitation passe à une autre tâche qui est prête.
- Le matériel nécessite l'aide du système d'exploitation et émet une interruption.
- Le système d'exploitation décide de préempter le processus et de passer à une autre (c'est-à-dire que le processus a épuisé sa tranche de temps).

6. On considère le code en C ci-dessous.

```c
int main() {
    fork();
    fork();
    return 0;
}
```

Combien de processus au total sont créés lors de l'exécution de ce programme ? *1 point*

4 processus au total sont créés : 1 (main) + 1 (1er fork) + 2 (2e fork).

7. Lorsqu'on souhaite afficher une page web dans son navigateur, celui-ci doit télécharger les éléments de la page depuis un ou plusieurs emplacements distants. Il doit également réaliser un rendu graphique de ces éléments pour les afficher à l'écran. Quel mécanisme du système d'exploitation pourrait-on utiliser pour faire en sorte que l'affichage d'une page web par un navigateur se fasse rapidement. *1 point*

On pourrait utiliser des threads. Le téléchargement de chaque élément depuis un emplacement différent aurait lieu dans son propre thread. De façon concurrente, le rendu graphique serait géré par un thread. Les threads pourraient communiquer selon une logique producteurs-consommateur avec un tampon partagé. Ainsi, le temps nécessaire aux entrées/sorties de téléchargement des éléments serait utilisé pour réaliser le rendu graphique de la page.

8. Écrivez une commande shell qui stocke dans le fichiers `recentes.txt` la liste des 10 dernières commandes shell précédemment exécutées. *1 point*

```sh
history | tail > recentes.txt
```

9. On considère le programme ci-dessous.

```c
int main(int argc, char *argv[]) {
    printf("%p\n", main);
    printf("%p\n", malloc(100e6));
```

```
    int x = 3;
    printf("%p\n", &x);
    return 0;
}
```

Lors de son exécution, on obtient l'affichage suivant dans la console :

```
0x40057d
0xcf2010
0x7fff9ca45fcc
```

Quelle adresse mémoire (virtuelle) se trouve dans la pile (*stack*) et quelle adresse se trouve dans le tas (*heap*) ? *1 point*

`0xcf2010` se trouve dans le tas et `0x7fff9ca45fcc` dans la pile.

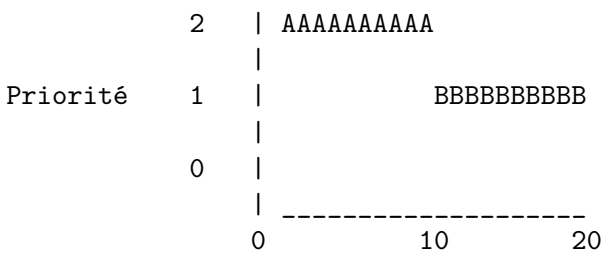
Exercice 2 : multi-level feedback queue

1. Quelles sont les règles d'une politique d'ordonnancement multi-level feedback queue (MLFQ) ? *1 point*

- Règle 1 : les tâches dans la file avec le niveau de priorité le plus haut, sont exécutées en priorité.
- Règle 2 : plusieurs tâches peuvent avoir la même priorité, dans ce cas l'ordonnancement round-robin est utilisé entre ces tâches.
- Règle 3 : les nouvelles tâches ont la priorité la plus élevée.
- Règle 4 : lorsqu'une tâche a épuisé son temps alloué à un niveau donné, sa priorité est réduite.
- Règle 5 : toutes les tâches sont déplacées périodiquement vers la file d'attente la plus prioritaire.

On considère un ordonnanceur multi-level feedback queue (MLFQ). Dans cette question, vous devez dessiner un schéma pour indiquer comment l'exécution des tâches se déroule avec cet ordonnanceur.

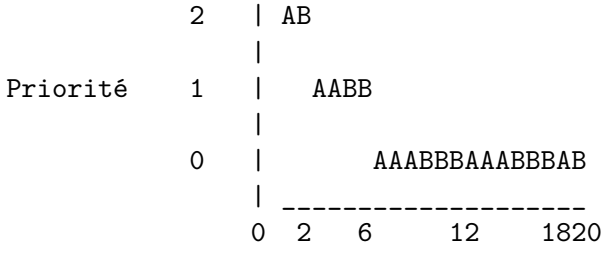
Par exemple, supposons que l'on exécute la tâche `A` pendant 10 unités de temps, au niveau de priorité 2, puis que l'on exécute `B` pendant 10 unités au niveau de priorité 1. Notre schéma aurait l'apparence suivante :



Ainsi, l'axe des ordonnées indique la priorité de la tâche en cours d'exécution. Notez bien que seul la tâche en cours d'exécution apparaît sur le schéma, les tâches en attente dans les files n'apparaissent pas.

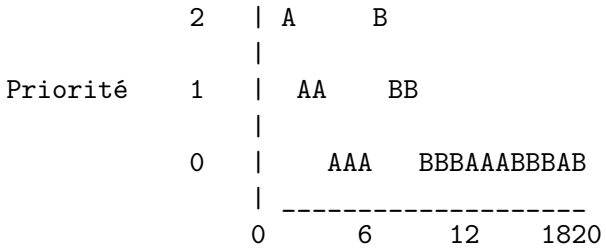
2. On considère un MLFQ à 3 niveaux (la priorité la plus haute est 2, la plus basse 0). L'ordonnanceur ne déplace pas les tâches entre les files d'attente. On considère deux tâches (`A` et `B`). Ces tâches ne réalisent pas d'entrées/sorties. Elles ont des temps d'exécution de 10 unités de temps et arrivent dans le système à `T=0`. La tranche de temps d'exécution (sans interruption) est de 1 unité de temps, au niveau de priorité le plus haut, 2 au niveau moyen et 3 au niveau le plus bas.

Dessinez un schéma pour indiquer comment ces tâches seront exécutées par l'ordonnanceur. Assurez-vous de compléter l'axe des abscisses de manière appropriée. *1 point*



3. (a) On considère à présent un scénario dans lequel les paramètres de l'ordonnanceur sont les mêmes que dans la question 2. Les tâches sont différentes : `A` et `B` s'exécutent pendant 10 unités de temps et ne réalisent pas d'entrées/sorties. En revanche, cette fois, `A` arrive dans le système à `T=0` et B à `T=6`.

Dessinez un schéma pour indiquer comment ces tâches seront exécutées par l'ordonnanceur. Assurez-vous de compléter l'axe des abscisses de manière appropriée. *1 point*



3. (b) Étant donné l'ordonnancement des tâches dans 3. (a), quel est le temps de traitement *(turnaround time)* et le temps de réponse de la tâche A ? *1 point*

temps de traitement $= 19 - 0 = 19$

temps de reponse $= 0$

3. (c) Étant donné l'ordonnancement des tâches dans 3. (a), quel est le temps de traitement *(turnaround time)* et le temps de réponse de la tâche B ? *1 point*

temps de traitement $= 20 - 6 = 14$

temps de reponse $= 0$

Exercice 3 : translation-lookaside buffer (TLB)

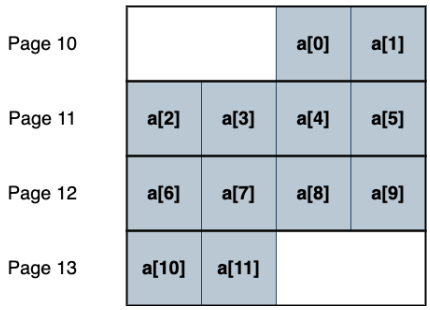
On considère un système avec un TLB comportant deux entrées.

Étant donné le programme suivant :

```
int product = 1;

int i = 0;
for (int j=0; j < 6; j++) {
    product = product * a[i];
    i = (i + 4) % 12;
}
```

et la disposition des données dans les pages mémoire suivante :



1. Combien de TLB "*hits*" et "*misses*" se produisent avec une politique de remplacement des entrées *least recently used (LRU)*, lors de l'exécution du programme ? Expliquez. *1 point*

6 misses (`a[0]`, `a[4]`, `a[8]`, `a[0]`, `a[4]`, `a[8]`) et 0 hits.

Le programme boucle sur 3 pages avec un TLB de taille 2, ainsi le TLB LRU échoue à chaque accès.

2. Quelle politique de remplacement des entrées privilégier dans ce cas là ? Combien de TLB "*hits*" et "*misses*" peut on espérer dans le meilleur des cas ? Expliquez. *1 point*

Dans ce cas là, on privilégiera une politique de remplacement des entrées aléatoire.

On peut espérer 4 misses et 2 hits.

Exercice 4 : implémentation d'un verrou

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

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected) { *ptr = new; }
    return actual;
}
```

Vous devez définir les fonctions `lock_init()`, `lock()` et `unlock()` et une structure `lock_t` pour implémenter un *spin lock* en utilisant `CompareAndSwap()`. *2 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 (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ;
}

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

Exercice 5 : primitives de synchronisation

Dans cet exercice, on vous présente le code source de deux programmes *multi-threads*. Ces programmes ne sont pas corrects : suivant l'ordre d'exécution des threads on peut obtenir des résultats inattendus.

Annotez le code de ces programmes avec les primitives de synchronisation adaptées pour rendre ces programmes corrects et faire en sorte qu'ils produisent des résultats déterminés. La documentation sur les primitives de synchronisation est fournie en annexe.

1. *1 point*

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

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int compte = 100;

void *retrait(void *args) {
    pthread_mutex_lock(&lock);
    compte = compte - 10;
    pthread_mutex_unlock(&lock);
    return NULL;
}

void *depot(void *args) {
    pthread_mutex_lock(&lock);
    compte = compte + 20;
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, retrait, NULL);
    pthread_create(&t2, NULL, depot, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Montant sur le compte: %d\n", compte);
    return 0;
}
```

2. *2 points*

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

int continuer = 0;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *thread1(void *args) {
    printf("Ce message doit s'afficher en premier!\n");

    pthread_mutex_lock(&lock);
    continuer = 1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&lock);
    return NULL;
}

void *thread2(void *args) {
    pthread_mutex_lock(&lock);
    while (continuer == 0)
        pthread_cond_wait(&cond, &lock);
    pthread_mutex_unlock(&lock);

    printf("Ce message doit s'afficher en second!\n");
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

Exercice 6 : 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).

1. Comment un inode fait-il référence aux blocs de données d'un fichier ? *1 point*

Une approche simple consiste à avoir un ou plusieurs pointeurs directs (adresses de disque) à l'intérieur de l'inode. Chaque pointeur renvoie à un bloc de disque appartenant au fichier.

Pour prendre en charge des fichiers plus volumineux, une idée courante est d'avoir un pointeur spécial : le pointeur indirect. Au lieu de pointer vers un bloc de données utilisateur, il pointe vers un bloc qui contient plusieurs pointeurs, chacun d'entre eux pointant vers des données utilisateur.

2. Un nouveau fichier `/home/new.txt` a été ouvert grâce à l'appel système `open()` et un descripteur de fichier a été obtenu. Décrivez les opérations qui prennent place dans le système de fichiers lorsqu'on écrit dans le fichier `/home/new.txt` avec l'appel système `write()`. *1 point*

1. Lecture de l'inode (I) pour déterminer si le fichier possède déjà des blocs alloués.
2. Lecture du tableau de bit de données (DB) pour trouver un bloc libre.
3. Écriture du tableau de bit de données (DB) pour allouer un bloc de données (D) pour le fichier.
4. Mise à jour de l'inode (taille, bloc alloués, date de modification…).
5. Écriture du bloc de données.

3. On suppose que le système de fichiers n'est pas journalisé. On souhaite allouer le bloc de données `D2` pour le fichier `/home/new.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 du tableau de bits de données (DB). *1 point*

Exemple : Seul le tableau de bits de données (DB[v2]) mis à jour est écrit sur le disque. Il indique que le bloc de données D2 est alloué, mais il n'y a pas d'inode qui pointe vers lui. Ainsi, le système de fichiers est dans un état incohérent.

4. On suppose à présent que le système de fichiers est journalisé, avec une journalisation des métadonnées. On souhaite mettre à jour le bloc de données `D2` dans les blocs de données du fichier `/home/new.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 des données utilisateur (D2[v2]) à l'emplacement final.
2. Écriture des métadonnées dans le journal (TxB, I[v2], IB[v2], DB[v2]).
3. Commit de la transaction : écriture du bloc de validation de la transaction (TxE).
4. Checkpoint : écriture des métadonnées à leur emplacement final.
5. Libération.

Annexes

pthread_mutex_lock(3p) — Linux manual page

NAME

```
pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock - lock and unlock a mutex
```

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

The mutex object referenced by mutex shall be locked by a call to pthread_mutex_lock() that returns zero or [EOWNERDEAD]. If the mutex is already locked by another thread, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

pthread_mutex_destroy(3p) — Linux manual page

NAME

```
pthread_mutex_destroy, pthread_mutex_init - destroy and initialize a mutex
```

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION

[...]

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes. The effect shall be equivalent to dynamic initialization by a call to pthread_mutex_init() with parameter attr specified as NULL, except that no error checks are performed.

pthread_cond_timedwait(3p) — Linux manual page

NAME

pthread_cond_timedwait, pthread_cond_wait - wait on a condition

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

DESCRIPTION

The pthread_cond_timedwait() and pthread_cond_wait() functions shall block on a condition variable. The application shall ensure that these functions are called with mutex locked by the calling thread; otherwise, an error (for PTHREAD_MUTEX_ERRORCHECK and robust mutexes) or undefined behavior (for other mutexes) results.

pthread_cond_destroy(3p) — Linux manual page

NAME

pthread_cond_destroy, pthread_cond_init - destroy and initialize condition variables

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

DESCRIPTION

[...]

In cases where default condition variable attributes are appropriate, the macro PTHREAD_COND_INITIALIZER can be used to initialize condition variables. The effect shall be equivalent to dynamic initialization by a call to pthread_cond_init() with parameter attr specified as NULL, except that no error checks are performed.