Exam 2019

Questions de cours ;

1) **Qu'est ce qu'un hyperviseur ?**

Un hyperviseur (également appelé moniteur de machine virtuelle ou VMM) est un logiciel de virtualisation qui permet de créer et de gérer plusieurs machines virtuelles sur une même machine physique.
L'hyperviseur est responsable de l'allocation des ressources matérielles (comme le processeur, la mémoire, le stockage, etc.) aux différentes machines virtuelles, ainsi que de la gestion des interactions entre ces machines virtuelles et le matériel physique

2) **Quel est l'intérêt de la paravirtualisation?**

La paravirtualisation est une technique de virtualisation dans laquelle le système d'exploitation invité (ou la machine virtuelle) est modifié pour prendre en compte l'environnement virtualisé et interagir directement avec l'hyperviseur.
L'intérêt de la paravirtualisation est de réduire l'overhead de virtualisation, c'est-à-dire la perte de performances due à la virtualisation. En permettant à la machine virtuelle de communiquer directement avec l'hyperviseur, la paravirtualisation permet d'améliorer les performances des applications s'exécutant dans la machine virtuelle, ainsi ça permet d'alléger le traitement des accès au matériel par l'OS invite.

3) **Pourquoi, sous Android, la machine virtuelle Dalvik remplace la JVM ?**

La machine virtuelle Dalvik a été développée par Google pour répondre aux besoins spécifiques des appareils mobiles sous Android. Cette VM a été conçue pour être adaptée aux architectures de téléphones mobiles avec des ressources limitées, en termes de mémoire, de puissance de traitement et d'énergie.
En effet, la machine virtuelle Dalvik permet un accès efficace au système d'exploitation Linux d'Android, ce qui permet de garantir une interaction fluide entre les applications et le système d'exploitation.
Le format bytecode Dalvik utilisé par la machine virtuelle Dalvik est moins gourmand en mémoire que le bytecode de la JVM (Java Virtual Machine qui utilise JIT), ce qui permet une optimisation des ressources mémoire disponibles sur les appareils mobiles. De plus, grâce à cette optimisation, la machine virtuelle Dalvik permet une minimisation des répétitions d'exécution, ce qui permet d'améliorer les performances des applications.
En résumé, la machine virtuelle Dalvik est une VM Java adaptée pour les systèmes à ressources limitées tels que les téléphones mobiles sous Android. Elle permet un accès efficace à un système Linux, utilise un bytecode moins gourmand en mémoire et minimise les répétitions d'exécution, ce qui en fait une solution optimale pour les environnements mobiles avec des ressources limitées.

4) **Un micro-noyau comme L4 peut être utilisé comme hyperviseur d'une machine virtuelle exécutant un système Linux. Que se passe-t-il lorsque le système Linux exécute la commande fork? Qui contrôle le processus créé?**

Lorsque le système Linux exécute la commande fork() dans une machine virtuelle sur un micro-noyau comme L4, cette commande est gérée par le micro-noyau L4 qui crée le nouveau processus enfant et alloue les ressources nécessaires. Le contrôle du processus enfant est ensuite assuré par le système d'exploitation invité, c'est-à-dire le système Linux.

2ème version : Effectivement, un micro-noyau comme L4 peut être utilisé comme hyperviseur d'une machine virtuelle exécutant un système Linux.
Lorsque le système Linux exécute la commande fork(), cela crée un nouveau processus enfant en dupliquant le processus parent. Le processus enfant hérite de la mémoire, des fichiers ouverts et des autres ressources du processus parent.
Dans le cas d'une machine virtuelle exécutant un système Linux sur un micro-noyau comme L4, le processus parent créé par le système Linux est en fait un processus utilisateur dans la

machine virtuelle. Ce processus est géré par le système d'exploitation invité, c'est-à-dire le système Linux.
Lorsque le processus parent de la machine virtuelle exécute la commande fork(), cette commande est interceptée par le micro-noyau L4 qui gère la création du nouveau processus enfant. Le micro-noyau L4 va donc allouer les ressources nécessaires au nouveau processus enfant, telles que de la mémoire et des fichiers ouverts, et va lui donner accès aux ressources du parent.
Le contrôle du processus enfant est assuré par le système d'exploitation invité, c'est-à-dire le système Linux. Le micro-noyau L4 agit simplement en tant que gestionnaire de ressources pour la machine virtuelle et assure la séparation des processus entre les différents systèmes invités exécutés sur la machine hôte.
En résumé, lorsqu'un système Linux exécutant dans une machine virtuelle sur un micro-noyau comme L4 utilise la commande fork(), cette commande est interceptée par le micro-noyau L4 qui gère la création du nouveau processus enfant. Le contrôle de ce processus enfant est ensuite assuré par le système d'exploitation invité, c'est-à-dire le système Linux.

Exercice 1 :

4 tâches T1... Ta s'exécutent sur un système dont l'ordonnanceur utilise un mécanisme de priorité et un tourniquet au sein d'une priorité donnée. La priorité d'une tâche est calculée en cumulant le temps pendant lequel la tâche s'est exécutée; elle est exprimée en unité de temps et n'est pas nécessairement entière. La tâche la plus prioritaire est celle dont la valeur numérique de la priorité est la plus faible.
À l'instant O, les quatre tâches ont une priorité initialisée à 0 et sont stockées dans une liste dans l'ordre de sortie T1...T4 (la tâche T1 a donc la priorité la plus élevée). L'unité de temps utilisée est la durée du quantum. Le basculement ne se fera que si la tâche en cours se bloque ou si son quantum est terminé. Le temps de basculement est négligeable.
Ces différentes tâches manipulent des ressources communes protégées par deux verrous Vi et V. Le tableau 1 montre l'utilisation que chaque tâche fait des verrous (L(Vi) signifie que la tâche chercher à verrouiller le verrou Vi, U(Vi) signifie qu'elle le déverrouille).
Dans le tableau 1, pour chaque tâche, la première ligne donne la date (en unité de temps et dans le temps tel que le « voit » la tâche), la deuxième donne l'action réalisée.

Question 1: Montrer l'ordonnancement dans le temps de ces quatre tâches en supposant que chacune d'elles nécessite une durée d'exécution totale de quatre unités de temps.

T1	T2	T3	T4
0.5 2.5	0.5 2.5	0.5 2	0.7 1
L(V1) U(V1)	L(V2) U(V2)	L(V1) U(V1)	L(V2) U(V2)

Voici l'ordonnancement dans le temps des quatre tâches en supposant que chacune nécessite une durée d'exécution totale de quatre unités de temps :

- Au temps 0, la tâche T1 est la plus prioritaire et est exécutée. Elle demande l'accès exclusif au verrou V1.
- Au temps 0.5, la tâche T2 devient la plus prioritaire et est exécutée. Elle demande l'accès exclusif au verrou V2.
- Au temps 1, la tâche T4 devient la plus prioritaire et est exécutée. Elle demande l'accès exclusif au verrou V2, qui est déjà détenu par la tâche T2. La tâche T4 doit donc attendre la libération de ce verrou par T2.

- Au temps 2, la tâche T3 devient la plus prioritaire et est exécutée. Elle demande l'accès exclusif au verrou V1, qui est déjà détenu par la tâche T1. La tâche T3 doit donc attendre la libération de ce verrou par T1.
- Au temps 2.5, la tâche T1 redevient la plus prioritaire et est exécutée. Elle libère le verrou V1, ce qui permet à la tâche T3 d'acquérir ce verrou et de continuer son exécution.
- Au temps 3, la tâche T2 redevient la plus prioritaire et est exécutée. Elle libère le verrou V2, ce qui permet à la tâche T4 d'acquérir ce verrou et de continuer son exécution.
- Au temps 3.5, la tâche T1 devient la plus prioritaire et est exécutée. Elle a terminé son exécution et sort de la liste des tâches en attente.
- Au temps 4, la tâche T3 redevient la plus prioritaire et est exécutée. Elle a terminé son exécution et sort de la liste des tâches en attente.
- Au temps 4, la tâche T4 devient la plus prioritaire et est exécutée. Elle a terminé son exécution et sort de la liste des tâches en attente.
- Au temps 4, la tâche T2 est la seule tâche restante dans la liste des tâches en attente, mais elle n'a plus de quantum à exécuter et est mise en attente jusqu'à ce qu'elle redevienne la plus prioritaire.

Exercice 2 : Mise en œuvre de fonctions d'une pile de protocole

On désire implanter les échanges des données entre 2 couches de niveau différent d'une pile de protocoles.
1. La couche N+1 fournit les données à émettre à la couche N. Une fonction « de Couche_N+1 » (void de Couche_N+1 ()) de la couche N permet récupérer ces données via un buffer, lorsque celui-ci n'est pas vide.
2. La couche N reçoit des données de la couche N-1. Après traitement ces données sont transmises à la couche N+1. La fonction « vers_Couche_N+1» (void vers_ Couche N+1 ()) exécutée par la couche
N insère les données dans un buffer si celui-ci n'est pas plein.

- Question 1 : Qu'est-ce qu'une tâche ? Quelles sont les différences entre un processus lourd et un processus léger?

Les deux fonctions « de_Couche_N+1 » et « vers_Couche_N+1 » sont exécutées dans des processus légers.

-Question 2: Ecrire, en langage C, les instructions permettant d'exécuter les fonctions « de Couche N+1 » et « vers Couche_N+1 » dans deux threads différents.
-**Question 1 :** Qu'est-ce qu'une tâche ? Quelles sont les différences entre un processus lourd et un processus léger?
Dans le contexte du cours de système d'exploitation, une tâche peut être considérée comme une unité d'activité exécutée par le processeur, généralement associée à un processus. Elle peut inclure plusieurs threads exécutant des tâches différentes au sein d'un même processus.
Un processus lourd est un processus qui utilise beaucoup de ressources système, telles que la mémoire et le processeur, pour effectuer ses tâches. Les processus

lourds peuvent ralentir le système en raison de la quantité de ressources qu'ils utilisent.
En revanche, un processus léger (aussi appelé thread léger) est une tâche qui partage les mêmes ressources qu'un processus mais qui utilise moins de ressources en raison de sa légèreté. Les threads légers sont souvent utilisés pour effectuer des tâches en arrière-plan, tels que des opérations d'entrée/sortie ou des opérations de traitement de données, sans perturber l'interaction de l'utilisateur avec le système. Ils peuvent également améliorer les performances en permettant une utilisation plus efficace du processeur.

Question 2 : Ecrire, en langage C, les instructions permettant d'exécuter les fonctions « de Couche N+1 » et « vers Couche_N+1 » dans deux threads différents.
Voici un exemple de code en langage C qui permet d'exécuter les fonctions "de_Couche_N+1" et "vers_Couche_N+1" dans deux threads différents :

```
#include <pthread.h>
void* de_Couche_N+1(void* arg) {
    // Récupération des données depuis la couche N+1
    // Ajout des données dans un buffer si celui-ci n'est pas vide
}
void* vers_Couche_N+1(void* arg) {
    // Récupération des données depuis la couche N-1
    // Traitement des données
    // Transmission des données à la couche N+1 en ajoutant les données
dans un buffer si celui-ci n'est pas plein
}
int main() {
    pthread_t thread_de_Couche_N+1, thread_vers_Couche_N+1;
    // Création des threads pour exécuter les fonctions "de_Couche_N+1" et
"vers_Couche_N+1"
    pthread_create(&thread_de_Couche_N+1, NULL, de_Couche_N+1, NULL);
    pthread_create(&thread_vers_Couche_N+1, NULL, vers_Couche_N+1, NULL);
    // Attente de la fin d'exécution des threads
    pthread_join(thread_de_Couche_N+1, NULL);
    pthread_join(thread_vers_Couche_N+1, NULL);
    return 0;
}
```

Ce code crée deux threads qui exécutent les fonctions "de_Couche_N+1" et "vers_Couche_N+1" respectivement, en utilisant la bibliothèque pthread. Les threads sont créés à l'aide de la fonction pthread_create et sont ensuite joints à l'aide de la fonction pthread_join pour attendre leur fin d'exéc

Question 3 : Proposez une implantation des fonctions « de_Couche_N+ l » et « vers_Couche_N+1 » en tenant compte du partage des buffers d'émission et de réception entre les couches de différents niveaux. Quelles modifications doit-on apporter à la structure de données Buffer?

la question 3, voici une proposition d'implémentation des fonctions « de_Couche_N+1 » et « vers_Couche_N+1 » en utilisant un buffer partagé entre les couches :

- La couche N+1 écrit les données dans un buffer d'émission partagé avec la couche N. Si le buffer est plein, elle attend qu'il se vide.
- La couche N lit les données du buffer d'émission et les traite. Elle écrit ensuite les données dans un buffer de réception partagé avec la couche N+1. Si le buffer est plein, elle attend qu'il se vide.
- La couche N+1 lit les données du buffer de réception et les transmet à la couche N+2 via la fonction « vers_Couche_N+2 ». Si le buffer est vide, elle attend qu'il se remplisse.

Pour la couche N+1, la lecture dans le buffer d'émission (les données sont écrites par la couche N via la fonction « vers_Couche_N+1 ») est effectué par la fonction de_Couche_N. Le buffer d'émission peut être rempli par plusieurs couches N.
Question 4 : Comment gèreriez-vous cette contrainte ?

la question 4

Pour gérer la contrainte du buffer partagé entre les couches de niveaux différents, il est important d'utiliser des mécanismes de synchronisation pour éviter les problèmes de concurrence. On peut par exemple utiliser des sémaphores pour signaler si le buffer est plein ou vide, et des mutex pour protéger l'accès concurrent aux buffers.

Question 5 : Puisque les processus légers semblent offrir de meilleures performances que les processus Unix tra-ditionnels, pourquoi les systèmes de type Unix ne sont-ils pas ré-écrits de sorte à ne plus utiliser que les processus

La question 5, les processus légers (ou threads) offrent en effet de meilleures performances que les processus Unix traditionnels car ils sont plus légers et plus rapides à créer et à détruire. Cependant, la gestion des processus légers peut être plus complexe, en particulier en ce qui concerne la synchronisation et la gestion des ressources partagées.

De plus, les processus Unix traditionnels offrent une isolation plus forte entre les différentes tâches du système, ce qui peut être important pour la sécurité et la fiabilité du système. En résumé, les processus légers sont une bonne solution pour les applications qui nécessitent une forte concurrence, mais les processus Unix traditionnels restent une solution plus adaptée pour les systèmes d'exploitation où la sécurité et la fiabilité sont des priorités.