Exam 2022

1) A quoi sert un système d'exploitation?

Un système d'exploitation (SE) est un logiciel qui permet à un ordinateur ou à un appareil électronique de fonctionner en gérant les ressources matérielles et logicielles de l'appareil. En résumé, le système d'exploitation sert à gérer les ressources de l'appareil, fournir une interface utilisateur, gérer les fichiers et les répertoires, exécuter des programmes et gérer les processus en cours d'exécution.

2) Qu'est-ce qu'une tâche ? Quelles sont les différences entre un processus lourd et un processus léger?

Une tâche, en informatique, est une unité d'activité ou de travail qui doit être accomplie par un programme ou un système. Elle peut être une commande, une opération, une demande de l'utilisateur, ou toute autre activité qui doit être exécutée par un ordinateur.

Un processus est un programme en cours d'exécution, qui peut être composé de plusieurs tâches ou threads. Les processus lourds sont des processus qui consomment beaucoup de ressources du système, tels que la mémoire ou le processeur. Ce sont souvent des applications complexes ou des programmes de traitement de données. Les processus légers, également appelés threads, sont des unités d'exécution plus légères qui partagent les mêmes ressources qu'un processus lourd, mais qui ont des contextes d'exécution distincts. Ils sont souvent utilisés pour des opérations simultanées et des tâches en temps réel, tels que les applications de traitement d'image ou de son en temps réel, ou les programmes qui doivent réagir rapidement aux entrées de l'utilisateur.

Les threads sont plus rapides et plus efficaces que les processus lourds, car ils consomment moins de ressources système et sont plus faciles à créer et à détruire.

3) L'ordonnanceur est un élément important des systèmes d'exploitation. Quel est son rôle?

L'ordonnanceur (ou scheduler en anglais) est un élément clé des systèmes d'exploitation. Son rôle est de gérer l'exécution des processus et des threads sur le système. Plus précisément, l'ordonnanceur est responsable de décider quel processus ou quel thread sera exécuté à un moment donné et pendant combien de temps. Il attribue des ressources telles que le temps CPU et la mémoire à chaque processus ou thread en fonction de certaines règles et priorités. L'objectif de l'ordonnanceur est d'optimiser l'utilisation des ressources système pour garantir un traitement efficace des tâches en cours d'exécution tout en minimisant les temps d'attente et en garantissant une réponse rapide aux entrées utilisateur.

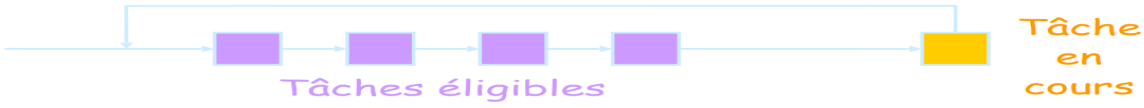
Si aucune tâche n'est en cours d'exécution, l'ordonnanceur choisit une tâche qui éligible qui est alors exécutée. Lorsque la tâche en cours d'exécution se termine, l'ordonnanceur choisit une nouvelle tâche

4) Qu'est-ce que le tourniquet (round-robin)?

Le tourniquet (ou round-robin en anglais) est un algorithme d'ordonnancement préemptif utilisé dans les systèmes d'exploitation pour partager le temps CPU entre plusieurs processus ou threads de manière équitable.

L'algorithme consiste à allouer des tranches de temps CPU égales à chaque processus ou thread en rotation, sans donner de priorité à aucun d'entre eux. Chaque processus ou thread est exécuté pendant un temps prédéfini appelé quantum, généralement de l'ordre de quelques millisecondes, avant d'être mis en attente et remplacé par le processus ou le thread suivant dans la file d'attente.

L'avantage de l'algorithme de tourniquet est qu'il assure un temps d'attente équitable pour chaque processus ou thread et évite les situations de starvation (ou famine) où certains processus ou threads peuvent être négligés au profit d'autres. Cependant, l'algorithme peut ne pas être adapté pour les tâches nécessitant une réponse rapide ou pour les systèmes à forte charge, car il peut entraîner une latence élevée et une inefficacité dans l'utilisation des ressources système.



2ème : .Le tourniquet (round-robin) est un algorithme de planification des processus dans les systèmes d'exploitation. Il permet une extension multi-tâche du PAPS, c'est-à-dire que plusieurs tâches peuvent être exécutées simultanément. Les tâches sont placées dans une file d'attente et exécutées chacune leur tour, de manière cyclique (d'où le nom "round-robin"). Cet algorithme est très

simple et équitable, car il garantit que chaque tâche a une chance égale d'être exécutée et évite qu'une tâche ne monopolise le processeur pendant trop longtemps. En résumé, le tourniquet permet une exécution équitable des tâches dans un système d'exploitation, en leur donnant un temps d'exécution limité et en alternant entre elles de manière cyclique.

5) Un utilisateur a écrit un sous-programme de temporisation sensé durer une minute. Pour cela, il a simplement utilisé une boucle à compteur de sorte à exécuter n instructions où n est tel que n/p = 60 sachant que p est le nombre d'instructions exécutées en une seconde par le micro-processeur. Lorsqu'il compare le fonctionnement de ce sous-programme avec sa montre. L'utilisateur s'aperçoit que le résultat n'est pas celui attendu. Pourquoi ? La temporisation exécutée par le sous-programme est-elle trop longue ou trop courte?

Prenons un exemple : supposons que le microprocesseur exécute 10 instructions par seconde (p=10). Le sous-programme de temporisation vise à exécuter des instructions tel que n/p = 60, c'est-à-dire n=600. Cela signifie que la boucle sera exécutée 600 fois, prenant donc 60 secondes pour s'exécuter, ce qui est exactement ce qui est voulu.
Cependant, le problème est que la boucle elle-même prend du temps pour s'exécuter. Par exemple, supposons que chaque itération de la boucle prend 0,01 seconde pour s'exécuter. Dans ce cas, la boucle de 600 itérations prendrait en réalité 6 secondes pour s'exécuter, pas 1 minute. Cela signifie que le temps d'attente réel sera plus long que le temps d'attente prévu.
En résumé, la boucle elle-même prend du temps pour s'exécuter, ce qui signifie que le temps de temporisation réel sera plus long que le temps de temporisation prévu.

En conséquence, il est peu probable que la temporisation produite par le sous-programme soit précise ou cohérente. Pour obtenir une temporisation fiable et précise, il est préférable d'utiliser des fonctions de temporisation spécifiquement conçues pour cette tâche.

Exercice 2: Virtualisation

Question 1 : Quels sont les intérêts et les inconvénients de la paravirtualisation ?

La paravirtualisation est une technique de virtualisation qui permet à plusieurs systèmes d'exploitation de s'exécuter sur un seul ordinateur physique, en partageant les ressources de manière efficace. Cette technique offre plusieurs avantages et inconvénients :

Intérêts de la paravirtualisation :

- Hautes performances : la paravirtualisation permet une utilisation efficace des ressources matérielles en évitant la surcharge liée à l'émulation de matériel virtuel, ce qui améliore les performances.
- Flexibilité : les systèmes d'exploitation virtualisés peuvent être facilement configurés et ajustés pour répondre aux besoins de l'application.
- Sécurité : la paravirtualisation peut permettre d'isoler les systèmes d'exploitation virtuels les uns des autres, offrant ainsi une sécurité renforcée.

Inconvénients de la paravirtualisation :

- Limitations matérielles : la paravirtualisation nécessite que le matériel soit compatible avec les instructions de virtualisation, ce qui peut limiter les choix de matériel.
- Complexité : la paravirtualisation peut être plus complexe à mettre en œuvre et à gérer que d'autres techniques de virtualisation, car elle nécessite des modifications dans le noyau du système d'exploitation.
- Dépendance de la plateforme : la paravirtualisation est généralement spécifique à une plateforme particulière, ce qui peut rendre la migration vers d'autres plateformes plus difficile.
-

Question 2 : Quels sont les avantages à utiliser les conteneurs Docker? A-t-on réellement un système virtualisé ?

Les avantages de l'utilisation de conteneurs Docker incluent :

- La portabilité : les conteneurs peuvent être exécutés sur n'importe quelle machine qui prend en charge Docker, ce qui facilite le déploiement et la gestion des applications.
- L'isolation : les conteneurs permettent d'isoler les applications et leurs dépendances, ce qui permet de garantir que chaque application fonctionne de manière cohérente et prévisible, sans interférence avec les autres applications.

- L'efficacité : les conteneurs partagent le même système d'exploitation et ne nécessitent pas de ressources supplémentaires pour chaque application, ce qui permet d'économiser de l'espace de stockage et des ressources informatiques.
- La rapidité : les conteneurs peuvent être créés, déployés et exécutés rapidement, ce qui permet de réduire le temps de déploiement et de mise à jour des applications.
- La flexibilité : les conteneurs peuvent être facilement configurés et personnalisés pour répondre aux besoins spécifiques de chaque application.

Oui, Docker utilise une forme de virtualisation appelée virtualisation de conteneurs pour créer des environnements d'exécution isolés pour les applications.

Exercice 3: Micro-noyaux

Question 1: Quels sont les avantages des micro-noyaux ? A quoi servent les IPC?

Les avantages des micro-noyaux sont les suivants :

- La stabilité : les services essentiels sont isolés dans des espaces protégés, réduisant ainsi la possibilité de plantage du système.
- La modularité : les services sont indépendants les uns des autres, ce qui permet une plus grande flexibilité dans la configuration du système.
- La sécurité : en minimisant le noyau, les attaques potentielles peuvent être limitées aux services utilisateur et non au noyau lui-même.
- La facilité de maintenance : la modularité du système facilite la mise à jour ou le remplacement des composants.

Les IPC (Inter-Process Communication) sont utilisés dans les micro-noyaux pour permettre la communication entre les différents services exécutés sur le système. Les IPC permettent aux processus de partager des données et des ressources, de s'envoyer des messages et de synchroniser leur exécution. Les IPC sont généralement plus rapides et plus efficaces que les mécanismes de communication entre processus utilisés dans les noyaux monolithiques, car ils ne nécessitent pas de passage par le noyau pour transférer les données

Question 2 : Qu'est-ce que le TCB? A quoi sert le buffer dans le TCB des threads du micro-noyau L4?

Le TCB (Thread Control Block) est une structure de données utilisée pour représenter chaque thread d'exécution dans un système d'exploitation. Le TCB contient des informations sur l'état du thread, telles que les registres de processeur, la priorité, l'état de la sémaphore, etc. Il contient également des informations sur les ressources allouées au thread, telles que les blocs de mémoire alloués, les fichiers ouverts et les connexions réseau.

Dans le micro-noyau L4, le buffer dans le TCB des threads est utilisé pour stocker les messages envoyés entre les threads via les IPC. Le buffer est une zone de mémoire réservée dans le TCB pour stocker les messages en attente d'être reçus par le thread. Lorsqu'un message est envoyé à un thread, il est placé dans le buffer correspondant du TCB. Le thread peut ensuite accéder au message en lisant le contenu du buffer. Le buffer permet d'éviter de devoir copier les données entre différents espaces d'adressage lors de la communication entre les threads, ce qui améliore les performances et réduit la complexité du système.

EX4: Systèmes d'exploitation pour l'IoT

Question 1 : Les systèmes d'exploitation pour l'IoT doivent avoir une empreinte mémoire faible. Expliquez pourquoi?

Consommation d'énergie : Les appareils IoT sont souvent alimentés par des batteries ou des sources d'énergie limitées. Les SE pour l'IoT doivent être conçus pour minimiser la consommation d'énergie, car l'exécution de processus et de services inutiles peut rapidement vider la batterie.

Coût : Les appareils IoT sont souvent fabriqués en grandes quantités à faible coût. En conséquence, le coût de chaque composant doit être maintenu aussi bas que possible. Les SE pour l'IoT doivent être conçus pour fonctionner sur des appareils peu coûteux, ce qui signifie qu'ils doivent avoir une empreinte mémoire faible pour ne pas augmenter le coût du matériel.

Limitations matérielles : Les appareils IoT ont souvent des ressources matérielles limitées, notamment en termes de mémoire vive (RAM) et de stockage. Par conséquent, les SE pour l'IoT doivent être conçus pour fonctionner efficacement avec ces ressources limitées.

Question 2: Ecrire en langage C pour le système d 'exploitation Contiki le code du thread light_collect qui émet une requête de luminosité périodiquement toutes les 10 ms en utilisant le protocole COAP.

```
PROCESS_THREAD(light_collect_process, ev, data) {
 PROCESS_BEGIN();
  static coap_message_t request;
  coap_init_message(&request, COAP_TYPE_CON, COAP_POST, 0);
  coap_set_header_uri_path(&request, RESOURCE_URI);
  while(1) {
    etimer_set(&sample_timer, SAMPLE_INTERVAL);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&sample_timer));
    uint16_t light = board_light_sensor_convert_single();
    char payload[16];
    snprintf(payload, sizeof(payload), "%u", light);
    coap_set_payload(&request, (uint8_t *)payload, strlen(payload));
    COAP_BLOCKING_REQUEST(&server_ep, &request, receipt_func);
 coap_set_payload(&request, NULL, 0);
 }
  PROCESS_END();
}
```

Question 3 : Dans contiki, le capteur de luminosité se nomme opt 3001 _sensor. Une lecture de la luminosité se fait en activant le capteur via la commande SENSORS_ACTIVATE(). Lorsque la lecture est terminée, un événement sensors_changed est émis. La fonction value sans parámêtre associée au capteur permet de récupérer la valeur. Ecrire en langage C, le code du thread local_light Contiki qui lit périodiquement la luminosité ambiante toutes les 10ms.

```
#include "contiki.h"
#include "dev/sensors.h"
#include "ti-lib.h"
PROCESS(local_light, "Local Light");
AUTOSTART_PROCESSES(&local_light);
PROCESS_THREAD(local_light, ev, data)
{
  static struct etimer timer;
  static int light;
  PROCESS_BEGIN();
  SENSORS_ACTIVATE(opt_3001_sensor); // activer le capteur OPT-3001
  while(1) {
    /* Attente pendant 10ms */
    etimer_set(&timer, CLOCK_SECOND / 100);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));
    /* Lecture de la luminosité */
    light = opt_3001_sensor.value(0);
    /* Traitement de la luminosité lue */
    printf("Luminosite locale : %d lux\n", light);
  }
  PROCESS_END();
}
```

Question4 : Ecrire Hes quelques lignes de code qui permettent de créer les 2 threads light_collect et local_1ight et qui lancent ces 2 threads

```
#include "contiki.h"
#include "coap-engine.h"
#include "sys/etimer.h"
#include "dev/sensors.h"
```

```
#include "dev/opt-3001-sensor.h"

// Déclaration des fonctions thread
PROCESS(light_collect, "Light Collect Thread");
PROCESS(local_light, "Local Light Thread");

// Définition des événements pour les timers
#define LIGHT_COLLECT_INTERVAL_MS 10
static struct etimer light_collect_timer;

// Fonction de réception des messages CoAP
void receipt_func(coap_message_t *response) {
  const unsigned char *data;
  if (response == NULL) {
    puts("Request timeout");
    return;
  }
  int len = coap_get_payload(response, &data);
  printf("light: %s lux\n", (char *)data);
  coap_set_payload(response, NULL, 0);
}

// Thread light_collect qui collecte la luminosité avec le capteur OPT-3001
PROCESS_THREAD(light_collect, ev, data) {
  PROCESS_BEGIN();

  // Activation du capteur de luminosité
  SENSORS_ACTIVATE(opt_3001_sensor);

  // Boucle infinie de collecte de la luminosité toutes les 10ms
  while (1) {
    // Initialisation du message CoAP
    coap_message_t request;
    coap_init_message(&request, COAP_TYPE_CON, COAP_POST, 0);
    coap_set_header_uri_path(&request, "/light");

    // Envoi de la requête CoAP et attente de la réponse
    COAP_BLOCKING_REQUEST(&server_ep, &request, receipt_func);

    // Attente de l'intervalle de temps
    etimer_set(&light_collect_timer, CLOCK_SECOND * LIGHT_COLLECT_INTERVAL_MS / 1000);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&light_collect_timer));
  }

  PROCESS_END();
}

// Thread local_light qui lit la luminosité ambiante avec le capteur OPT-3001
PROCESS_THREAD(local_light, ev, data) {
  PROCESS_BEGIN();

  // Activation du capteur de luminosité
  SENSORS_ACTIVATE(opt_3001_sensor);

  // Boucle infinie de lecture de la luminosité toutes les 10ms
  while (1) {
    // Lecture de la valeur de luminosité du capteur
    int light_value = opt_3001_sensor.value(0);
```

```
    // Affichage de la valeur de luminosité
    printf("local light: %d lux\n", light_value);

    // Attente de l'intervalle de temps
    etimer_set(&light_collect_timer, CLOCK_SECOND * LIGHT_COLLECT_INTERVAL_MS / 1000);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&light_collect_timer));
  }

  PROCESS_END();
}

// Fonction principale
int main(void) {
  // Initialisation du système d'exploitation Contiki
  PROCESS_INIT();

  // Démarrage des threads light_collect et local_light
  process_start(&light_collect, NULL);
  process_start(&local_light, NULL);

  // Lancement du système d'exploitation Contiki
  PROCESS_RUN();

  // Sortie de la fonction principale
  return 0;
}
```

Avantages des micro-noyaux

Adaptation au besoin :

- Les serveurs peuvent être configurés pour être adaptés au matériel cible (petits systèmes embarqués, PC, systèmes multiprocesseurs, ...)
- Ajout/Suppression des serveurs au besoin

Meilleure gestion des accès au système :

- Interface bien définie entre les composants (IPC)
- Pas d'accès à ces composants en dehors de ces interfaces
- Evolution/mise à jour facilitée

Meilleures protections contre les erreurs :

- Erreur d'un composant ne crache pas tout le système
- Plus sécurisée via une protection inter-composant

Sous-système de protection / isolation