Mini-projet : Noyau

17 avril 2013
Rendu: 31 mai 2013

En dehors de la description ci-dessous, seules les informations complémentaires données sur Moodle font foi (pas les informations données oralement).
Pour tout point qui vous semble ambigu dans l'énoncé, vous pouvez décider de la façon de lever l'ambiguité (dans ce cas, veuillez documenter votre choix).

Le projet s'effectuera par groupes de 2 étudiants au maximum. ***La conception de la solution et son implémentation seront faites séparément dans chacun des groupes, et ne seront pas partagées entre les groupes.***

Coefficient: 20% de la note finale

I Introduction

L'objectif du projet est de réaliser sur la carte FPGA4U un noyau en C implémentant les **conditions de Java**. Une priorité sera associée aux processus. Cette priorité sera prise en compte **uniquement** dans l'allocation du CPU (ce qui n'est fait en Java que pour les tâches temps réel).[1] Le noyau sera réalisé à l'aide des fonctions `newProcess(...)` et `transfer(...)` (étape 1) ainsi que `ioTransfer(...)` (étape 2), mises à disposition.[2] Ces fonctions se trouvent dans les fichiers `system_m.h` et `system_m.c`, qui utilisent les fichiers suivants: `assembly.h`, `asm.s`, `interrupt.h` et `interrupt.c`. Tous ces fichiers peuvent être téléchargés à partir de Moodle (fichier `fpga.zip`).

II Déroulement du projet et rendu

Le projet est décomposé en 2 étapes. Dans l'étape 1, les interruptions seront ignorées. Elles seront prises en compte dans l'étape 2.

Le fichier `kernel.h`, qui contient les définitions des fonctions du noyau, vous est fourni et ne devra pas être modifié. Votre implémentation du noyau sera placé dans un fichier `kernel.c`.

Le rendu se fera à la fin de l'étape 2. Tous les fichiers seront regroupés dans un seul fichier `zip`, nommé avec les noms de famille (sans accent) des membres du groupe. Le fichier zip contiendra les fichiers `kernel.h` et `kernel.c`, plus d'autres éventuels fichiers sources. Le non respect de ces consignes conduira à une pénalité dans la note. En cas de doute, utilisez le forum.

Dates:

- Enoncé du projet et démarrage de l'**étape 1**: 17 avril 2013
- Démarrage suggéré pour l'**étape 2** : 1er mai 2013
- Rendu des étapes 1 et 2 : 31 mai 2013.
  Le lien pour déposer le fichier `zip` sera fournit ultérieurement sur Moodle.

[1] La priorité sera donc ignorée dans la gestionn des différentes queues d'attente.
[2] L'exercice 2 de la série 6 vous a permis de vous familiariser avec la carte FPGA4U.

Conseil important: Développez votre noyau pas à pas (par exemple fonction par fonction), et testez chaque pas au fur et à mesure. C'est la seule façon de réduire le temps nécessaire à la réalisation du noyau. Cela nécessite bien évidemment de réfléchir à l'ordre de développement, et aux tests intermédiaires à réaliser.
Remarque: Le debugger ne peut être utilisé. Pour tracer une exécution, vous pouvez utiliser les leds ou `printf()`. Attention, l'affichage de `printf()` est asynchrone. Cela signifie qu'une erreur dans votre code qui survient *après* l'appel de `printf()` peut empêcher l'affichage de `printf()`.

III Etape 1 du projet

Dans l'étape 1, le noyau fournira les fonctions suivantes:

- `void creerProcessus(void (*f), int stackSize, int priority)`
  Crée un processus de priorité spécifiée, alloue le descripteur de processus et la pile du processus.[3]

  La priorité des processus prendra les valeurs 1 ou 2, avec 1 la priorité minimale et 2 la priorité maximale. Une priorité supérieure à 2 sera considérée comme égale à 2, et une priorité inférieure à 1 sera considérée comme égale à 1. Le CPU ne sera alloué à un processus de priorité 1 que si aucun processus de priorité 2 n'est prêt à être exécuté.
- `void start()`
  Fonction à appeler après la création de tous les processus (cf. cours).
- `int creerVerrou()`
  Alloue le descripteur de verrou et retourne l'identificateur du verrou créé.
- `int creerCondition(int verrouID)`
  Alloue le descripteur de condition et l'associe au verrou passé en paramètre; retourne l'identificateur de la condition créée.
- `void verrouiller(int verrouID)`
- `void deverrouiller(int verrouID)`
- `void await(int conditionID)`
- `void signal(int conditionID)`
- `void signalAll(int conditionID)`

L'inversion de priorité sera résolu par la technique de *priorité plafond*. On supposera que toute section critique est à la priorité 2.

Autres indications

- Pour simplifier, on vous conseille fortement de n'utiliser que des structures de données statiques: introduisez une constante pour le nombre maximum de processus qui peuvent être créés, une constante pour le nombre maximum de verrous qui peuvent être créés et pour le nombre maximum de conditions par verrou. Identifiez ces entités par des entiers positifs. Afficher un message d'erreur si lors de la création d'une de ces entités le maximum est dépassé (et ne pas retourner de l'appel).

  De manière générale, toute erreur détectée à l'exécution conduira à afficher un message avant de terminer l'exécution.
- N'oubliez pas de gérer les appels réentrants (depuis une section critique protégée par un verrou `v`, réentrer dans cette même section critique) et imbriqués (depuis une section critique protégée par un verrou `v`, entrer dans une section critique protégée par un autre verrou `v'`).

Test

Le noyau de l'étape 1 sera testé avec l'application suivante, fournie dans le fichier `applicationEtape1.c`, qui utilise trois (des quatre) boutons, numérotés de 0 à 2 (0 est le bouton le plus à l'extérieur de la carte):

[3] `CreerProcessus(...)` appellera `newProcess(...)`. Le 2ème paramètre de `newProcess(...)` (adresse de la pile) est l'adresse retournée par l'appel à `malloc(...)`.

- L'application contient trois *tampons*, numérotés de 0 à 2. Chaque tampon permet de déposer et de prélever un message (un message est un entier).
- L'application comporte un processus *producteur* de priorité 1 qui lit les boutons en mode attente active. Chaque fois que le bouton i a été pressé, le producteur dépose un message dans le tampon i (la valeur déposée est quelconque).
- L'application comporte trois processus *consommateurs* numérotés de 0 à 2, de priorité 2. Le consommateur i prélève des messages du tampon i. A chaque consommateur est associé une led. Chaque fois qu'un consommateur a prélevé un message, il allume la led qui lui est associée pour une certaine durée D, avant de prélever le message suivant.

IV Etape 2 du projet

L'étape 2 ajoutera la mesure du temps au noyau de l'étape 1. Pour cela, la fonction suivante sera ajoutée au noyau:

- `int timedAwait(int conditionID, int time)`
  Identique à `await()`, avec un timeout égal à `time` (exprimé en millisecondes). Retourne `0` si le timeout a expiré, `1` sinon.

  Si le paramètre `time` est `0`, l'appel sera considéré comme équivalent à `await()`.

Pour l'étape 2, on vous fournit les deux fichiers supplémentaires `interrupt.h` et `interrupt.c` avec les fonctions suivantes:

- `void init_clock()`: Cette fonction enclenche les interruptions de l'horloge. Les interruptions de l'horloge ont lieu à chaque milliseconde. La fonction doit être appelée dans le noyau.
- `void maskInterrupts()`: Cette fonction masque toutes les interruptions. Elle doit être appelée à l'entrée de toute fonction du noyau.
- `void allowInterrupts()`: Cette fonction autorise les interruptions. Elle doit être appelée à la sortie de toute fonction du noyau.

Un exemple montrant l'utilisation de `init_clock()` et des interruptions de l'horloge peut être téléchargé depuis Moodle. Il s'agit du programme `testClockInterrupts.c`.

Test

Le noyau de l'étape 2 sera testé avec l'application suivante, fournie dans le fichier `applicationEtape2.c`, qui utilise trois (des quatre) boutons, numérotés de 0 à 2 (0 est le bouton le plus à l'extérieur de la carte):

- L'application contient trois *tampons*, numérotés de 0 à 2. Chaque tampon permet de déposer et de prélever un message (un message est un entier).
- L'application comporte un processus *producteur* de priorité 1 qui lit les boutons en mode attente active. Chaque fois que le bouton i a été pressé, le producteur dépose un message dans le tampon i (la valeur déposée est quelconque).
- L'application comporte trois processus *consommateurs* numérotés de 0 à 2, de priorité 2. Le consommateur i prélève des messages du tampon i. A chaque consommateur est associé une led. Chaque fois qu'un consommateur a prélevé un message, il fait clignoter la led qui lui est associée, avant de prélever le message suivant. Toutefois, un consommateur est bloqué au plus une seconde. Si aucun message n'est reçu, le consommateur allumera sa led pour une durée D' plus grande que D.