Projet de simulation du comportement de l’algorithme d’ordonnancement.

Thomas Kling et Guillaume Lechantre

## Définition du sujet

Il nous a été demandé de modéliser le fonctionnement d’un algorithme sur le modèle de l’algorithme **Round Robin** avec priorité.

Nous devions donc simuler des processus et les insérer dans des listes d’attentes en fonctions de leur priorité.

Le processus simulant le CPU pioche, à chaque quantum, le processus qu’il va exécuter dans la liste d’attentes correspondant à la priorité actuelle (N).

Une fois le quantum de temps terminé, si le processus est terminé, il est supprimé, sinon, on le remet dans la liste d’attente de priorité N-1.

## Nos choix techniques

# Threads vs Forks

Nous avons choisi d’utiliser des threads et non des forks.

Tout d’abord car la communication entre les threads est plus simple qu’entre les forks puisqu’ils partagent le même contexte, on est donc pas obligé d’utiliser des IPC, par exemple la priorité courante est une variable globale mis-à-jour par la table d’allocation et lu par le CPU.

De plus, contrairement aux forks, les threads sont supprimés automatiquement à la fin de l’exécution du processus qui les a initiés.

Ainsi, si notre programme venait à planter, nos threads sont automatiquement supprimés et on ne risque pas d’avoir un processus zombie qui va lire nos files de messages.

Dans le cas contraire, si nous avions utilisé des forks, nous aurions dû les tuer à la main si notre programme venait à planter.

# Files de messages vs Pipes

Nous avons choisi d’utiliser des files de messages et non des pipes.

Les files de messages se prêtent le mieux à l’algorithme Round Robin puisqu’on a un déjà un système de FIFO et de type pour les messages.

Dans la file de messages processus exécution nous avons donné au message un type correspondant à leur priorité.

# Sémaphores

Pour synchroniser la soumission des processus, leur mise en place puis leur exécution nous avons mis en place deux sémaphores. De plus, elles permettent de s’assurer que la variable globale de priorité courante n’est pas lue et changer « en même temps ».

## La vie d’un processus

Une fois créé, le processus est mis dans la file de message processus entrant par le générateur de processus.

Le processus à ensuite un PID qui lui est attribué et est mis dans la file processus exécution avec comme type sa priorité.

Lorsque que c’est son tour, le CPU le sort de la file de message et l’exécute durant un quantum de temps. S’il est terminé à la fin du quantum, il est supprimé, sinon il est remis dans la file processus exécution avec comme type son ancienne priorité +1. Cela va baisser sa priorité de un.

# Les différents modules

Nous avons choisi de découper le projet en plusieurs modules en fonction de leurs rôles. Tout d’abord, nous avons un module CPU va simuler le processeur et exécuter les processus. Ensuite, la table d’allocation va lire les processus entrants, les ajouter aux files de processus à exécuter et calculer la nouvelle priorité en fonction du pas. Ensuite, on a un module GénérateurProcessus qui permet de créer les nouveaux processus au démarrage en lisant un fichier puis de générer aléatoirement des nouveaux processus. Le main est réduit à créer les threads, les IPC et lire les paramètres, c’est vraiment l’entrée du programme. Enfin, on a un module utils, qui possède les constantes globales du projet, un module MySemaphore qui reprend les sémaphores qu’on a vu en cours et qui permet une utilisation simple et intuitive partout dans le programme.

**Main.c**

1. Récupère les paramètres :

* Durée d’un quantum
* Le pas

1. Fermeture si nécessaire et ouverture de 2 files de messages :

* File processus entrants
* File processus exécutions

1. Création de deux sémaphores : 0 et 1
2. *readFile()*
3. Lancement de *start\_cpu(dureeQuantum)* dans un thread
4. Lancement de *start\_alloc(pas)* dans un thread
5. Attente de la fin des threads
6. Fermeture des files de messages et des sémaphores

**TableAllocation.c**

**addProcess(processus)**

Si le processus est nouveau, on lui attribue un pid.

Sinon on augmente sa priorité de un.

On ajoute le processus dans la file processus exécution.

**start\_alloc(t)**

Tant qu’il a des processus à exécuter :

1. Attend activation du sémaphore 0
2. Regarde s’il y a des processus dans la file processus entrants.

S’il y en a : *addProcess(processus)*

1. *computeNewPriority()*
2. Active le sémaphore 1

**computeNewPriority()**

Met à jour la priorité.

Les probabilités sont, de la table 1 à 10 : 26 ;20 ;16 ;12 ;9 ;7 ;5 ;3 ;2 ;1

**CPU.c**

**start\_cpu(pas)**

Pour chaque quantum de temps :

1. *soumission()*
2. Attend activation du sémaphore 1
3. Regarde dans la file de priorité courante s’il y a un processus.

S’il n’y en a pas il va dans la file suivante.

1. Exécute le processus récupéré durant le quantum.
2. Si le processus est terminé on le retire. S’il n’était pas terminé, on le remet dans la file des processus entrants.

**GenerateurProc.c**

**readFile()**

1. Génération de processus à partir du fichier proc.txt
2. Ces processus sont mis dans la file des processus entrants

**soumission()**

1. Génération de processus (paramètres aléatoires)
2. Soumission ne crée pas de processus à chaque fois, la probabilité d’en créer est de 0,2.
3. Une fois créé, le processus est mis dans la file processus entrants
4. Active le sémaphore 0

**Utils.h**

Défini la structure d’un processus :

* Priorité : long
* myPid\_t : pid
* tempsExec : int
* dateSoumission : int