Rapport de Projet SR70

« simulation du ROUND ROBIN »

Alexandre Delahousse - Hugo LAUREN

31/11/2019

1. Définition du sujet

Il s'agit de reproduire le comportement de l'algorithme d'ordonnancement de processus vu en cours. Cet algorithme s'appuie sur le modèle du Round Robin avec priorités considérant qu'à chaque priorité est associée un tourniquet. Le déroulement de l'algorithme est le suivant : chaque processus, identifié par un numéro unique, possède une priorité de 1 à 10 (1 étant considéré comme le plus prioritaire), une date de soumission et un temps d'exécution. Le système crée alors un tourniquet par priorité qui correspond pour chaque priorité à une liste d'attente contenant les processus idoines. Durant chaque quantum de temps, on exécute un processus d'une certaine priorité en fonction de la table d'allocation CPU. Si la priorité du prochain processus devant s'exécuter est équivalente à la priorité du processus en cours d'exécution, on diminue la priorité du processus en cours d'exécution et on traite le nouveau processus. Dès lors, le processus réquisitionné est inséré à la fin du tourniquet correspondant à sa nouvelle priorité. Si un tourniquet cible relatif au choix opéré est vide alors le processus du tourniquet de priorité directement inférieure est élu. Dans le cas où le tourniquet de priorité 10 est choisi et que celui-ci est vide alors le choix portera sur le tourniquet de priorité 1.

2. Objectifs du projet

1. Créer une table d'allocation CPU qui pourra être modifiée par l'utilisateur ;
2. Créer aléatoirement les processus en tenant compte des spécificités propres à la simulation ;
3. Créer les structures propres à la simulation en respectant les principes de synchronisation et de communication vus en cours ;
4. Simuler le comportement du système ;
5. Être en mesure de modifié la valeur de quantum et garantir un fonctionnement optimal de l'algorithme.

3. Solutions proposées

Notre programme se décompose en deux grandes parties, la partie réalisée par le générateur et la partie réalisée par le processeur, les deux basée sur la table d'allocation CPU et communiquant grâce à une file de message.

Ces deux parties représentent les deux principaux processus (fork()) hérité de notre main.

3.1. Création de la table d'allocation CPU

Pour simuler la table d'allocation CPU, nous avons écrit dans un fichier un tableau où la première colonne désigne le quantum de temps et la deuxième colonne désigne la propriété qui lui est associée.

Après cela nous avons créé une méthode « lectureTableCPU » qui lit ce fichier et renvoie un tableau d'entier contenant les valeurs des propriétés. Les quantums de temps correspondent à l'index du tableau.

Nous avons choisi cette méthode car c'est à l'utilisateur de rentrer les valeurs de la table d'allocation CPU. Il lui suffit de remplir le fichier avec les données qu'il souhaite.

3.2. Les files de messages

Pour communiquer entre le générateur et le processeur nous avons créé une file de message. Elle peut être assimilé comme une file d'attente où chaque processus attend son tour pour s'exécuter. Celle-ci permet d'une part au générateur d'envoyer les processus créer et d'autre part elle permet également au processeur de récupérer le processus « élu » et de l'ajouter à la fin de la file s'il n'est pas terminé.

La recherche du processus « élu » ce fait à l'aide de la table CPU. Cette table nous indique qu'elle est la priorité du processus à exécuter au quantum de temps actuel. C'est pourquoi la recherche dans la file d'attente se fait en cherchant le premier processus qui possède cette priorité (cela correspond à l'entête), puisque les processus sont ranger en mode FIFO (premier créé, premier dans la fille d'attente).

3.3. Partie Générateur

Tout d'abord nous avons simulé un processus par une structure qui se nomme « processus » et qui a comme attributs

- une priorité de type long ;
- un temps d'exécution de type int ;
- une date de soumission de type int ;
- un pid de type int.

Ensuite, la création des processus se fait par l'instanciation de notre structure « processus » mais également par le biais d'un fork(), tout cela est réalisé par le générateur (étant lui-même un processus). Grâce à celui-ci, un nombre aléatoire de processus est créé à chaque quantum de temps avec une priorité et un temps d'exécution aléatoire également. Ces processus peuvent être également créer via un fichier de données d'entrées ;

Une fois le processus créer, le générateur envoie la structure du processus dans la file de message

1.1. Partie Processeur

Le processeur quant à lui, élit le processus à exécuter suivant la valeur de la table CPU et du quantum actuel et le fait s'exécuter. De plus, une fois la valeur du quantum terminé, le processeur remet le processus à la fin de la file de message si ce processus n'est pas terminé.

1.2. Synchronisation

Suite à la création du générateur et du processeur, nous avons dû synchroniser ces deux processus puisque les deux doivent s'exécuter pendant le même quantum de temps. Pour cela nous avons utilisé deux sémaphores. Notre but étant de bloqué l'un jusqu'à que l'autre se termine et inversement, sachant que le générateur doit s'exécuter avant le processeur car il faut déjà remplir la file avant de la traiter. Le rôle principal de ces sémaphores et de dire qui peut ou ne peut pas s'exécuter.

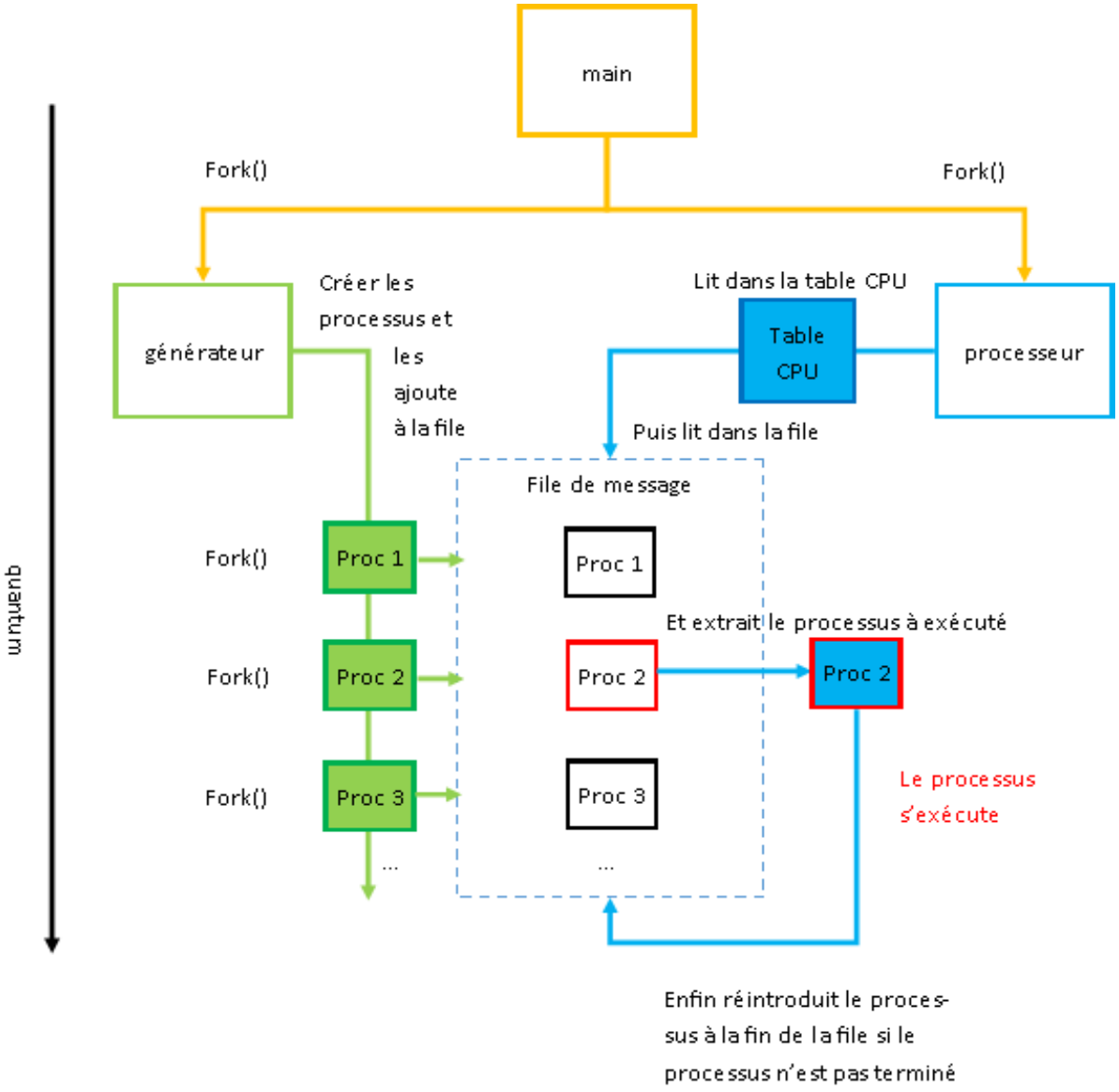
Nous pouvons donc représenter cette synchronisation par ceci :

Générateur	Processeur
Q0	P(semaphore1) Verrouillage du processeur
Exécution du générateur	
V(semaphore1) Déverrouillage du processeur	
P(semaphore0) Verrouillage de générateur	
	Exécution du processeur
	V(semaphore0) Déverrouillage du générateur
	P(semaphore1) Verrouillage du processeur
Q1 …	

1.3. Déroulement du programme

Au début on verrouille le processeur pour qu'il ne puisse pas lire dans la file puisqu'elle est encre vide. C'est donc au générateur de s'exécuter et donc de créer et d'ajouter les processus dans la file de message. Maintenant le générateur déverrouille le processeur et se verrouille. Là le processeur détermine quel processus doit s'exécuter grâce à la table CPU. Un processus est donc « élu » et c'est que lorsque que le quantum est terminé, que le processeur doit remettre le processus « élu » à la fin de la file de message durant le quantum suivant. Cependant cela doit se faire uniquement après la création des nouveaux processus (puisque le quantum à changer). Donc une fois que le processus est « élu », il s'exécute et à la fin du quantum on déverrouille le générateur et on verrouille le processeur (pour qu'il attend le générateur pour lire dans la file). Comme le processeur est déverrouillé, il génère les processus et seulement là déverrouille le processeur pour qu'il puisse ajouter à la file le processus « élu » au quantum suivant, et ainsi de suite.

Nous pouvons représenter le déroulement de notre programme par ce schéma :



1.4. Choix de départ

Au début du programme nous demandons à l'utilisateur de choisir s'il veut lancer le programme via des données d'entrées lu dans un fichier, ou alors ajouter en plus de cela des processus créés dynamiquement par le biais du générateur.

La création d'un fichier de données d'entrées peut permettre de vérifier le résultat obtenu.

1.5. Modification du quantum de temps

Après le choix de départ, nous demandons à l'utilisateur de saisir la valeur du quantum de temps. Cette valeur permet de modifier la période à laquelle le processeur s'exécute, incluant la lecture de la table d'allocation CPU. Exemple, si la valeur du quantum est de 3, le processeur lit dans la table CPU tout les 3 quantums, mais fait également exécuter le processus « élu » pendant 3 quantums maximum. C'est-à-dire si le processus possède un temps d'exécution

de 2, le quantum restant sera perdu puisque pendant un quantum de temps, un processus s'exécute.

1.6. Problèmes rencontrés

Nous avons rencontré un problème lorsque nous devions récupérer le processus à exécuter dans la file. Nous n'arrivions pas à retourner le processus « élu » pour ensuite le faire exécuter. Pour pallier à ce problème nous avons créé une deuxième file de message « streamFile ». Maintenant la fonction qui permet de récupérer le processus dans la file d'attente le pousse directement dans l'autre file. A son tour la fonction qui permet d'exécuter le processus lit dans la nouvelle file et le fait s'exécuter.

2. Conclusion

Tous les objectifs ont été réalisé que ce soit la table d'allocation CPU qui peut être modifiée par l'utilisateur ; la création aléatoirement des processus en tenant compte des spécificités propres à la simulation ;la création des structures propres à la simulation en respectant les principes de synchronisation et de communication vus en cours ; la simulation du comportement du système et enfin la possibilité de modifié la valeur de quantum et garantir un fonctionnement optimal de l'algorithme.