Manuel du développeur

Projet de système d’exploitation

Le but de ce projet est de simuler des cas de section critique. Ces cas sont présentés en sortie du programme, en fonction d’un fichier entré par l’utilisateur, qui contient les sémaphores et leurs initialisations, ainsi que des instructions. Notre code source est écrit en Java. Nous allons expliquer de manière générale le fonctionnement de notre programme, en décrivant les différentes classes que nous avons implémentées.

Carla MARTIN

Alexandre PACHOUD

# fonctionnement Lecture du fichier et gestion des données

L’utilisateur passe un fichier en argument sur la ligne de commande. Celui-ci doit être découpé en section commençant par le caractère ‘%’. L’algorithme reconnait ces différentes sections et exécute les portions de code qui leurs sont associées. Ce fichier contient impérativement les sections %IN (Initialisation) et %PA (Paramètres). Dans la section %IN, les sémaphores sont déclarés et initialisés à une certaine valeur. Dans la section %PA est indiqué le nombre de processus dans chaque corps de section critique (lecture L, écriture E, exécution X), ainsi que le nombre total de simulations à effectuer. Viennent ensuite les sections %PL, %EL, %PE, %EE, %PX, %EX, qui listent les différentes instructions pour chaque corps de section critique (L, E ou X).

Pour gérer les données contenues dans ce fichier, nous avons implémenté trois classes d’objets : SemaphoreObject, Instruction, Processus, Lecture, Memory, Simulation, Semaphore(main).

SemaphoreObject.class

Les sémaphores sont l’élément central du programme. Ils permettent de bloquer ou non un processus et détecter les cas de section critique. On les identifie grâce à une lettre. Leur structure est la suivante :

|  |  |
| --- | --- |
| SEMAPHOREOBJECT | |
| Champs | |
| char id | Une lettre majuscule qui correspond à l’identifiant du sémaphore |
| int compteur | Un nombre variable qui correspond à la « quantité » de sémaphore |
| Processus [] list | Un tableau de processus bloqués par le sémaphore s’il est à 0 (initialement vide) |
| Constructeurs | |
| public SemaphoreObject(char id, int compt, Processus [] list) | Initialise l’identifiant à id, le compteur à compt et le tableau des processus bloqués à list |
| public SemaphoreObject(char id) | Initialise seulement l’identifiant, les autres champs sont mis à 0 |
| Méthodes | |
| public void addBLOQUE (Processus P) | Ajoute un processus à la liste des processus bloqués, quand le compteur du sémaphore est à 0 et qu’un processus essaie de prendre le sémaphore |

Nous créons les sémaphores dans la section %IN. Le compteur est initialisé au nombre qui suit leur identifiant et le tableau de processus est vide.

Processus.class

Dans la section %PA, nous créons un tableau dont les cellules sont des Processus. L’ordre des processus dans ce tableau n’a pas d’importance, car les processus sont tirés aléatoirement.

Les processus ont la structure suivante :

|  |  |
| --- | --- |
| PROCESSUS | |
| Champs | |
| char type | L, E ou X  Un processus est associé à un type de section critique |
| Instruction instru | Prochaine instruction à exécuter en cas d’exécution (initialement null, elle est utilisée dans la partie simulation, nous y reviendrons) |
| int id | Identifiant unique qui permet de différencier les processus. |
| Constructeurs | |
| public Processus() | initialise tous les champs à leur valeur nulle |
| public Processus(char type,Instruction I,int i) | Initialise les champs avec les valeurs passées en argument |

Instruction.class

Dans les sections %PL, %EL, %PE, %EE, %PX, %EX, nous traitons les différentes instructions possibles. Nous avons choisi d’organiser les instructions en 3 types. Notons S un sémaphore, le premier type d’instruction est P(S), que nous avons associé à la valeur -1, le deuxième est V(S), associé à la valeur 1, et le dernier est Section Critique, associé à la valeur 0. Durant la lecture du fichier, nous initialisons 3 listes chaînées d’instructions nommées L, E et X. Pour chaque liste (en notant ‘\*’ L, E ou X), les premières instructions sont celles listées dans la partie %P\*, puis nous ajoutons 50 instructions de type Section Critique, et enfin viennent celles de la partie %E\*.

|  |  |
| --- | --- |
| INSTRUCTION | |
| Champs | |
| int type | -1, 0 ou 1  Correspond au type d’instruction (P, Section Critique, V) |
| SemaphoreObject Sem | Semaphore auquel est associée l’instruction, null par default. |
| int id | Numéro dans l’une des trois listes chaînées d’instructions L, E ou X initialisées à la lecture |
| Instruction suiv | Instruction qui suit l’instruction actuelle |
| Constructeurs | |
| public Instruction() | Initialise tous les champs à leur valeur nulle, sauf id qui est initialisé à -1 |
| public Instruction(SemaphoreObject s,int t) | Initialise Sem à s, type à t, id à  -1 et suiv à null |
| public Instruction(SemaphoreObject s,int t,Instruction suivant) | Initialise Sem à s, type à t  et suiv à null |

Lecture.class

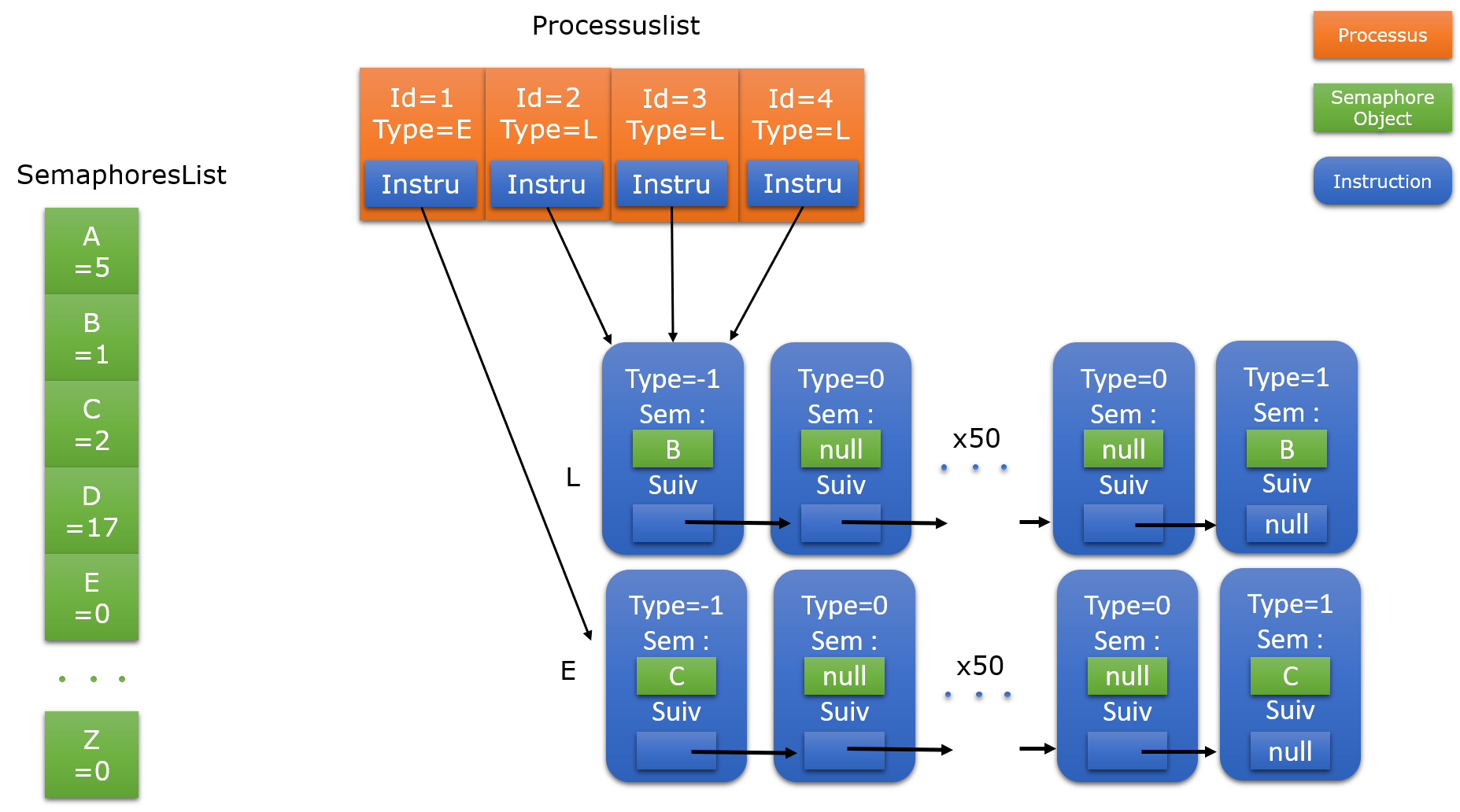
|  |  |
| --- | --- |
| LECTURE | |
| Champs | |
| boolean Valide | Indique si la lecture du fichier n’a rencontré aucun problème, true par default |
| int NbSimulation;  int nbL,nbE,nbX; | Valeurs extraites de la section %PA, 0 par default |
| Instruction L;  Instruction E;  Instruction X; | Premier maillon des listes chainées correspondant a la séquence d’instructions de chaque type de processus (Lecture, Ecriture, Exécution)  Important elles sont uniques, ie les processus pointent vers le même maillon d’instruction, cela permet d’économiser de la place en mémoire |
| SempahoreObject [] SempahoreObjectList | Un tableau des 26 sémaphores |
| Processus [] ProcessusList; | Un tableau de l’ensemble des processus (différentiable par leurs identifiants) |
| Constructeurs | |
| Lecture(String filename) | Ligne par ligne , cette fonction met dans les variables les données du fichier (« filename » ici) |

La lecture peut se résumer très grossièrement de la manière suivante :

A chaque %XX rencontré, une variable type change avec une valeur correspondante.

Ensuite quand le programme lit une ligne, il sait dans quelle « type » / « %XX » on se trouve. L’avantage de cette méthode est que les %XX peuvent intervenir dans n’importe quel ordre, l’issu de la lecture sera la même.

Voici une représentation plus visuelle des relations et organisation des données après lecture, pour un cas simple :



Memory.class

|  |  |
| --- | --- |
| MEMORY | |
| Champs | |
| int [] L  int [] E  int [] X | Ces listes d’entiers sont toujours de la même taille, chaque i-eme case correspond a la quantité de processus de type L/E/X dans le cas i, chaque cas correspond donc a un unique triplet.  Il n’y a pas de doublons. |

Cette classe permet, plus tard dans la classe simulation, de répertorier plus facilement les différents cas, de section critiques, les résultats que on cherche avec le programme.

Simulation.class

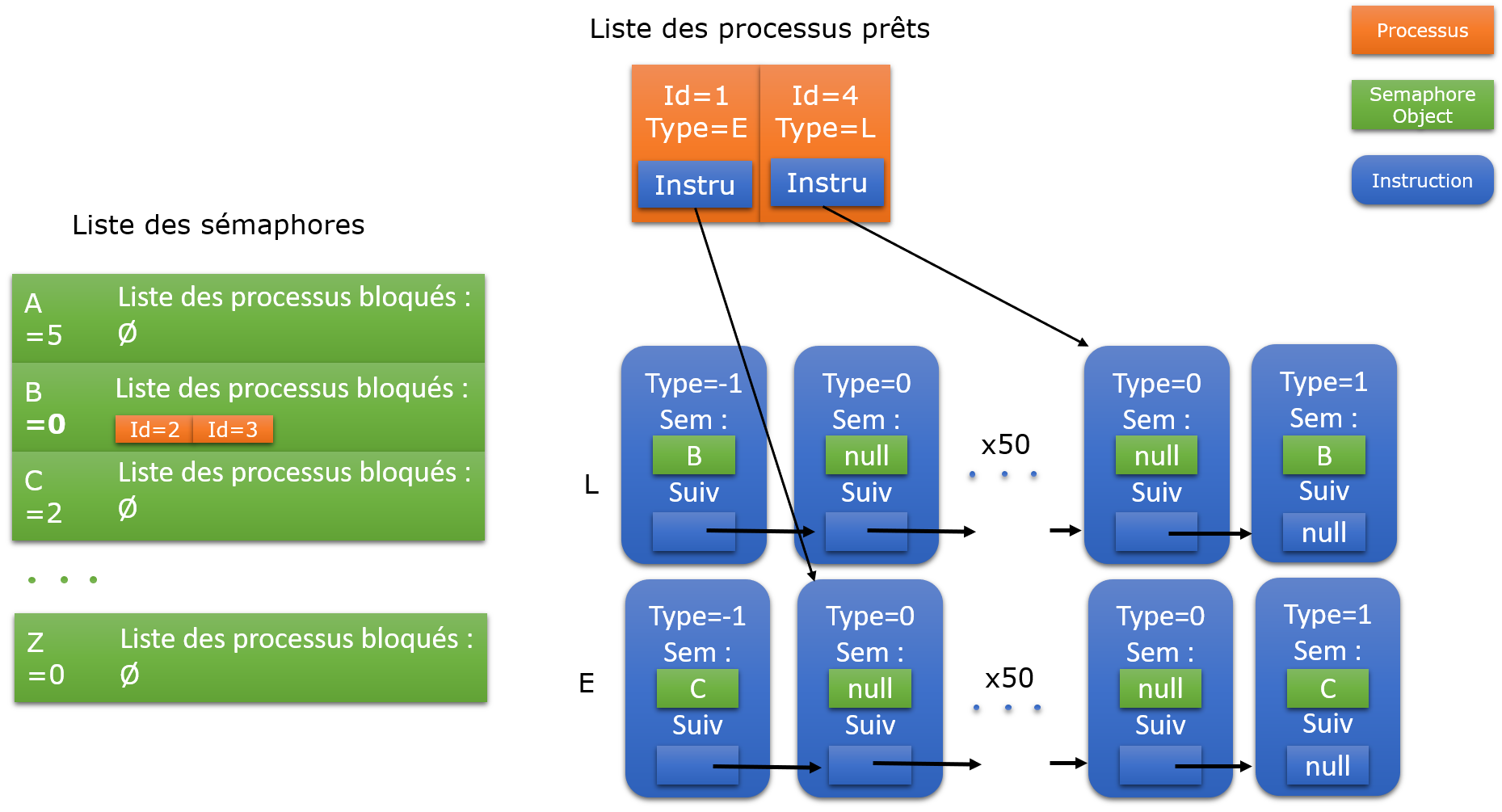
|  |  |
| --- | --- |
| SIMULATION | |
| Champs | |
| Memory memory | Dans cet objet est répertorié tous les cas possibles de processus en sections critiques. |
| long NINSTRU | Cette variable a une utilité pour le développeur, celle-ci lui permet de compte le nombre d’instruction exécuté lors d’une simulation. Si un certain nombre prédictible en fonction des paramètres n’est pas atteint, c’est qu’il y a eu un problème. |
| int readyvidecompteur | Le nombre de simulation qui échoue |
|  |  |

Simulation est un objet dans lequel N simulation ont lieu et enregistre au fur et a mesure le processus en section critique. Le constructeur de Simulation prend en argument un objet Lecture.

Explication du déroulement d’une simulation type (non exhaustif) :

* Le programme fait N fois le procédé suivant :
  + A partir de la liste des processus prêt à être exécuté, le programme **choisit aléatoirement** l’un d’entre eux, celui-ci est exécuté après avoir été retiré des processus prêts.
    - Une exécution consiste, à réaliser entre 1 et 5 instructions (aléatoirement)
      * Si la prochaine instruction a réalisé est de type P(X), on regarde le compteur du sémaphore, s’il est à zéro, le processus est replacé dans la liste des processus bloqués, cette liste se trouve dans l’objet SemaphoreObject correspondant ; l’exécution du processus s’interrompt. Sinon on décrémente le compteur du sémaphore.
      * Si la prochaine instruction a réalisé est de type V(X), on regarde la taille de la liste des processus des bloqués du sémaphore X. Si elle n’est pas à zéro, un processus choisis aléatoirement est retiré de la liste et est remis dans la liste des processus prêt. Sinon on incrémente le compteur du sémaphore.
      * Si la prochaine instruction est une instruction de type Section, rien de particulier.
      * Si la prochaine instruction est null (ie fin de liste chainée) le processus a fini son exécution.
    - Après les instructions réalisées, le processus est remis dans la liste des processus prêts, excepté s’il a été bloqué ou qu’il a fini son exécution.
    - On compte **le nombre de processus de chaque type** en section critique, et on l’ajoute a l’objet memory s’il n’est pas déjà dedans.
  + Tant que tous les processus n’ont **pas terminé leurs exécution** (ie leur prochaine instruction est null) on recommence.

Voici une représentation visuelle des relations et organisation des données pour un cas simple à un moment donné où les processus 1 et 4 s’exécutent normalement, quant à 2 et 3 sont bloqué sur le sémaphore B :



A comparer avec le schéma précédant, pour plus de clarté.

Semaphore.class

|  |
| --- |
| SEMAPHORE (main) |

Dans cette dernière classe est réunis tout le travail des classes précédentes, et c’est d’ici que les résultats sont affichés sur le terminal.

On commence par **lire** le fichier avec LECTURE.

On effectue la **simulation** avec SIMULATION

L’objet memory contenue dans l’objet SIMULATION est trié puis affiché sur le terminal.

A l’utilisateur d’interpréter les résultats à l’aide du Manuel Utilisateur (section Comment se servir de ce programme ?).

# Notes de developpeurs

**AMELIORATION POSSIBLE**

Ce programme n’est pas parfait, et peut donc toujours être améliorer. Voici quelques pistes d’améliorations que nous suggérons :

* Ajouter la fonctionnalité de mutex.
* Plus d’instructions possibles.
* La possibilité de détecté les erreurs que le programme ne déclare pas. (cf Manuel Utilisateur section Comprendre les alertes du programme)
* Une meilleure interface graphique, pour facilité la prise en main de l’utilisateur.

**LIMITES**

A cause de tous les tests que le programme réalise, celui-ci est bien plus long qu’un processeur. Pour des tests qui ont plus de 100 millions d’instructions à réaliser, le temps d’exécution du programme peut dépasser les 10 secondes.

**DIFFICULTES RENCONTREES**

Lorsque que nous avons commencé à tester notre programme, nous utilisions le test du sujet, et la simulation se passait correctement. Cependant avec d’autre fichiers que nous créions, nous avions des segmentation fault. Après de longues heures a essayé de comprendre d’où venait le problème. Nous avons réalisé que dans certaines solutions, tous les processus étaient bloqués, et que notre programme oubliait cette possibilité. C’est ainsi que notre code a évolué.