

SEG 2506

Construction de logiciels.

LABORATOIRE 8 :

Evaluation de Performance
SDL vs. Java

présenté à:

prof. G. Bochmann, Ph.D.

par:

Renaud Bougueng Tchemeube

#4333634

rboug039@uottawa.ca

Note :
13/10

École d'Ingénierie et de Technologie de l'Information.

Université d'Ottawa.

Le lundi 07 Avril 2008

Barème de correction --- en ligne avec la section anglaise.

- 1/1 Introduction
- 3/3 Modification made to the given Java program in Exercise 1,
3/3 and discussion on the design and implementation of the SDL version of the system
- 1/1 Results of the performance measurements
1/1 and discussion
- 1/1 Problems encountered and lessons learned
- /0 Conclusions

Bonus : 3pts
-saisie exceptionnelle des buts
=> contenu des discussions
-qualité hors-pair
-SDL fonctionne

I. Introduction

Il s'agissait dans ce laboratoire d'évaluer, faire des mesures et comparer la performance observée dans un système écrit en SDL versus un système en créé en Java. Chacun de ces systèmes devaient implémenter une application simulant les comportements concurrents d'un ensemble de producteurs et de consommateurs de messages.

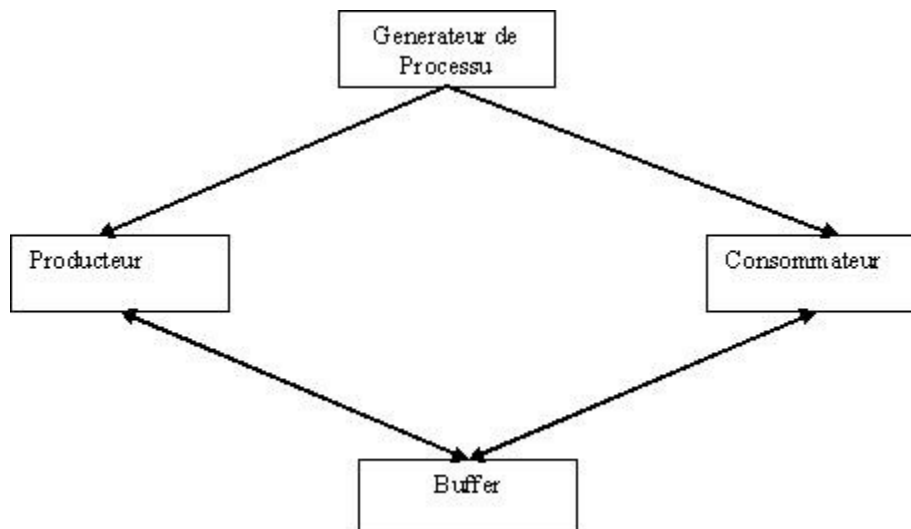
D'un côté, les producteurs produisent des messages qu'ils stockent dans un tampon commun de capacité 1; de l'autre, les consommateurs récupèrent dans ce même tampon des messages.

L'application devait alors respecter les règles de synchronisation suivantes :

1. Bien sûr, un producteur ne peut produire un message que si le tampon n'est pas plein; dans cet exemple, le tampon doit être vide
2. Un consommateur ne peut lire un message que s'il y a un message dans le tampon
3. Les producteurs attendent en file dans le cas où le tampon contient déjà un message (ou quand un autre producteur met un message dans le tampon)
4. Les consommateurs attendent en file dans le cas où le tampon est vide (ou quand un autre consommateur prend le message dans le tampon)
5. Après avoir fini un producteur/consommateur réveille un consommateur/producteur dans la file, s'il y en a.

Le code source de la simulation coté Java nous est fourni en partie.

Le schéma suivant montre la coordination entre les différents processus :



II. Modifications apportées au programme de l'exercice 1

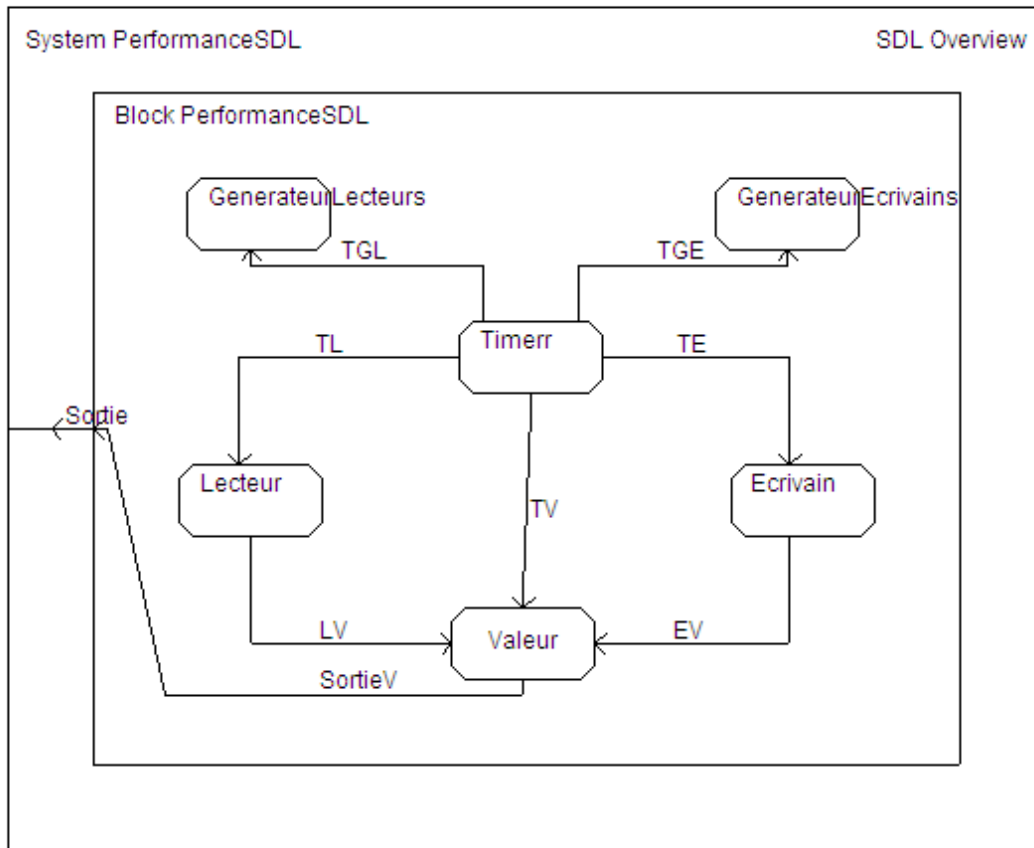
Comme nous venons de le dire, l'implémentation Java de l'application nous est fournie, mais est incomplète. Les modifications que nous lui avons apportées pour la rendre effective sont les suivantes :

- Premièrement, il fallait ajouter les instructions « `l.start()` » et « `e.start()` » respectivement dans les classes « `generateurLecteurs` » et « `generateurEcrivains` » pour qu'il fasse exécuter les processus lecteurs et écrivains qu'ils créent.
- Nous avons aussi ajouté une minuterie (le fichier « `Timer.java` ») permettant de contrôler la durée générale de la simulation. La durée par défaut a été réglée à 2s.
- Nous avons ajouté la classe `Sync.java` pour synchroniser les appels de modification (incréméntation et décrémentation) sur la variable entière « `num` » correspondant nombre de processus qui roulent. Cela nous permettait de savoir quand est-ce que tous les processus ont terminé.
- Il nous a donc fallu passer un objet `syncThreads` de type `Sync` et un objet `timer` de type `Timer` pour les contrôles à chaque fils.
- Il fallait aussi préciser pour chacune des structures génériques « `Vector` » utilisées, le type d'objet qu'elles devaient contenir. Ex : `Vector<Ecrivain>` ou `Vector<Lecteur>`.
- Il nous fallait aussi synchroniser les appels au « `system.out.println("lecteur " + nom + " lit " + val);` » et « `system.out.println("ecrivain " + nom + " écrit " + val);` » d'affichage de lectures et d'écritures pour qu'ils restent cohérents avec l'exécution réelle de la simulation. Pour cela, nous les avons tous simplement dans les méthodes déjà synchronisées « `public synchronized void ecrire(long x, String nom)` » et « `public synchronized void lire(String nom)` » de la classe « `UneValeur` ».
- Nous avons remplacé la ligne de code `this.notify()` par `this.notifyAll()` pour éviter les deadlocks.

A noter que nous avons effectué les tests avec l'environnement de développement Eclipse.

III. Conception de la version SDL du système


Au niveau de la conception SDL, nous avons adopté une approche d'implémentation simple. Dans le but de pouvoir comparer équitablement les résultats des simulations générées par les deux langages, nous avons tenu à rester dans la même optique d'orienté-objet en remplaçant alors nos classes par des processus. Cela nous donne une architecture SDL semblable à celle obtenue en Java :



1

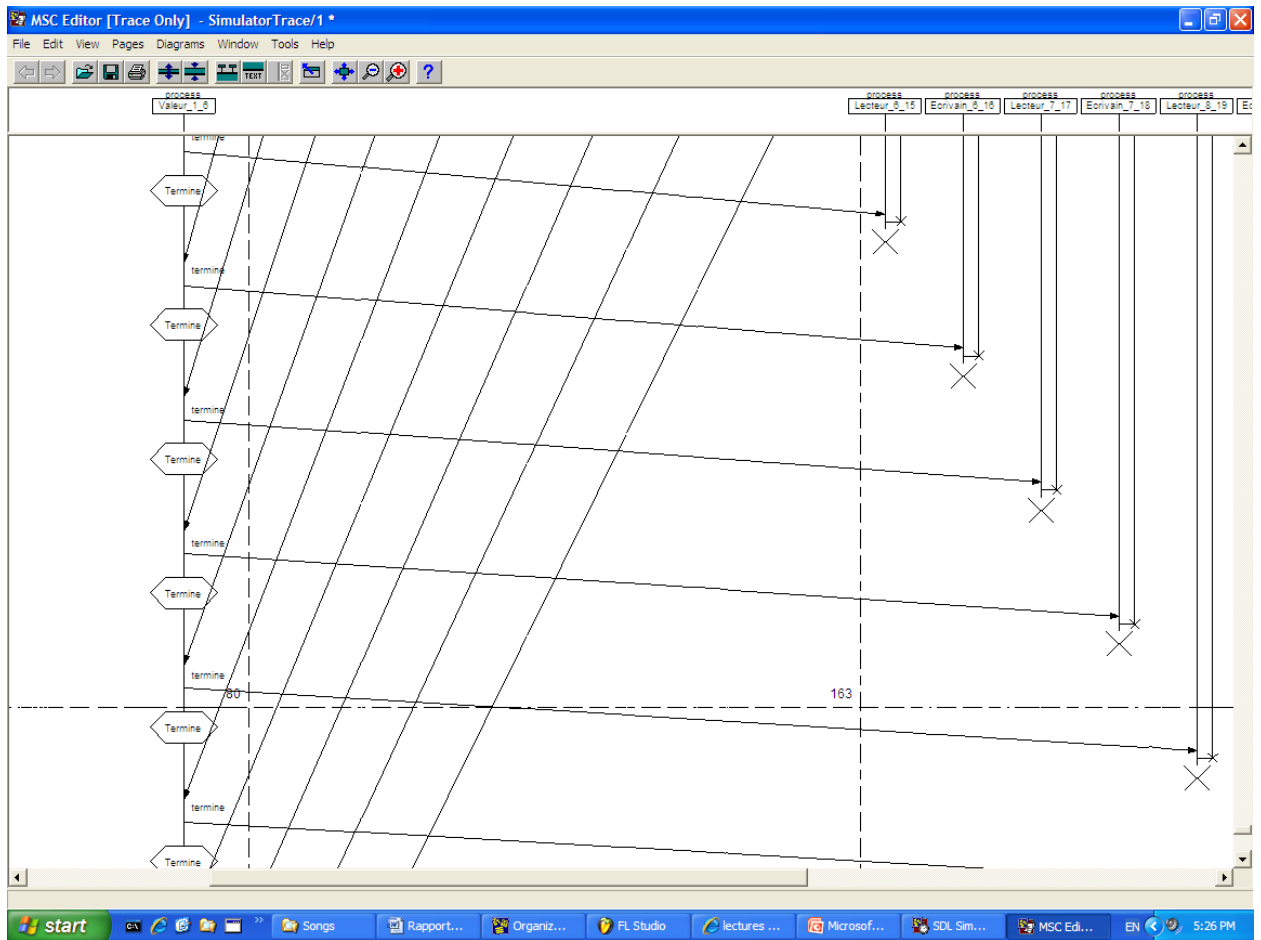
Diagramme SDL

Cette architecture considère que nous avons un générateur d'écrivains et un générateur de lecteurs qui se chargent de générer ces processus durant la simulation. Nous avons donc utilisé

le symbole  permettant de générer un processus en SDL. Les lecteurs ou écrivains une fois créés tentent indéfiniment de lire ou d'écrire respectivement dans le processus `Valeur` en transmettant les signaux « `ecrire(Integer)` » ou « `lire` ».

Le processus tampon a un attribut entier « `valeur` » et deux états : Vide et Plein.

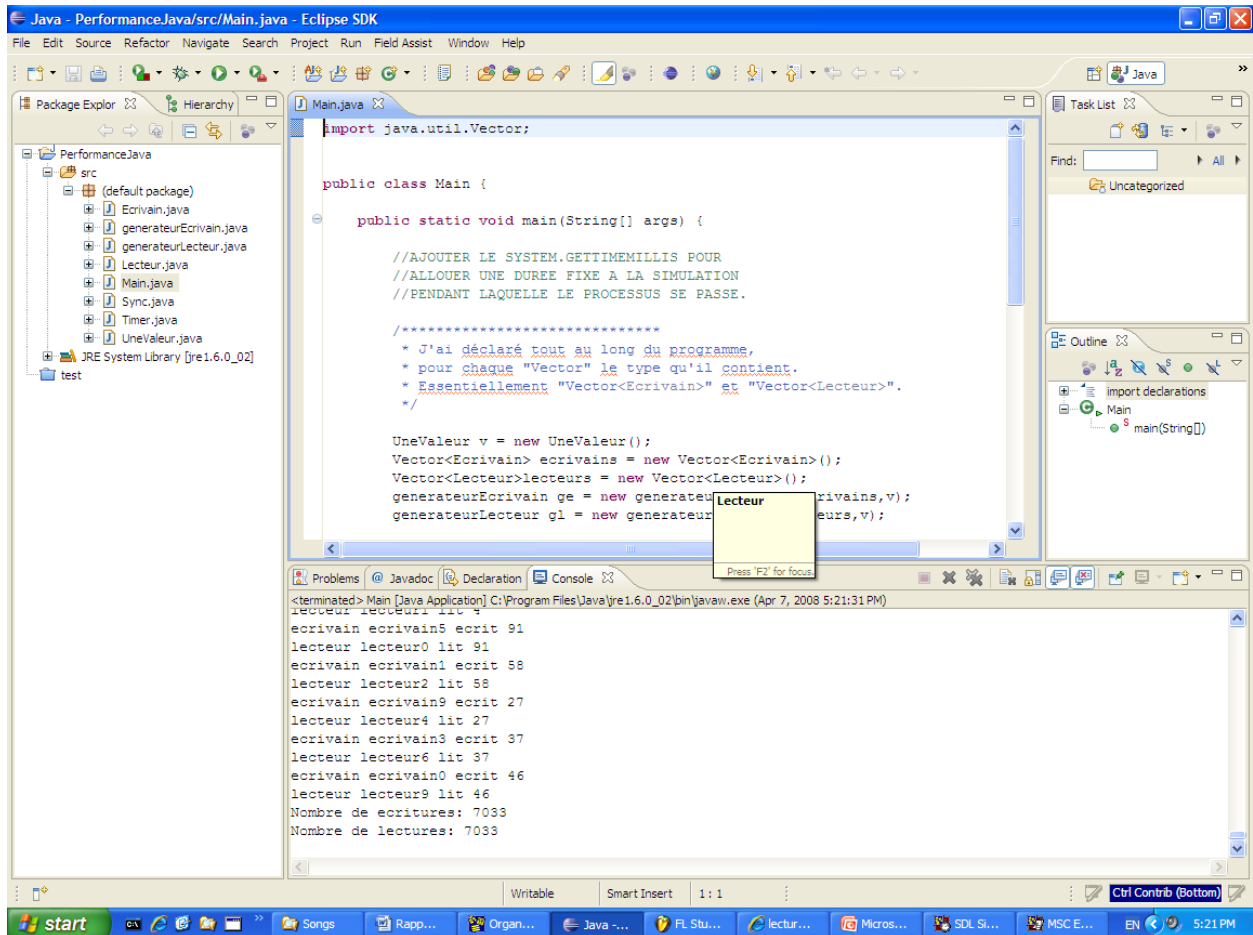
Dans l'état vide, le tampon peut alors recevoir des signaux d'écriture et mettre à jour sa valeur. Pour être vraiment fidèle à l'implémentation Java, si on reçoit des signaux de lectures dans l'état vide, on les sauvegarde dans cet état. Ils seront ensuite traités comme dans une FIFO (premier arrivé, premier servi) lorsque le tampon passera dans l'état plein. De même, dans l'état plein, le tampon consomme les signaux de lectures et sauvegarde les signaux d'écritures reçus.



Les processus sont terminés par le processus Valeur

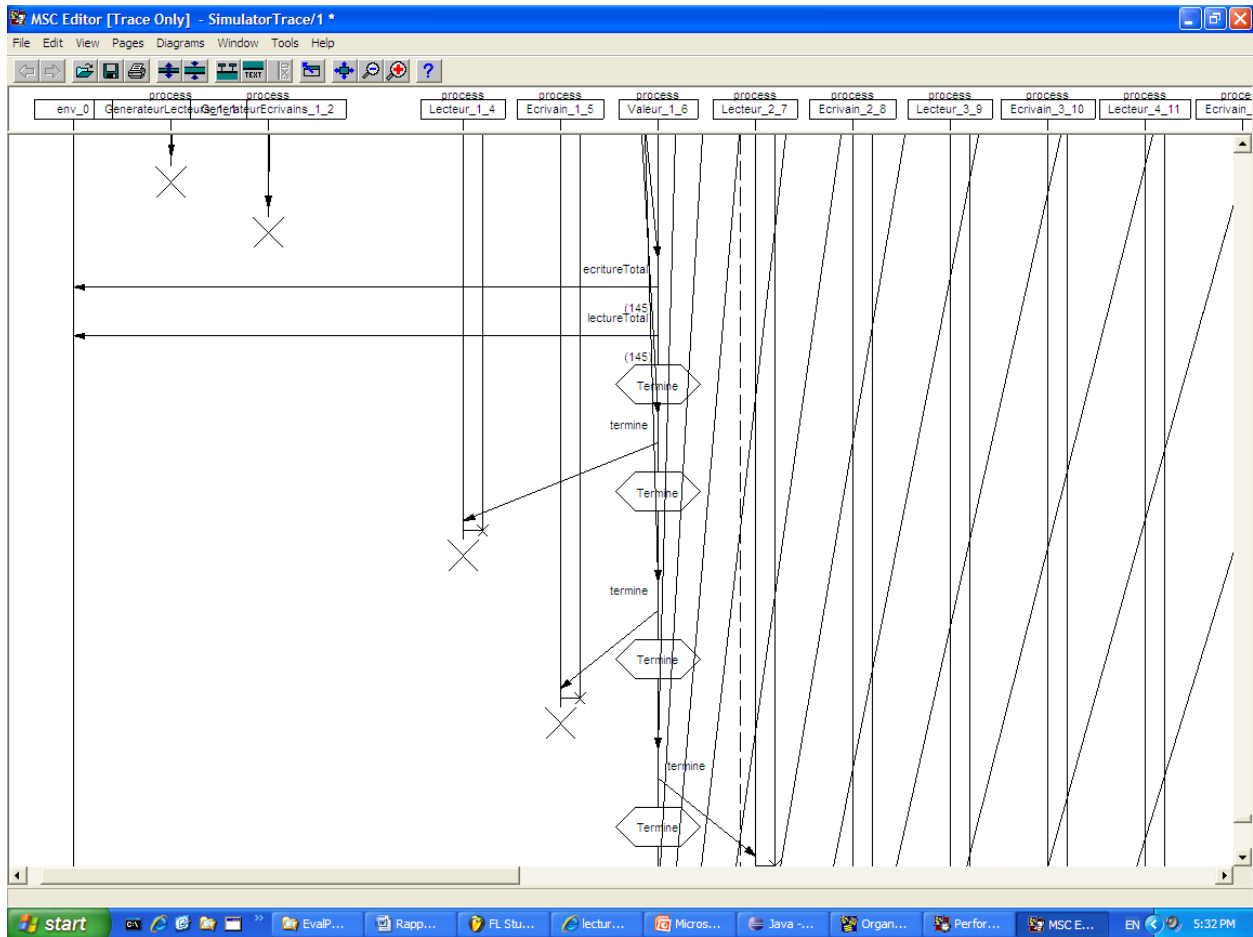
IV. Résultats des mesures expérimentales et discussion

Nous avons juste envoyé dans la soumission, le cas de simulation pour 10 Lecteurs et 10 Ecrivains. Dans Eclipse, nous avons obtenu lors de plusieurs simulations d'une durée de 2000ms des nombres de lectures et d'écritures tournant autour des 7000.



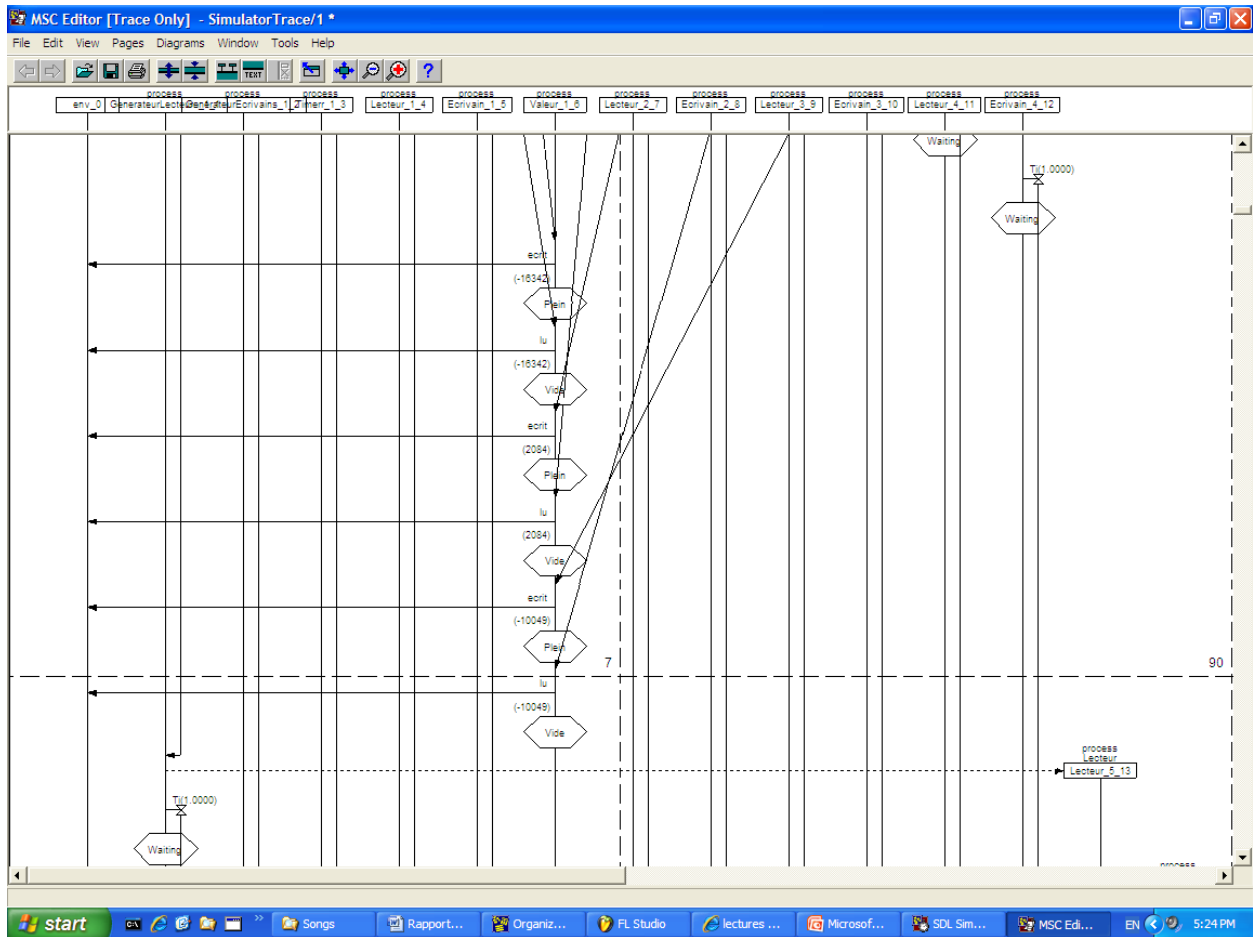
Simulation d'Eclipse qui affiche 7033 lectures et écritures sur 2000ms

En SDL, nous n'avons pas pu vraiment effectuer des simulations sur 2000ms à cause du fait qu'il est très lent vu qu'ils effectuent beaucoup d'affichages au système. Cela lui prend énormément de temps. Sinon, une simulation SDL de 20ms a donné 145 comme nombre pour les lectures et les écritures dans le tampon.



Une comparaison équitable ne peut donc pas vraiment se faire vu que les temps de simulations sont différents. A la limite nous pouvons faire une approximation linéaire sur la simulation SDL. Aussi, on aura environ 14500 écritures et lectures pour la simulation.

Évidemment, cette performance est très biaisée puisque que nous savons que, plus il y aura de processus à rouler, plus la performance diminuera, car il faut traiter plus d'opérations pendant le même temps. Dire que cela pourrait diminuer la performance de moitié ($14500/2 = 7250$)? Peut-être mais pas sur. On pourra donc donner de l'avantage en performance au SDL qui de plus, est plus spécialisé pour ce type d'application et donc le paradigme est plus adapté.



Lectures et Écritures effectuées par les processus

N.B : Eclipse ou DrJava.

Nous avons remarqué qu'en DrJava, la performance est clairement diminuée. Ceci est en grande partie du au fait que DrJava roule sur une Machine Virtuelle (JVM). Cela implique donc qu'il prend du temps pour rouler d'autres processus tel que son « garbage collector » et autres.

V. Problèmes rencontrés et leçons apprises

- Notify/NotifyAll

Dans l'implémentation Java qui nous avait été donnée, la commade « this.notify() » avait été utilisé. Cependant, il s'avère que cela pouvait causer des « deadlocks ». En effet, considérons le scénario suivant :

Tous les fils d'écritures et de lectures sont en attentes (le tampon est en wait()); sauf un fil qui est en train d'écrire au tampon. Supposons que le fil finit d'écriture, le tampon est plein, et il fait un notify(). Cette commande ne destine à réveiller qu'un seul fil. Si ce dernier est un fil d'écriture, on aura un « deadlock », car il ne pourra pas écrire dans le

tampon plein, et ne pourra notifier aucun autre fil. Le tampon restera indéfiniment en mode wait().

Alors avec le notifyAll(), tous les fils sont mis au courant et même si un écrivain ne peut pas écrire, un lecteur lira puisque tout le monde est mis au courant.

- « Deadlock » pour Java $t < 100\text{ms}$.

Avec l'implémentation Java que ce soit avec Eclipse ou avec DrJava, nous avons eu un « deadlock » lorsque notre temps de simulation était très court (dès que $t < 100\text{ms}$ environ).

Nous n'avons pas vraiment pu trouver une explication à cela, puisque nous avons bien une minuterie qui comptait le temps, et que tous nos processus demandaient justement si c'est la fin de la simulation avant d'effectuer à nouveau des instructions de boucles.

- Sauvegarder les signaux pour créer une file de processus de signaux

En SDL, l'outil de sauvegarde a été très utile puisque qu'il permettait avec un seul symbole, de faire ce que Java faisait avec plusieurs wait() et notifyAll().

- Communiquer au processus d'arrêter avec une minuterie :

Notre gros problème en SDL a été de pouvoir faire terminer les processus quand arrivait la fin de la simulation. Nous avons pensé à plusieurs conceptions :

- o Alerter tout le monde : Impossible. Le timer ne reconnaît pas les processus lecteurs et écrivains générés.
 - o Avertir le processus tampon et terminer les générateurs.
 - Mettre une valeur sentinelle dans le processus tampon qui indique aux lecteurs et écrivains qui essaient d'envoyer des signaux qu'ils doivent terminer aussi. Cependant, difficile pour les écrivains car ils tentent de mettre à jour la valeur du tampon. De plus, la valeur sentinelle n'est pas vraiment unique puisque l'écrivain génère des valeurs entières aléatoires.
 - Mettre le tampon dans un état de terminaison. Cela oblige les lecteurs et écrivains qui voient que le tampon est en état de terminaison de terminer eux aussi. Ceci est celui que nous avons choisi. Il est souple et facile à implémenter.
- Temps de simulation
 - Fichiers manquant pour les commandes « analyze » et « make » de SDL

Nous avons tout simplement copié les fichiers Visual Studio de type « sctos_smc.obj » et « sctpost_smc.obj » d'autres documents SDL qui fonctionnaient déjà et les avaient générés.

- Valeur random `val := ANY(Integer).`

On ne savait pas comment générer des valeurs aléatoires. Mais cela nous a donné par le TA.

VI. Conclusion

En somme, ce laboratoire nous aura permis de comprendre les enjeux de la programmation concurrente dans l'exemple simple d'accès simultanés en « multithreading ». Nous avons été aptes à implémenter ce genre de simulation avec un langage orienté comme Java et avec le langage de diagramme SDL. Cela nous a permis alors de débattre des questions de performances en multiprogrammation soulevés par les deux paradigmes. Nous avons donc beaucoup appris, en plus d'approfondir nos connaissances dans les deux langages de programmation.