IFT2245 – Travail pratique #2 – 2009-10-26

PROCESSUS LÉGERS ET SYNCHRONISATION

Marc Feeley

Le TP2 a pour but d'étudier un système d'exploitation minimal possédant des processus légers ("MINOS2"), de compléter l'implantation des mécanismes de synchronisation et de compléter l'implantation d'un jeu video de type "Pong" et le jeu de la vie qui utilisent des processus légers. Une partie importante du TP2 est de lire les sources de MINOS2 pour comprendre son fonctionnement et le lien entre les différentes composantes et en particulier l'ordonnanceur de processus.

---

1 Survol de MINOS2

Comme pour le TP1, les sources incomplets de MINOS2 vous sont fournis et il faut les modifier pour implanter les fonctionalités précisées dans ce document. Pour obtenir une copie des sources que vous pouvez modifier, entrez la commande:

```
% tar zxf ~dift2245/tp2/minos2.tar.gz
```

Cela va créer le sous-répertoire `minos2`. Vous pouvez dès maintenant compiler MINOS2 en entrant la commande "`make`" dans le sous-répertoire `minos2`. Cela crée le fichier "`floppy`" qui est une image de floppy 1.44 MB qui contient le SE. Après toute modification des sources, assurez-vous de faire "`make`" pour créer le fichier "`floppy`" à nouveau.

Chaque endroit qui doit être (possiblement) modifié dans les sources est indiqué par un commentaire contenant quatre étoiles. Modifiez uniquement les fichiers "`makefile`", "`main.cpp`", "`fifo.cpp`", "`include/fifo.h`" et "`thread.cpp`". Le fichier "`thread.cpp`" contient l'implantation incomplète des classes "`condvar`" (variables de condition), "`barrier`" (barrière de synchronisation), "`thread`" (processus légers) et "`scheduler`" (l'ordonnanceur de processus). Le fichier "`main.cpp`" contient l'implantation incomplète de la classe "`pong`" (jeu vidéo Pong), la classe "`life`" (jeu de la vie) et la classe "`cpu_load`" (calcul de la charge CPU). Le fichier "`fifo.cpp`" contient l'implantation incomplète de la classe "`fifo`" (une file d'attente d'octets de longueur bornée).

2 Tâches à accomplir

2.1 Classe "`fifo`"

La classe "`fifo`" implante une file d'attente d'octets de longueur bornée. Les méthodes suivantes sont implantées:

```
void put (uint8 byte);
void get (uint8* byte);
bool get_or_timeout (uint8* byte, time timeout);
```

Un `fifo` permet à deux processus d'échanger des informations : un processus écrit des octets sur le `fifo` avec la méthode "`put`" et l'autre processus lit des octets du `fifo` avec la méthode "`get`" ou

"get_or_timeout". Les méthodes "put" et "get" bloquent le processus appelant jusqu'à ce que l'opération complète. La méthode "get_or_timeout" retourne FALSE si le processus attends plus que le temps absolu indiqué par "timeout". Le type "time" correspond à un temps absolu. Le temps présent est retourné par l'appel current_time(), et le temps x nanosecondes dans le futur est retourné par l'appel add_time(current_time(),nanoseconds_to_time(x)).

Vous devez compléter l'implantation de la classe fifo en modifiant les fichiers "include/fifo.h" et "fifo.cpp". Cette classe est utile pour synchroniser deux ou plusieurs processus qui s'échangent des messages (par exemple pour implanter la classe pong).

2.2 Variables de condition

Modifiez la classe "condvar" pour implanter les variables de condition. Les méthodes suivantes doivent être implantées:

```
void wait (mutex* m);
bool wait_or_timeout (mutex* m, time timeout);
void signal ();
void broadcast ();
```

La méthode wait suspend le processus présent sur la variable de condition et cède le mutex "m". Au réveil du processus, l'acquisition du mutex "m" se fait automatiquement avant de retourner de wait.

La méthode wait_or_timeout est similaire à wait mais si le processus attends plus que le temps absolu indiqué par "timeout" le processus sera réveillé, FALSE est retourné et le mutex "m" n'est pas acquis.

La méthode signal réveille le processus qui est en attente sur la variable de condition depuis le plus longtemps (c'est-à-dire, premier arrivé premier réveillé).

La méthode broadcast réveille tous les processus qui sont en attente sur la variable de condition.

Les méthodes signal et broadcast sont déjà implantées correctement. Il reste à compléter l'implantation de wait et wait_or_timeout. Inspirez-vous de l'implantation des méthodes lock et lock_or_timeout de la classe "mutex". Ceci est la seule partie du TP2 où vous pouvez utiliser les macros "disable_interrupts" et "enable_interrupts".

Il est important de comprendre comment fonctionne l'ordonnanceur de processus. Celui-ci maintient deux files d'attente de processus: le readyq (processus exécutables) et le sleepq (processus qui sont bloqués sur une synchronisation avec timeout). À chaque interruption du timer, l'ordonnanceur réveille les processus sur le sleepq qui ont dépassé leur timeout. Un processus peut être à la fois sur une file d'attente (d'un mutex ou variable de condition) et sur le sleepq (ce cas arrive en particulier lorsqu'un processus est bloqué dans un appel à wait_or_timeout). Lorsque l'ordonnanceur réveille un processus il est retiré des deux files avant de le mettre sur le readyq.

2.3 Classe "barrier"

Modifiez la classe "barrier" pour implanter les barrières de synchronisation. Les méthodes suivantes doivent être implantées:

```
barrier (int n);
void wait ();
```

Le constructeur `barrier` crée une barrière qui a la capacité de synchroniser `n` processus légérs.

Chaque appel à la méthode `wait` suspend le processus présent sur la barrière, sauf le n^{eme} appel (et le $2n^{eme}$, le $3n^{eme}$, etc) qui libère tous les processus suspendus sur la barrière.

Votre implantation doit être basée sur les variables de condition.

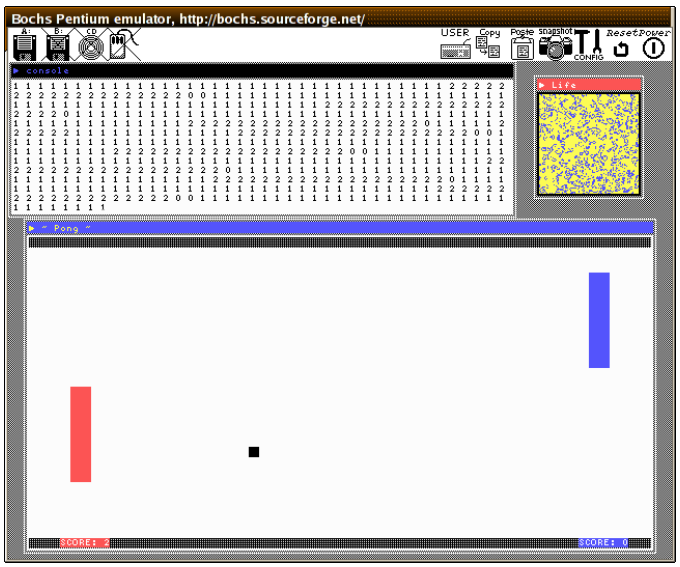
2.4 Classe "`life`"

Afin de tester vos méthodes, le fichier "`main.cpp`" contient une implantation du jeu de la vie qui crée 17 processus légers (16 pour l'animation de la grille de jeu et un pour l'affichage) et les synchronise avec une barrière de synchronisation pour que le jeu évolue à exactement 10 cycles par seconde. Pour que l'animation fonctionne correctement il faut que l'implantation des variables de condition et barrière soient correctes (sinon l'évolution des zones sera désynchronisée).

Un autre problème à régler concerne l'affichage. Les accès à l'écran par les divers processus ne sont pas synchronisés, ce qui cause des défauts d'affichage ("visual artifacts"). Souvenez-vous que le mode graphique VGA utilisé demande au programme de sélectionner à tour de rôle les 4 couches de la mémoire graphique pour y faire les écritures nécessaires. Des accès concurrents par de multiples processus doivent donc être synchronisés pour éviter que les écritures se fassent dans la mauvaise couche. Rajoutez ce qu'il faut dans "`main.cpp`" pour qu'il y ait une exclusion mutuelle entre les divers affichages à l'écran.

2.5 Classe "`pong`"

La classe "`pong`" dans le fichier "`main.cpp`" contient une implantation très incomplète d'un jeu vidéo "Pong". Pong est un des plus vieux jeux vidéo. Il simule une partie de tennis de table entre deux joueurs. La balle, un petit carré, rebondit sur les murs (en haut et en bas de la zone de jeu) ainsi que sur les palettes contrôlées par les deux joueurs (situées du coté gauche et droit de la zone de jeu). À chaque rebondissement est émit un court "bip" dont la tonalité dépend de la surface touchée par la balle (palette ou mur). La balle est mise en jeu au centre de l'écran à une position et une direction aléatoire. À chaque fois que la balle dépasse la palette d'un joueur, un son grave est émit et l'autre joueur marque un point. À 11 points la partie est terminée. Voici ce que notre programme complet affiche sous Bochs:



Le bas de l'écran contient le jeu de Pong. En haut à droite une fenêtre montre l'évolution du jeu de la vie (chaque pixel de la grille 64×64 représente une case pouvant contenir une vie). Finalement le

système d'exploitation démarre un processus qui affiche à chaque seconde sur la console en haut de l'écran un entier qui indique la charge du système. Plus le nombre est faible, plus le système est chargé. Le nombre est proportionnel à la portion de cycles CPU qui est restée inutilisée par les autres processus. Une implantation efficace des mécanismes de synchronisation et du jeu de Pong devrait atteindre un nombre supérieur à 50000 sur le vrai PC. Toute attente active va diminuer ce nombre considérablement.

Votre implantation du jeu de Pong doit respecter les contraintes suivantes. Le jeu débute lorsque la touche "enter" est enfoncée. La palette du joueur de gauche est contrôlée par la souris (la palette se déplace à une vitesse constante jusqu'à la coordonnée "y" du curseur) et le joueur de droite contrôle sa palette avec les flèches haut/bas sur le clavier (la palette se déplace à la même vitesse constante dans la direction indiquée par la touche tant que la touche est enfoncée). Cela peut se faire avec un appel à la fonction "`keypressed(`*keycode*`)`" qui lit le contenu du "keymap" mémorisant les touches du clavier présentement enfoncées. Vous devez aussi utiliser au moins 3 processus légers pour contrôler les objets mobiles du jeu (palettes et balle) et les synchroniser correctement. Vous pouvez utiliser des processus additionnels (par exemple pour générer le son).

À part ces contraintes, vous devez faire des choix de design raisonnables (taille des palettes, vitesse de la balle, couleurs, sons, etc). Nous accorderons jusqu'à 10% de boni pour le "game play" et les designs créatifs (qui respectent quand même les contraintes données ci-dessus). En particulier, nous vous signalons qu'une simulation de Pong fidèle à la première version en arcade est disponible sur l'internet à cette adresse: `http://robotubegames.com/play_game.php?gameid=26` et que ceux qui réaliseront une implantation fidèle à cette version obtiendront le 10% de boni.

Pour produire les tonalités sur le haut-parleur du PC, vous devez utiliser le circuit 8253 du PC (le PIT ou "Programmable Interval Timer"). Les détails d'utilisation de ce circuit sont disponibles sur l'internet à cette adresse: `http://www.dcc.unicamp.br/~celio/mc404s2-03/8253timer.html`. Soyez conscient que l'ordonnanceur utilise un des trois timers du PIT pour faire le multiplexage du CPU (le CPU est interrompu par le PIT à chaque 0.0001 seconde). Il ne faut pas perturber son fonctionnement. Sachez aussi que Bochs ne simule pas le haut-parleur du PC (il faut donc utiliser un vrai PC pour tester cette partie).

3 Évaluation

Il n'y a pas de rapport à rédiger pour le TP2. Il suffit de faire la remise des fichiers à modifier ainsi que le fichier "`floppy`". Il faut aussi faire un test en présence du démonstrateur sur un vrai PC, pour montrer que votre système fonctionne correctement. La procédure à suivre vous sera expliquée plus tard. Le contenu de votre disquette au moment du test doit être identique au contenu du fichier "`floppy`" et celui qui est produit avec un "make" à partir de vos sources (sinon vous obtiendrez zéro pour le test).

- Pondération:
  - 20%: Implantation des variables de condition
  - 10%: Implantation de la barrière de synchronisation
  - 10%: Élimination des défauts d'affichage
  - 10%: Implantation de la classe `fifo`
  - 30%: Implantation de la classe `pong`
  - 20%: Implantation du son
  - 10%: boni pour le "game play"
- L'élégance et la lisibilité du code, l'exactitude et la performance sont des critères d'évaluation.
- **Vous devez faire le travail par groupes de 2 personnes. Vous devez confirmer la composition de votre équipe (noms des coéquipiers) au démonstrateur au plus tard à la démonstration du 2 novembre.** Indiquez vos noms dans le fichier "`makefile`" tel que demandé.

- Vous avez jusqu'à 23h59 le 15 novembre pour remettre vos fichiers. Vous serez pénalisés de 30% par jour de retard.
- Vous devez remettre vos programmes par remise électronique grâce à la commande suivante:

```
remise ift2245 tp2 floppy makefile main.cpp fifo.cpp include/fifo.h thread.cpp
```