

Annexe M

Travaux Pratiques 7

L'objectif de ce TP est de mettre en pratique les concepts présentés dans le chapitre sur la gestion fine de la mémoire.

Nous vous demandons de télécharger depuis le serveur SVN (voir chapitre 5) le projet TP7_Eleves et de nommer l'importation TP7 (dernier écran d'importation).

Le déroulement du TP est subdivisé en deux parties. La première consiste à valider les acquis du TP5. On vous demande de développer une application permettant de calculer la valeur binaire d'un nombre via la synchronisation de plusieurs activités.

La seconde partie se consacre à la manipulation puis l'implémentation de mécanismes de gestion de mémoire.

Attention : Il vous est demandé de configurer qemu pour initialiser la taille mémoire disponible sur le PC à 4 Mo. Pour cela, rajouter ou début des options de démarrage de qemu `-m 4`. Vous trouverez ces options le champs Arguments de la fenêtre disponible dans Run->External Tools->External Tools Configurations.

Question 1 - Durée estimée 30 minutes

On vous demande de compléter le programme Pipeline, situé dans le répertoire `Applications/Pipeline`, qui convertit des nombres entier de valeur inférieure à 256 de la base 10 vers la base 2. Le nombre à convertir est stocké dans la variable globale `globalCounter`. Le calcul d'une tâche (ou activité/thread) se résume à afficher un 0 ou 1 selon que le nombre contenu dans la variable `globalCounter` est divisible ou non par deux. Pour afficher la valeur binaire d'un nombre, nous devons lancer autant de tâches que de bit composant le nombre. Nous prenons pour hypothèse que le nombre à convertir sera inférieur à 256, il nous faut donc 8 tâches pour afficher la valeur binaire du nombre (en effet $2^8=256$). Les tâches sont identifiées par un identifiant unique (un numéro entre 0 et 7). Une fois que la tâche n a affiché 0 ou 1 elle divise par 2 `globalCounter`. (l'opération `globalCounter=globalCounter >> 1`; permet de diviser par 2 `globalCounter` par décalage de bit).

Pour la première partie de la question, nous n'imposons pas d'ordre strict. -Chacune des activités peut démarrer le calcul dès qu'elle le peut. (Indication : un seul sémaphore suffi). En d'autres termes, la tâche n peut s'exécuter avant la tâche $n-1$. En revanche, une seule tâche doit être active au même moment.

Dans la seconde partie, nous vous demandons d'adapter votre code pour respecter un ordre d'exécution prédéterminer des tâches. Dans cette partie, la tâche n ne devra s'exécuter qu'après la tâche $n-1$ (Indication : 8 sémaphores).

Question 2 - Durée estimée 15 minutes

Etudiez le code du fichier `Memoire.cpp`. Cette architecture permet de remplacer assez simplement un gestionnaire mémoire par un autre en redéfinissant le type de la variable `uneMemoire`. Ce code respect le patron de conception `Singleton`. Détaillez ce point à votre encadreur de TP.

Nous vous rappelons qu'en génie logiciel, le singleton est un patron de conception) dont l'objet est de restreindre l'instanciation d'une classe à un seul objet. Il est utilisé lorsque l'on a besoin d'exactly un objet pour coordonner des opérations dans un système tel qu'ici le gestionnaire de mémoire (qui doit être unique dans le système sextant). On implémente le singleton en écrivant une classe contenant une méthode qui crée une instance uniquement s'il n'en existe pas encore. Sinon elle renvoie une référence vers l'objet qui existe déjà.

Question 3 - Durée estimée 30 minutes

Nous vous demandons de réimplémenter le code gestionnaire de la mémoire pion dans cette architecture (modèle Singleton). Vous pouvez bien évidemment vous aider des codes déjà implémentés dans le TP3.

Pour rappel : Un pointeur (`void *debut;`) pointe sur l'adresse du début du tas disponible (initialisé par la méthode `mem_setup`). A chaque demande d'allocation de n octets, vous devez retourner l'adresse de ce pointeur, puis décaler ce pointeur de n octets.

Pour la libération mémoire, nous ne vous demandons pas d'implémenter une gestion chaînée des blocs libres, mais simplement, dans un premier temps, de développer une méthode `freemem(int adresse, int taille)` inscrivant des zéros dans les zones mémoires libérées. Aidez vous de la fonction `memset` présente dans le fichier `op_memoire.cpp`

Le principal défaut de la méthode `freemem` est qu'elle ne peut être utilisée par l'opérateur `delete`. En effet ce dernier ne dispose pas de la taille mémoire à désallouer. La prochaine question permet de passer outre cette limitation.

Question 4 - Durée estimée 60 minutes

Nous avons décrit en cours un algorithme permettant d'allouer de la mémoire sur le tas, puis de la libérer en ne disposant que de l'adresse. L'idée consiste, dans la fonction `malloc()` à utiliser les premiers octets de la zone pour sauvegarder la taille de la zone allouée. Puis, dans la fonction `free()` de récupérer la taille en allant lire dans cette zone.

Implémentez le code du `malloc` et du `free` dans le fichier `memoirepion.cpp`.

Question 5 - Durée estimée 60 minutes

Le code détaillé dans le fichier `memoirepion.cpp` permet d'écraser par des zéros les zones allouées, mais ne permet pas de les réutiliser. Une amélioration consiste à remettre dans la partie zone allouable les zones libérées. C'est ce que propose de faire le code du fichier `memoireliste.cpp`.

Le code présenté permet de gérer dynamiquement une liste de zones mémoires libres. Détaillez sur papier l'algorithmique de ce code.

Dans la pratique, cela ne fonctionne pas car il manque au code la partie associée à la gestion de la taille de la zone à désallouer. Corrigez en vous aidant de la question 4.

Question 6 - Durée estimée 60 minutes

Comme vous pouvez vous en douter, le dernier gestionnaire de mémoire induit une fragmentation importante de la mémoire. Expliquez pourquoi.

Les gestionnaires de mémoire utilisent généralement des mécanismes d'adressage virtuel pour gérer cette fragmentation. L'adressage virtuel repose sur un découpage uniforme de la mémoire par bloc de taille fixe. Avant d'étudier les mécanismes d'adressage virtuel, nous devons disposer d'un gestionnaire de mémoire de bloc de taille fixe.

Le principe reste très simple. La mémoire est découpée en bloc de taille fixe K . Si la taille d'une demande mémoire est inférieure à K , le gestionnaire retourne l'adresse du premier bloc libre, et marque ce bloque occupé. Si la taille mémoire est supérieure à K , le gestionnaire retourne une erreur. Lors de la libération mémoire, le bloc est réinitialisé par l'écriture de zéro, et marqué comme libre. Initialement, tous les blocs sont marqués libre.

Le mécanisme permettant de gérer l'occupation des blocs (libre ou occupé), peut être implémenté de différentes manières. Nous vous laissons choisir cette implémentation, toute fois, nous vous conseillons de valider votre algorithme auprès de votre tuteur avant tout implémentation.