TP Thread Posix : Lecteur vidéo ogg/theora/vorbis

SEPC : Ensimag 2A

Résumé

L'objectif du TP est la programmation d'un lecteur de vidéo en utilisant les processus légers (threads) de la bibliothèque de threads POSIX. Nous avons étudié en travaux dirigés quelques problèmes standards de synchronisation. Vous devez implanter une solution combinant plusieurs de ces problèmes et respectant quelques contraintes.

Les synchronisations seront réalisées avec des moniteurs.

1 Décodage d'une vidéo ogg/theora/vorbis

Dans un fichier vidéo ogg/theora/vorbis, ogg est le format de stockage des données brutes, theora est le format de codage vidéo et vorbis le format de codage audio.

Le fichier ogg contient des `pages`. Ces pages contiennent des `paquets`. Chaque paquet est associé à un flux (*stream*). Un des flux code l'audio. Un autre flux code la vidéo.

Pour décoder une vidéo, les opérations suivantes sont donc réalisées :

1. lire un bout du fichier et injecter les données dans le décodeur ogg jusqu'à pouvoir en extraire une page complète
2. injecter la page complète dans le décodeur ogg
3. extraire du décodeur ogg un paquet complet
4. injecter le paquet dans le décodeur du flux (vorbis ou theora)
5. extraire du décodeur les données : échantillons (*samples*) pour l'audio, une image pour la vidéo

Chaque flux commence par des entêtes associés qui sont utilisés pour la détection et la description du flux. Mais vous n'aurez pas à gérer les aspects algorithmiques du décodage.

2 Le sujet

Pour vous aider, un squelette de code est fourni. Il fait quelques impasses, par exemple lorsque les performances du processeur sont insuffisantes, mais il devrait être fonctionnel pour une vidéo « classique ».

Vous devez implanter toute la partie concernant la gestion des threads et leurs synchronisations. Il faut, bien sûr, ajouter des structures de données et les initialiser correctement. Cela inclut les variables mutex, et les variables de conditions, ainsi que tout ce qui vous semblera nécessaire. Il faudra créer les threads en leur faisant exécuter les fonctions adéquates. Il faudra aussi gérer correctement la terminaison.

2.1 Compilation

Vous devez préalablement modifier le début du fichier `CMakeLists.txt` pour y insérer vos logins.

La création du Makefile s'effectue en utilisant `cmake` dans un répertoire où seront aussi créés les fichiers générés. Le répertoire "build" du squelette sert à cet usage. Tout ce qui apparaît dans "build" pourra donc être facilement effacé ou ignoré.

La première compilation s'effectue donc en tapant :

```
cd ensimag-video
cd build
cmake ..
make
make test
make check
```

et les suivantes, dans le répertoire `build`, avec

```
make
make test
make check
```

2.1.1 Coccinelle

Le test utilise le logiciel *coccinelle* et cherche des lignes de code C + POSIX Threads qui ressemblent à des bugs : une variable `pthread_mutex_t` différente entre le lock et le unlock dans une même fonction par exemple.

Les lignes s'affichant avec un `-` au début sont les lignes detectées dans le pattern.

Exemple avec le programme faux suivant :

```c
#include <pthread.h>

pthread_mutex_t m,m2;
pthread_cond_t c;

void A() {
    pthread_mutex_lock(&m);
    while(0)
        pthread_cond_wait(&c , &m);
    pthread_mutex_unlock(&m2);
}
```

Le test affichera :

```
diff =
--- toto.c
+++ /tmp/cocci-output-21774-1f97be-toto.c
@@ -4,8 +4,5 @@ pthread_mutex_t m,m2;
 pthread_cond_t c;

 void A() {
-    pthread_mutex_lock(&m);
     while(0)
-        pthread_cond_wait(&c , &m);
-    pthread_mutex_unlock(&m2);
 }
```

Le test fourni ne vérifie pas l'exécution du lecteur. Il faudra prendre votre propre video pour cela.

2.2 Rendu des sources

L'archive des sources que vous devez rendre dans `Teide` est généré par le makefile créé par cmake :

```
cd ensimag-video
cd build
make package_source
```

Il produit dans le répertoire `build`, un fichier ayant pour nom (à vos login près)
`Ensithreadsvideo-1.0.login1;login2;login3-Source.tar.gz`.

C'est ce fichier tar qu'il faut rendre.

2.3 Architecture du lecteur

Pour simplifier votre codage, le fichier est lu deux fois simultanément par deux threads. L'un va lire, décoder et jouer le flux audio (vorbis). L'autre va lire et décoder le flux vidéo (theora), puis il va copier chaque image à un autre thread qui est responsable de l'affichage (construction des textures) au bon moment de chaque image.

Afin de ne pas utiliser trop de mémoire à la fois, ces threads dorment (`SDL_Delay(...)`) en attendant la prochaine action à faire. Pour cela ils utilisent une référence initiale commune de temps et l'horloge temps-réel du système.

La partie audio est presque complètement gérée par la bibliothèque SDL2. Cette bibliothèque utilise elle-même des threads pour communiquer avec la carte audio.

3 Le code à réaliser

Vous devez ajouter les différents codes de gestion des threads et de synchronisation. La grande majorité de votre code de synchronisation par moniteurs sera ajouté dans le fichier `synchro.c` et `synchro.h`, mais pas uniquement.

N'oubliez pas de déclarer vos variables dans un `.c` et mettre les `extern` exposant les variables que vous souhaitez partager entre différents fichiers dans les fichiers `.h` correspondant.

3.1 Lancement et terminaison des threads

Dans la fonction `int main(int argc, char *argv[])` du fichier `ensivideo.c`, ajouter le lancement de deux threads qui exécutent, chacun, une des fonctions `theoraStreamReader` et `vorbisStreamReader`. Chacun reçoit en argument le nom du fichier à lire (`argv[1]`).

Vous devez aussi lancer le thread gérant l'affichage en exécutant la fonction `draw2SDL`. Le lancement a lieu vers la ligne 144 du fichier `stream_common.c` dans la fonction `decodeAllHeaders`. Il prend en argument le numéro du flux vidéo (`s->serial`).

Il faudra ensuite, dans la fonction `main`, attendre la terminaison du thread vorbis.

Le décodeur audio garde 1 seconde de marge. Après cette seconde de marge (le `sleep(1)` dans le code du main), vous tuerez les deux threads videos
(`pthread_cancel`) avant d'attendre leur terminaison.

3.2 Protection de la hashmap stockant les données de chaque flux

Les pointeurs vers les états des différents décodeurs sont stockés dans deux structures de type `struct streamstate`. La structure, coté vidéo, est maniée par deux threads. Il faudra donc la protéger des accès concurrents. Par simplification, la même fonction servant pour le décodeur vorbis, vous pouvez protéger les deux accès.

Le code est à ajouter à la fin de la fonction `getStreamState()` du fichier `stream_common.c` et au milieu de la fonction `draw2SDL()` du fichier `ensitheora.c`.

3.3 Attente et Producteur-consommateur

Il y a deux groupes de synchronisations qui correspondent à deux moments.

3.3.1 Affichage de la fenêtre vidéo

C'est le thread décodant le flux qui connaît la taille de l'image à afficher, il doit donc transmettre cette taille (en affectant les variables globales `windowsx` et `windowsy`) et attendre la création de la fenêtre avant de poursuivre.

Vous coderez les synchronisations des fonctions suivantes :

coté décodeur : `void envoiTailleFenetre(th_ycbcr_buffer buffer)` et `void attendreFenetreTexture()` ;

coté afficheur : `void attendreTailleFenetre()` et `void signalerFenetreEtTexturePrete()`.

Afin d'accéder aux informations de taille de la fenêtre à partir d'une variable `buffer` de type `th_ycbcr_buffer`, vous utiliserez `buffer[0].width` ainsi que `buffer[0].height`.

3.3.2 Producteur-consommateur de textures

Le fait d'avoir un seul consommateur et un seul producteur rend le code plus simple qu'un producteur-consommateur complet. Vous devez uniquement maintenir le nombre de textures déposées et pas encore affichées. La gestion du tampon de textures et des indices `tex_iwri, tex_iaff` est déjà codée par le squelette.

Le nombre maximal de textures stockable est `NBTEX`.

Vous coderez les synchronisations des fonctions suivantes :

— `void debutConsommerTexture()`,

— `void` finConsommerTexture(),
— `void` debutDeposerTexture(),
— `void` finDeposerTexture().