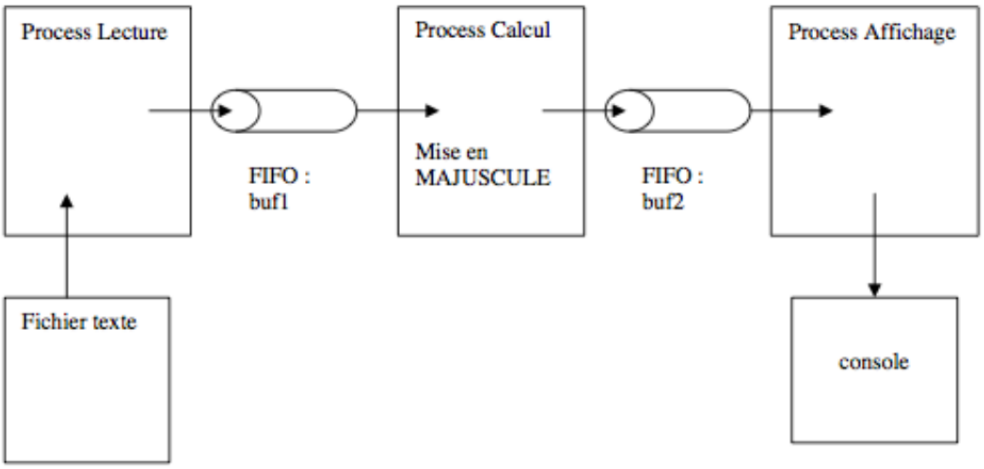
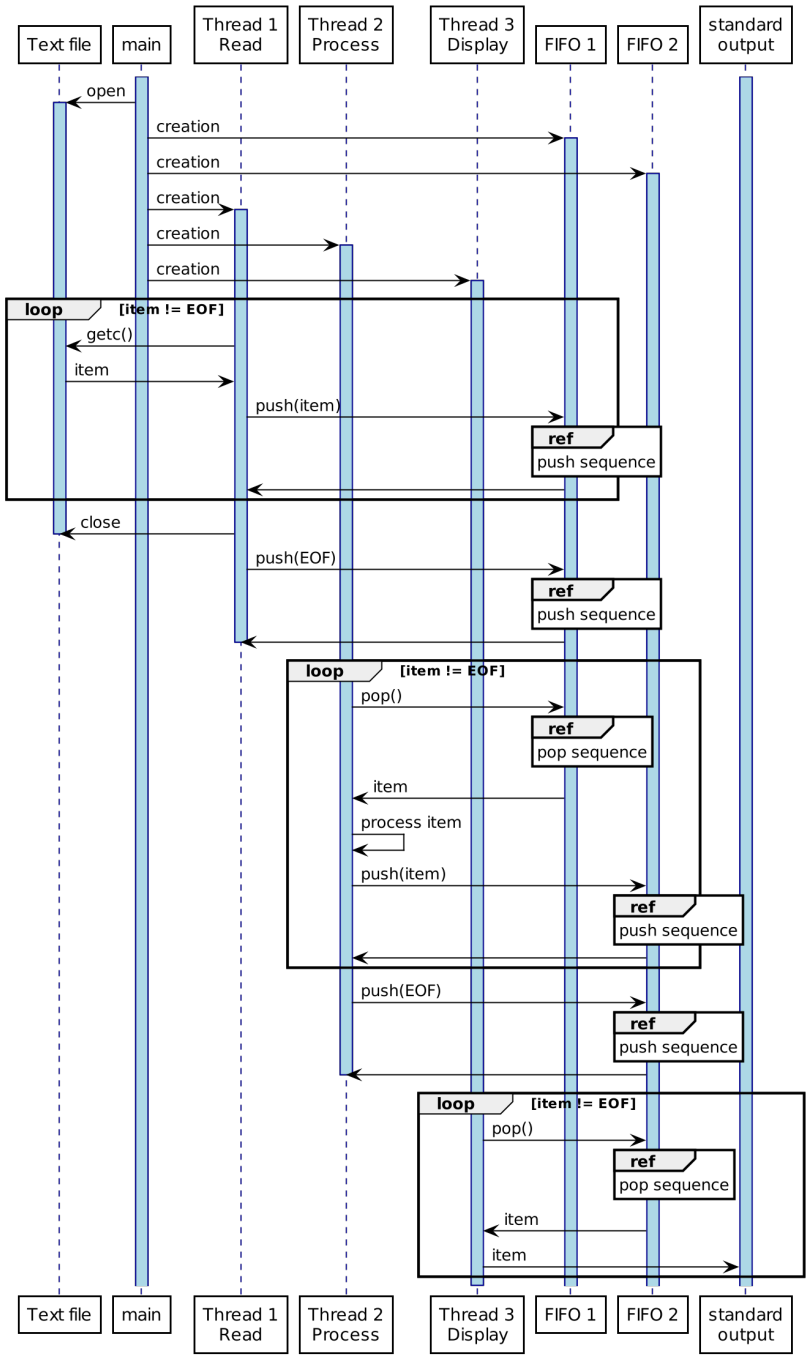
Exercice 2 – Producteurs-Consommateurs

1. Écrire l'implémentation d'une FIFO en C. La FIFO possèdera un tableau de caractères de taille `N`, deux indices (`entree` et `sortie`) qui serviront à pointer le prochain élément à entrer et à sortir de la FIFO ainsi que des méthodes `void push(char element)` et `char pop()` qui serviront à rentrer et à sortir un élément de la FIFO. (Fichiers *fifo.c* et *fifo.h*)
2. Réaliser un programme qui réalisera les opérations suivantes (fichier *prodcon.c*) :



   - Le thread *Lecture* sera implémenté par une fonction et lira un fichier texte caractère par caractère. Chaque caractère lu sera ajouté à la FIFO *buf1*.
   - Le thread *Calcul* sera implémenté par une fonction et lira ses données hors de la FIFO *buf1*. Chaque caractère lu sera alors converti en majuscule avant d'être envoyé dans la FIFO *buf2*.
   - Le thread *Affichage* sera implémenté par une fonction et lira ses données hors de la FIFO *buf2*. Chaque caractère lu sera affiché sur la console.
3. Gérer les protections en cas de sur et sous alimentation des FIFO à l'aide de sémaphores au sein du code de la FIFO.
4. Tester votre programme avec différentes tailles de FIFO (N=1, N=2, N=10, ...)
5. Dessiner un diagramme temporel montrant le séquencement de cotre programme.

1 Diagramme de séquence



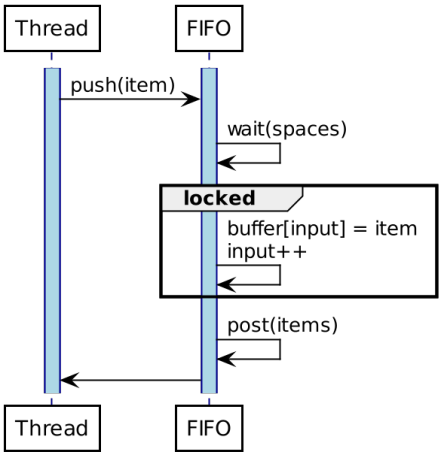
Les trois threads sont activés en même temps (lors de leur création par le `main`). Cela signifie par exemple que le thread *Process* peut appeler la fonction `pop()` de manière concurrente au `push()` du thread *Read*. C'est au sein de ces méthodes que sont implémentées l'attente d'un item disponible avant de le récupérer ou l'attente d'un espace libre avant d'insérer un nouvel item dans le buffer.

Un même thread ne peut par contre pas lancer plusieurs `push(item)` ou `pop()` en parallèle : ces méthodes bloquent le thread et doivent se terminer avant que le thread ne puisse continuer.

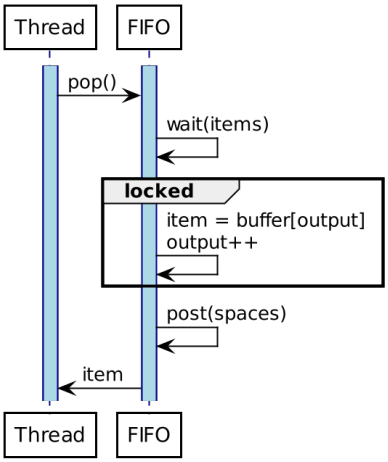
Le thread *Read* lit les caractères du fichier texte donné en paramètre au programme et les ajoute un à un à la *FIFO 1*. Le caractère `EOF` est lui aussi envoyé à la FIFO pour que le second thread sache quand s'arrêter. Il s'arrête alors.

Le thread *Process* lit un à un les caractères de la *FIFO 1* pour les mettre en majuscules et les insérer dans la *FIFO 2*. Ce thread envoie lui aussi le caractère `EOF` à la *FIFO 2* avant de s'arrêter.

Le thread *Display* lit les caractères de la *FIFO 2* un à un et les affiche en console. Il s'arrête dès qu'il a lu le caractère `EOF`.



(a) push sequence



(b) pop sequence

Les FIFO gèrent la protection de l'accès à leurs buffers par un mutex. Pour éviter de bloquer la lecture du buffer alors qu'il est vide, le sémaphore `items`, initialisé à 0, est incrémenté à chaque fois qu'un nouvel item est inséré dans le buffer et décrémenté à chaque fois qu'un item est lu. Pour éviter l'insertion d'un item dans un buffer déjà plein, le sémaphore `spaces`, initialisé à la taille du buffer, est décrémenté à chaque fois qu'un item est ajouté au buffer et incrémenté à chaque fois qu'un item est lu.

2 Compilation

La compilation est réalisée en utilisant cmake via le fichier *CMakeLists.txt* :

```
[1]  cmake_minimum_required(VERSION 3.10)

[2]  project(Exercice2_Producteurs_Consommateurs VERSION 1.0)

[3]  set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall -Wpedantic -Wextra")

[4]  include_directories(PUBLIC include)

[5]  set(SOURCE_FILES
         src/prodcon.c
         src/fifo.c)

[6]  find_package(Threads REQUIRED)

[7]  add_executable(prodcon ${SOURCE_FILES})

[8]  target_link_libraries(prodcon PRIVATE Threads::Threads)
```

`[1]` version de `cmake` utilisée
`[2]` nom et version du projet
`[3]` ajout des flags habituels pour l'affichage de tous les warnings
`[4]` le répertoire *include* contient les headers nécessaires à la compilation
`[5]` définition de la variable `SOURCE_FILES` contenant les fichiers sources du projet
`[6]` ajout du package pour l'utilisation des threads
`[7]` la compilation des fichiers sources produit l'exécutable `prodcon`
`[8]` le projet utilise la librairie `Threads` du package `Threads`

La commande `cmake ..` lancée depuis le sous-répertoire *build* permet de générer le *Makefile* :

```
lbin@tr-ubuntu18-server:~/2021_TR/ex2-producteurs-consommateurs/build$ cmake ..
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /home/lbin/2021_TR/ex2-producteurs-consommateurs/build
lbin@tr-ubuntu18-server:~/2021_TR/ex2-producteurs-consommateurs/build$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile  test.txt
```

Par défaut, le *Makefile* est généré en mode debug. Pour le générer en mode release, j'utilise la commande `cmake -DCMAKE_BUILD_TYPE=Release` ***`path`***, où ***`path`*** est l'endroit où se trouve le fichier *CMakeLists.txt*.

La commande `make` me permet de compiler le projet à partir du *Makefile* pour produire l'exécutable *prodcon* :

```
lbin@tr-ubuntu18-server:~/2021_TR/ex2-producteurs-consommateurs/build$ make
Scanning dependencies of target prodcon
[ 33%] Building C object CMakeFiles/prodcon.dir/src/prodcon.c.o
[ 66%] Building C object CMakeFiles/prodcon.dir/src/fifo.c.o
[100%] Linking C executable prodcon
[100%] Built target prodcon
lbin@tr-ubuntu18-server:~/2021_TR/ex2-producteurs-consommateurs/build$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile  prodcon  test.txt
```

3 Tests

Le programme est testé avec des temporisations aléatoires dans les threads (entre 1 et 20 000 µs). Il attend en paramètres le nom du fichier à lire et la taille des buffers :

```
lbin@tr-ubuntu18-server:~/2021_TR/ex2-producteurs-consommateurs/build$ ./prodcon test.txt 1
BANANA PANCAKES (POUR 2)

3/4 TASSE DE FARINE
+ 1CS SUCRE RAPADURA
+ 2CC BAKING POWDER
+ 1 PINCEE SEL

1 PETITE BANANE (ECRASEE)
+ 1 GROS OEUF
+ 1/2 TASSE DE LAIT
+ 2 CS DE BEURRE ET LE MELANGE DE FARINE
(JUSTE INCORPORER, PAS TROP MELANGER)

+ 1 BANANE EN MORCEAUX
```

```
lbin@tr-ubuntu18-server:~/2021_TR/ex2-producteurs-consommateurs/build$ ./prodcon test.txt 1000000000
BANANA PANCAKES (POUR 2)

3/4 TASSE DE FARINE
+ 1CS SUCRE RAPADURA
+ 2CC BAKING POWDER
+ 1 PINCEE SEL

1 PETITE BANANE (ECRASEE)
+ 1 GROS OEUF
+ 1/2 TASSE DE LAIT
+ 2 CS DE BEURRE ET LE MELANGE DE FARINE
(JUSTE INCORPORER, PAS TROP MELANGER)

+ 1 BANANE EN MORCEAUX
```

Fichier d'origine utilisé pour le test :

```
lbin@tr-ubuntu18-server:~/2021_TR/ex2-producteurs-consommateurs/build$ pr test.txt


2021-03-03 19:22                    test.txt                    Page 1


Banana pancakes (pour 2)

3/4 tasse de farine
+ 1cs sucre rapadura
+ 2cc baking powder
+ 1 pincee sel

1 petite banane (ecrasee)
+ 1 gros oeuf
+ 1/2 tasse de lait
+ 2 cs de beurre et le melange de farine
(juste incorporer, pas trop melanger)

+ 1 banane en morceaux
```