Introduction au Système

Projet dictionnaire multi-processus

“D.I.C.O”

BELLATIF Mamoune

BARKAOUI Meriam

Année 2021/2022

Université Grenoble Alpes
L3 MIAGE

Installation:

Le répertoire courant contient le makefile. *src/* contient les fichiers .sources .c et le répertoire include/ contient les header .h, le répertoire *bin/* est créé lors de l'exécution de make, et contient l'exécutable *dico.* Un répertoire obj/ est momentanément créé lors de l'exécution de make, puis supprimé par clean.
Pour compiler le programme, lancer la commande "*make*" dans le répertoire qui contient le makefile. make clean sera exécuté automatiquement,
Lors du lancement du programme (bin/dico <arg>) un fichier "dico.dat" sera créé automatiquement là où vous lancerez la commande, cela servira de fichier de sauvegarde.

Fonctions implémentés:

Toutes les fonctionnalités demandées (exit, set, lookup, dump) ont été correctement implémentées et testées, deux fonctionnalités supplémentaires ont aussi été implémentées, save et delete, pour la persistence des données et la suppression d'une entrée: voir section fonction supplémentaires.

Limitation:

Lors du chargement d'une sauvegarde, il se peut que la liste chainé soit chargée dans le mauvaise ordre (mais dans le bon processus néanmoins), malheureusement nous avons manqué de temps pour régler ce problème

Exécution globale:

L'exécution du programme dans la fonction *main* se déroule comme suit:
L'appel à la fonction init, va initialiser les tubes et leurs descripteurs de fichier: un tableau d'entiers de taille 2 appelé ***f_master (***`int f_master[2])` pour le tube du père, et un tableau d'une taille N (argument du programme) dont chacun des **élément** est un un tableau de taille 2 (`int (*f_node)[2])`, appelé ***f_node***, qui contient donc les **descripteurs** de fichiers des tubes des nodes. Tout cela est fait avant la création des processus fils, un appelle a la fonction pipe sur les éléments de f_node est réalisé ensuite dans un boucle pour ouvrir les tubes pour les futurs processus fils.

Exécution des fils:

Chaque processus fils (node) va en tout premier lieu initialiser et affecter sa propre liste chainé a partir de la sauvegarde précédente (voir section *sauvegarde*, fonction *chargerSauvegarde*), puis fermer les descripteurs utilisé (*voir section gestion des pipes*), Il attend de lire une requête, puis en vérifie la nature, la principale condition de l'execution d'une requete repose sur la vérification du ***flag***, quelque soit la requête, il doit être égal a "***IN_PROGRESS***", ensuite si la commande est DUMP, le node exécutera la commande sinon, on *appellera* une simple fonction *checkNode (src/dico_node.c)* qui vérifie que la différence entre la clé et l'index du node est un multiple du nombre de processus, si c'est le cas, c'est que le node est bien chargé de la clé, se chargera de l'exécution de la requête, modifiera la valeure du flag à "***DONE***" puis enverra la *requête* avec un appelle a la fonction *sendRequest* (le destinataire dépendra de plusieurs paramètres, voir *communication*)

Exécution du processus père:

Le processus père (ou Controller)
va commencer avec l'appel à la fonction (*ouvrirDescripteur* voir section tube/pipes), puis entrer dans la boucle d'exécution principale (qui a pour condition de sortie, la commande a 0), puis demandera la saisie de la commande.
Afin de sécuriser cette saisie, nous avons ajouté une fonction *saisieEntier (src/dico_utils.c)* qui s'assure que l'utilisateur rentre bien un entier, et non pas un caractère alphabétique ou spécial, et vide ce qui reste dans l'entré standard, cela nous permet d'éviter tout soucis que peut provoquer une mauvais entrée. le controler envoit ensuite la requête et attend une réponse, puis informera l'utilisateur de l'exécution et de quel processus s'en est chargé (voir section message et section communication)

Message et information:

Pour que la **communication** entre les processus (***Controller*** et ***Nodes***) soit efficace, il est nécessaire de mettre en place une **nomenclature** de message **simple, claire**, **concise** et **complète**.
Il se dégage de l'exercice trois informations différentes qui seront primordiale à l'exécution du programme:

1. **La commande choisie**
2. **La clé**
3. **La valeur**

Pour communiquer, les processus enverront une ***requête (request)*** qui contiendra ces trois informations.

En plus de cela, il est important que les processus puissent **communiquer** sur **l'état d'une requête**, cela permet d'éviter qu'un processus **exécute** une requête plusieurs fois.
Le ***Controller*** devra aussi savoir si la requête a **réussie** ou **échouée**.
Pour cela, on ajoutera donc aux 3 informations ci-dessus une quatrième information ***FLAG.***

En ce qui concerne les types d'information, les **commandes** sont représentées par des ***entiers (int)***: **0** pour ***Save and Exit***, **1** pour ***Set***, **2** pour ***Lookup***, **3** pour ***Dump***, et enfin **4** pour ***Delete***, que l'on a ajouté en plus (*voir section fonctionnalités supplémentaires*). De même pour la **clé**, qui ne peut être qu'un entier.

Par soucis de lisibilité et de maintenance du code, on utilisera des ***macros*** pour représenter les commandes, toutes définis dans le fichier *"include/dioc_macros.h".*

```
#define EXIT 0
#define SET 1
#define LOOKUP 2
#define DUMP 3
#define DELETE 4
```

La **valeur** à enregistrer quand a elle peut représenter n'importe quelle suite de caractère **alphanumérique**, donc une **chaîne de caractère (**char***).

La dernière information, le ***FLAG*** devra représenter 3 états possibles d'une requête:

1. **Non exécutée (*IN_PROGRESS)***

2. **Exécutée et réussi (*DONE*)**
3. **Exécutée et échouée *(FAIL)***

Comme les *FLAGS* représentent que 3 valeurs simples et déterminés, par soucis de **praticité**, on les représente comme suit sous forme d'entiers (int) et *macros:*

```
#define IN_PROGRESS 0
#define DONE 1
#define FAIL -1
```

Enfin, on **concatène** dans une chaine de caractère ***buffer*** ces 4 informations toujours de la manière qui suit:

'flag commande clé valeur'
Exemple: *IN_PROGRESS SET 1998 "QOTSA"*
(0 1 1998 "QOTSA")

Un problème se pose, on remarque assez vite que certaines commandes n'ont pas besoin d'une clé ou d'une valeure, le *controller* se chargera alors de rendre les informations inutiles ***nulles:***

```
strcpy(value,"null");
key=0;
```

Le message sera donc **construit** dans une fonction ***sendRequest*** (qui s'occupe aussi de l'envoie de la requête, voir section communication) appelé par le controller qui prend en paramètre les information puis construit une chaine de caractère **tampon *buffer*** avec "***sscanf"***:

```
sscanf(buffer, "%d %d %d %s", flag, cmd, key, value);
```

Et sera lu par les processus avec "***sptrinf***"

```
sprintf(buffer,"%d %d %d %s",flag, cmd, node_index, value);
```

Communications et synchronisation:

Communication:

Toute la partie **communication** est gérée par **3 fonctions** présentes dans le fichier *dico_communication.c,* une fonction ***readRequest***, une fonction ***sendRequest*** et une fonction ***syncNode***, ces fonctions seront utilisé aussi bien par le ***controller (processus père)*** que pas les ***nodes*** (**processus fils)**, pour savoir où et quand écrire/lire.

On décompose le problème:

La vie d'une requête (mis à part pour "*save and exit*" et "*dump"*) se déroule de la manière suivante:
Le controller envoie au premier node la requête, si la requête correspond, elle est traité, puis le node répond au controller. Si requête n'est pas traité, le node envoie la requête au node suivant, ainsi de suite jusqu'à ce qu'un node le prend, si on fait le tour des nodes, le dernier node la renverra au père.

On reviendra un peu plus tard sur les commandes particulières "dump" et "save and exit" qui nécessitent un fonctionnement à part pour la synchronisation

Gestion de l'envoie d'une requête:
Quatre situation d'envoie d'une requête qui englobent toute l'exécution, les quatres sont gérés par la seule fonction *sendRequest.*
Pour déterminer le bon destinataire de la requête, la fonction prend en paramètre un entier node_index, qui est l'index du node a l'origine de la requête (-1 si c'est le Controller, macro *MASTER*), la commande a envoyer, la valeure, la clé et le flag.
Avant de d'envoyer la requête, les arguments du message sont construite dans un *buffer* comme vu dans la section message
Deux cas de figure sont possible pour le *controller* et deux cas pour les *nodes* :

Cas du *Controller*:

1.Si la commande est un Dump ou un EXIT, on parcours le tableau de descripteurs nodes et on enverra la requête a chaque noeud en appelant la fonction write sur le descripteur en écriture:

```
for (int j = 0; j < n; j++){
write(f_node[j][1], buffer, 100*sizeof(char));}
```

2. Si la commande est un SET, LOOKUP, ou DELETE, on écrit dans le dernier descripteur, qui est celui où le node numéro 1 a un accès en lecture

```
write(f_node[n-1][1], buffer, 100*sizeof(char));
```

Cas du *Node*:

1. Cas le plus généraliste, si un node doit faire passer la requête au node suivant le principe est simple, chaque node d'index *node_index* doit forcément *écrire* dans le tube *node_index,* donc:

```
write(f_node[node_index][1], buffer, 100*sizeof(char));
```

2. Si le node s'est effectivement chargé d'exécuter la requête, donc si *flag* a pour valeure *DONE* ou *FAIL*: On écrit dans le tube du Controller (ici f_master)
   Petite précision: dans ce cas de figure (dans ce cas de figure le buffer est reconstruit pour inclue l'index du node qui envoie la requête voir section reception d'une requete)

```
write(f_master[1], buffer, 100*sizeof(char));
```

(un dernier cas de figure qui n'est pas censé arriver, mais qu'on a mis par sécurité: Si pour une raison X, aucun node n'arrive à vérifier la clé, et qu'il n'y a aucune exécution, le dernier *node* se chargera de renvoyer la requête tel quelle au père)

Gestion de la réception d'une requête:

Comme pour l'envoie d'une requête, toutes les situation sont gérés par la fonction *readRequest*, mais cette fois, on a que **trois** situations possibles:
readRequest prend en paramètre, l'index du node, ou du controller(-1/MASTER), un pointeur vers la commande, un pointeur vers la valeure, un pointeur vers la clé, et un pointeur vers le flag.

Controller:

Après avoir envoyé une requête, le controller attendra une réponse, c'est son seul a appel a la fonction readRequest, il lit la requête reçu dans son tube, la place dans une chaine de caractère buffer, et la déconstruit comme vu dans la section message:

```
read(f_master[0], buffer, 100*sizeof(char));
sscanf(buffer, "%d %d %d %s", flag, cmd, key, value);
```

*Petite précision, ici la clé *key* n'est pas la clé traitée, mais l'index du node qui a executé la requete (cela evite d'avoir un message avec un surplus de champs inutiles), ainsi le Controller sera au courant de la commande exécutée, la réussite ou non de l'exécution, et le node responsable de l'exécution.

Node:
Un node commence toujours sa boucle d'exécution par une lecture dans le tube approprié

1. Si le node qui reçoit la requête est le premier node, il va faire une lecture sur le descripteur correspondant au dernier tube, donc n-1 (n étant le nombre de node), donc
```
read(f_node[n-1][0], buffer, 100*sizeof(char));
```

2. Si le node qui reçoit est n'importe quel autre node, il va lire dans le node qui le précède, donc `f_node[node_index-1][0]`
```
read(f_node[node_index-1][0], buffer, 100*sizeof(char));
```

Comme pour le *controller*, les commande, clé valeur et flag sont construite ensuite à partir du buffer comme vu dans la section message

Synchronisation (*DUMP* et *SAUVEGARDE*):

La vie d'une requête "*dump*" ou "*save and exit*" se déroule de la manière suivante:

Le controller envoie la requête a tous les nodes, mais chaque node attendra que le node qui le précède lui envoie un **jeton** ***(token)***. Lorsque le node exécute la requête, il enverra à son tour un *token* au *node* suivant (Bien évidemment, le premier *node* n'attendra pas de *token*, car il sera le premier à démarrer l'exécution. Le dernier node n'enverra pas de token, mais informera le père de l'exécution de la requête). Ce mécanisme de *tokens (*géré par l'appel à la fonction *syncNode)* permet donc au *nodes* de s'exécuter un après l'autre et s'assure que le controller n'affiche pas d'invite de commande avant que les nodes aient fini car il devra attendre la réponse du dernier *node*.

La fonction syncNode prend trois paramètres: l'index du node, un *flag* (IN_PROGRESS, DONE, FAIL) et la commande. elle doit être appelée au début de la partie du code à

synchroniser avec un *flag IN_PROGRESS*, et à la fin de la partie a synchroniser avec un *flag DONE*

Pour résumer notre protocole d'échange, les nodes se mettent d'abord en lecture, en attente de réception d'une requête, puis vont soit exécutés, soit faire circuler la requête au suivant, puis se remettent en attente de lecture, le Controller démarre en construisant une requête qui contient les information nécessaire, et attend quoi qu'il arrive une réponse, et on recommence…

Gestion des ***tubes/pipes:***

Pour pouvoir communiquer, que ce soit en lecture ou écriture, chaque node devra écrire dans des tubes, accessibles depuis des descripteurs de fichiers, présents dans des tableau en tant que variable globale (les descripteurs de fichiers étant utilisé par plusieurs fonctions, après de multiples appels de fonction, on décide de les mettre en variables globales pour éviter du d'avoir trop de paramètres, cela simplifie le code et le rend plus lisible.

Comme l'initialisation des tubes et descripteurs se fait avant la création des processus fils, il faut pour chaque processus, fermer les descripteurs qu'il n'utilisera pas, pour faire en sorte de laisser ouvert, seulement un descripteur qui lui permet d'écrire, et un descripteur qui lui permet de lire: Cela est géré par la fonction *ouvrirDescripteur(node index)* (Le nom peut paraître contradictoire, mais, ici, ouvrir à pour sens de "laisser ouvert" les descripteurs que l'on souhaite utiliser. aussi cela évite la confusion avec la fonction *fermerDescripteur*).

Descripteur utilisés par un Node:
Un node d'index i aura besoin de 3 descripteurs, chacun correspondant a un tube différents. Ce node devra lire sur le tube i-1, et devra écrire sur le descripteur du tube i. Il pourra aussi écrire dans le tube du Controller sur son descripteur en écriture.

Maintenant que l'on sait quels descripteurs vont rester ouvert, on ferme le reste, donc chaque node devra parcourir le tableau de descripteur pour les fermer un à un.
Pour ce faire, la fonction prend en paramètre l'index du node, (ou -1 a.k.a "MASTER" si controller), puis parcours f_node[] (qui est le tableau de tubes fils), a chaque itération du

parcours, on compare l'index du node avec l'indice d'itération, et on ferme les descripteurs qui ne seront pas utilisé, ç-a-d, tous les descripteurs différent de f_node[i][1] et f_node[i-1][0], une exception pour le premier node, qui n'a pas de tube qui le précède, donc lira sur le tube du dernier node (f_node[node_count-1][0].

Le détail des conditions sont expliquées en commentaire dans le code.

Descripteur utilisés par le Controller:
Pour le controller, on doit garder ouvert le descripteur (f_master) en lecture, et chaque descripteur en écriture de chaque node, ici aucune exception, un simple parcours de tous les tubes suffit.

et puis selon la valeur de l'index décidera de quels descripteurs fermer, chaque processus fera appelle à cette fonction.
lorsque l'utilisateur choisit d'arrêter le programme, il faut s'occuper de la fermeture des descripteur avant de complètement terminer les processus

En sortie de la boucle principale d'exécution de chaque processus, un appel à la fonction *fermerDescrpteur* se charge de fermer les descripteurs que l'on a utilisé, on réalise donc l'opposé de la fonction ouvrirDescripteurs
'

Vie et **mort** des **processus:**

La fonction main (src/dico_main.c) s'assure d'initialiser les tubes, et d'appeler *fork()* 'n' fois ('n' étant le nombre de processus donné en argument).
Ensuite la fonction *node* est la seule appelé par chaque main du processus fils (vérifié par la valeure retournée par *fork()*) Avec en paramètre un **indice**(***index***) (l'index est tout simplement l'indice de l'itération de la boucle qui appelle fork(), par exemple pour le 2ème appel au fork, le node aura pour index 2) c'est index permettra d'identifier chaque node séparément, ce qui sera essentiel pour la communication et pour la gestion des tubes.
Quand node(i) a finis de s'exécuter, (c'est a dire quand l'utilisateur choisi d'arrêter le programme), le processus meurt (exit(0)) l'appelle a exit est essentiel pour éviter que les processus fils n'aillent exécuter eux même du code réservé au processus père.

Le processus père, après avoir finis de créer les processus fils, va exécuter sa fonction controller, qui se charge donc de faire l'interface entre l'utilisateur et les processus fils.
quand on sort de la fonction controller, on attend bien la mort de tous les processus fils avant de terminer l'exécution

```
while (wait(NULL)!=-1);
```

Fonctionnalités supplémentaires implémenté et execution de requête:

Exécution de la requête:
Lorsque la clé est vérifiée et l'exécution lancée, le node vérifie l'existence de la clé dans la table, si elle existe déjà, pour éviter de rentrer deux fois la même clé. l'exécution se fait par l'intermédiaire de la fonction *execRequest(src/dico_node.c),* qui en fonction de la commande choisi, fera l'appel à la fonction correspondante a la commande, et retournera un flag DONE si succès, ou FAIL si echec

Pour compléter la bonne manipulation des donnés, nous avons ajouté, deux fonctionnalités:

1. la persistance des données lorsqu'on arrête le programme et qu'on le reprend, nous avons ajouté deux fonctions (présente dans src/dico_save.c), *sauvegarde* et *chargerSauvegarde*
2. la suppression d'une entrée: La fonction supplémentaire *delete* se situe dans le fichier (src/dico_utils.c)

Sauvegarde:

La sauvegarde est effectuée par chaque node dans un fichier save.dat a la fin de l'exécution du programme, afin que les processus n'écrasent pas les sauvegarde des autres, nous avons forcé la sauvegarde à se faire de manière synchronisée à la manière d'un DUMP. La sauvegarde est une simple écriture dans le fichier de la struct qui représente chaque entrée de la liste.

Le chargement se fait lors de l'initialisation de chaque table, chaque node lira dans le fichier les struct enregistré dans l'ordre, puis vérifiera si la clé correspond au node, si c'est le cas, l'entré sera enregistrée dans la table

Suppression:

Pour la suppression, on parcours la liste chainé, lorsqu'on tombe sur l'entré que l'on souhaite supprimer, on fait une copie de l'adresse de l'élément suivant, une copie de l'adresse de l'élément précédent, et on modifie la variable next du précédent pour la faire pointer sur l'adresse de la suivante, ainsi, l'élément que l'on a voulu supprimer n'est plus connecté à la liste chainé
table.c:

Le fichier table.c a légèrement été modifié pour ne pas afficher de table nulle

Débogage:

Pendant le développement du programme, nous avons fait face à plusieurs comportements inhabituels et arrêts inattendus; à l'aide de *GDB*, nous avons pu afficher à chaque étape de l'exécution l'état des variables et avancer pas à pas dans le programme, ce qui nous permet de cibler l'origine des divers problèmes que l'on a pu rencontrer

```
Saisir commande (0 = exit, 1 = set, 2 = lookup, 3 = dump): 0
200             cmd=atoi(commande_input);
(gdb) print cmd
$1 = 0
(gdb) next
201             if((strcmp(commande_input,"0"))≠0){ //si input == 0 et non pas une lettre
(gdb) next
236                 strcpy(value,"null"); //on met la valeure aa null au cas on arrete a la pr
emiere iteration
(gdb) step
237                 key=0; //on met la clé a 0 au cas on arrete a la premiere iteration
(gdb) next
238                 sendRequest(MASTER, cmd, value, key, PROCESSING); // infanticide
(gdb) step
sendRequest (i=0, cmd=60, value=0x0, key=0, flag=0) at projet_3_bak.c:112
112     void sendRequest(int i, int cmd, char *value, int key, int flag){//status flag (0, non
 traité envoyer au prochai noeud, 1 done: envoyer au père | WAITING | SUCCESS | FAIL
(gdb) print value
$2 = 0x0
(gdb) print cmd
$3 = 60
(gdb) backtrace
#0  sendRequest (i=0, cmd=60, value=0x0, key=0, flag=0) at projet_3_bak.c:112
#1  0x0000555555555dcf in controler () at projet_3_bak.c:238
#2  0x0000555555555ec9 in main (argc=2, argv=0x7fffffffe068) at projet_3_bak.c:262
(gdb)
```

De même, la commande top dans un terminal à part nous permet de vérifier en temps réel de l'état des processus, et de s'assurer qu'on a pas de petits fils ou de processus zombie

```
mamoune@mamoune-pc: ~
top - 16:18:39 up 4 days, 17:08,  0 users,  load average: 0.01, 0.06, 0.04
Tasks:  39 total,   1 running,  37 sleeping,   1 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.2 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   6281.4 total,   2747.1 free,   1893.9 used,   1640.4 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   4192.2 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
27309 mamoune   20   0 1060048 206752  35172 S   2.0   3.2   3:30.87 node
  402 mamoune   20   0   89548  16204   7456 S   1.7   0.3   0:30.22 WD-TabNine
27648 mamoune   20   0 1679516  56792  15512 S   0.3   0.9   0:19.75 cpptools
27690 mamoune   20   0 3769620  98716  57328 S   0.3   1.5   0:08.00 vsls-agent
    1 root      20   0    1204    812    468 S   0.0   0.0   0:00.91 init
 8237 mamoune   20   0   58488  43592  29652 S   0.0   0.7   0:00.18 gdb
 8305 mamoune   20   0    2496    716    608 t   0.0   0.0   0:00.00 projet
 8309 mamoune   20   0    2496    296    184 S   0.0   0.0   0:00.00 projet
 8310 mamoune   20   0    2496    296    184 S   0.0   0.0   0:00.00 projet
 8311 mamoune   20   0    2496    296    184 S   0.0   0.0   0:00.00 projet
 8312 mamoune   20   0    2496    296    184 S   0.0   0.0   0:00.00 projet
 8313 mamoune   20   0    2496    296    184 S   0.0   0.0   0:00.00 projet
21457 root      20   0    1284    384     20 S   0.0   0.0   0:00.00 init
21458 root      20   0    1284    384     20 S   0.0   0.0   0:00.58 init
21459 mamoune   20   0   10056   5160   3424 S   0.0   0.1   0:00.05 bash
21504 root      20   0    1284    384     20 S   0.0   0.0   0:00.00 init
21505 root      20   0    1284    384     20 S   0.0   0.0   0:01.56 init
```