John-Michael JENY JEYARAJ 12013787

Système et Réseau Rapport

Description des fonctionnalité mise en oeuvre

Comptine_utils.c

read_until_nl - Pour lire les envoie caractère par caractère jusqu'au premier retour a la ligne et on le met dans un buffer.

Est_nom_fichier - On teste si le string contient un fichier .cpt, en regarder si il fini par ., c, p et t et qu'il contient au moins 5 caractères.

Init_cpt_depuis_fichier - On alloue de l'espace sur le tas et on met dedans le nom du fichier et son titre.

Creer_catalogue - On crée un arbre du répertoire, avec les noms des fichiers. On utilise dirent.h avec readdir pour d'abord trouver la taille qu'on veut allouer. Puis rewinddir pour mettre le pointeur au debut de la liste du répertoire et finalement readdir pour remplir le catalogue.

Wcp_clt

Creer_connect_sock - Une fonction standard pour créer un connexion et retourne la socket.

Recevoir_liste - Lit l'envoi de la liste par le serveur, ligne par ligne, et l'affiche sur le terminal. Il arrête de lire quand le serveur envoie '\n\n'.

Saisie_num_comptine - Le client écrit un numéro correspondant à la comptine qu'il veut lire. On fait des tests sur l'entrée du client.

Envoyer_num_comptine - Transforme le numéro choisi par le client en network byte order, puis l'envoie au serveur.

Afficher_comptine - Lit le retour du serveur ligne par ligne et les affiche sur le terminal. Il arrête de lire quand le serveur envoie '\n\n'.

Wcp_srv

Creer_configurer_sock_ecoute - Une fonction standard pour écouter une connexion avec un client. On utilise bind pour attacher la connexion au port. On met aussi une fonction accept dans une boucle infini pour accepter les clients.

Envoyer_liste - On met le catalogue dans un buffer et on l'envoie ligne par ligne, suivi d'une ligne vide.

Recevoir_num_comptine - On lit le numéro de la comptine demandé par le client et le transforme depuis network byte order avec ntohs en host byte order.

Envoyer_comptine - On ouvre la comptine demandée par le client et l'envoie la comptine demandée au client. Puis on envoie \n\n pour terminer l'envoie.

Descriptions d'ajouts au protocole WCP

Serveur Multithreadé

Dans le TP9, j'ai vu comment créer un serveur multithreadé. Pour faire en sorte que le serveur puisse être connecté avec plusieurs clients en même temps, le serveur crée une nouvelle connexion pour chaque client. Donc on crée un thread dans chaque connexion, avec une fonction worker et une structure work.

La structure work contient la socket, une structure catalogue fonction, l'adresse et le dirname (pour un répertoire quelconque).

Dans la fonction worker, qui prend comme argument la structure work, je copie le catalogue pour que chaque client ait une version locale du catalogue avec laquelle ils peuvent interagir. Ceci est nécessaire car, si un autre client téléversait une comptine au serveur, on aurait un changement au catalogue du serveur, et donc quand le premier client demande de voir une comptine, l'indice des comptines serait faux.

Client peuvent faire plusieurs demande à la suite

Cette étape était très simple avant d'avoir fait plusieurs approfondissement, car il suffit de mettre les fonctions pour recevoir_liste_comptine, saisir_num_comptine, envoyer_num_comptine et afficher liste et envoyer comptine dans une boucle infini. Et le client choisit entre soit des un numéro pour une comptine où "un caractère spécial" pour quitter que j'avais choisis est le nombre de comptine + 1.

Après avoir fait plusieurs fonctions et un menu, je devais changer la condition pour que le client quitte. Et donc maintenant que le client n'était pas obligé d'entrer des numéros, mon serveur est devenu susceptible quand le client quitte brutalement avec un CTRL+C où même CTRL+D.

Ma solution pour ce problème était de soigneusement traiter les entrées du client et fermer la socket quand il y a une erreur. Et du côté serveur, je devais être sûr qu'il écrit seulement à une socket ouverte, donc il écrit au client si et seulement si il reçoit des donnés par le client.

Le serveur maintient à jour un log

Pour maintenir un log des actions des clients, j'ouvre le fichier log.txt avec APPEND qui ajoute ce qu'on écrit à la fin du fichier.

Écrire dans un fichier est un action critique, donc j'ai utilisé mutex pour laisser écrire seulement une demande à la fois.

Téléverser une comptine

Pour que le client puisse téléverser une comptine au serveur, j'ai pris inspiration avec le partiel le l'année dernière pour le côté serveur et du code pour la fonction envoyer_comptine du serveur pour le côté client.

Tout d'abord il faut vérifier les entrées du client. On demande le nom de la comptine à téléverser, donc on doit tester s'il est bien un fichier comptine.
Du côté serveur, on lit le nom du fichier envoyer et on le crée s'il n'existe pas. S'il existe, j'ai décidé d'utiliser O_TRUNC, qui vide le contenu du fichier.

Le client envoie le contenu de la comptine et le serveur les reçoit ligne par ligne avant de l'écrire dans le fichier.

Créer une comptine

L'envoie des donnés utilise la même méthodologie que téléverser une comptine. Mais pour que le client puisse utiliser des espaces dans leur comptine, j'ai utilisé des fgets [5]. J'avais beaucoup de problèmes avec la fonction scanf donc j'ai cherché en ligne comment les résoudre en ligne. J'ai trouvé la fonction fgets qui lit les entrées du client jusqu'à qu'il envoie une ligne vide.

Le client envoie d'abord le nom du fichier, et on teste si c'est bien un nom de fichier,puis le serveur le lit et crée un fichier avec ce nom.

Puis le client envoie le titre et le contenu de la comptine ligne par ligne.Le client peut mettre des retour à la ligne dans la comptine en écrivant '\n' dans le terminal. Et le programme arrête de lire les entrées si le client envoie un ligne vide. Dans le côté serveur, on lit le titre avec un read_until_nl, et on l'écrit dans la nouvelle comptine avec un \n. Puis on lit le reste avec read_until_nl.

Problèmes rencontrés

Catalogue

À mon avis, il n'y avait pas assez d'informations dans le TP0 pour bien comprendre comment utiliser les fonctions, et en particulier rewinddir, de dirent.h. Je n'arrivais pas à comprendre comment avancer dans le répertoire et comment aller en arrière. J'ai regardé une vidéo par CodeVault [3], qui à soigneusement expliqué comment utiliser les fonctions utiles. Mon changement comporte d'abord d'utiliser readdir dans une boucle pour compter le nombre de comptine dans le répertoire et on alloue de l'espace, puis on rewinddir pour recommencer depuis le début du répertoire. Finalement, on utilise readdir dans une boucle encore pour remplir les noms de fichier dans le catalogue.

SIG_PIPE

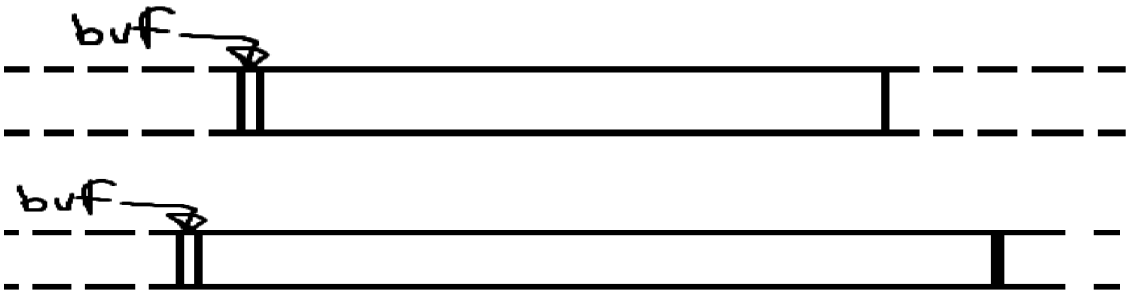
Pendant le développement de mon serveur, j'ai rencontré un problème avec le signal SIG_PIPE. Le problème apparaît quand le client fait une entrée non conforme comme CTRL+C ou CTRL+D et il quitte le programme brutalement. J'ai cherché en ligne la réponse pour résoudre ce problème sur stackoverflow [4] qui m'a bien éclairée sur comment les échanges du serveur et du client fonctionne. Ma première tentative était d'utiliser <signal.h> et de faire signal(SIGPIPE, SIG_IGN) dans le main. Mais, j'ai vu en ligne que c'est mieux de correctement gérer les transferts que d' ignorer SIG_PIPE. La seule raison mon serveur a reçu un SIG_PIPE était que j'écrivais vers le client après qu'il avait fermé sa socket.

Allocation Dynamique

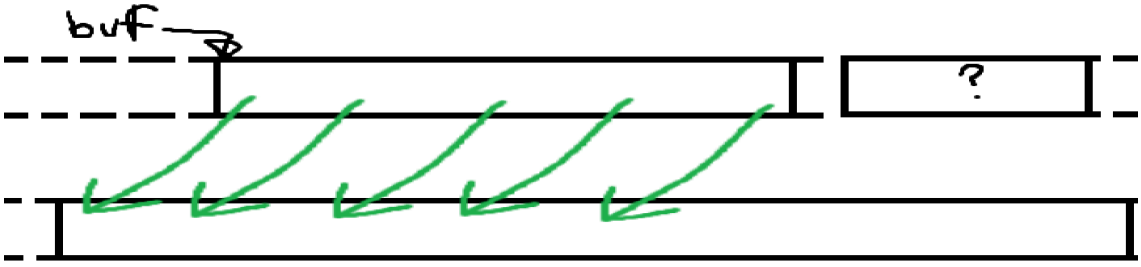
Et finalement le plus grand problème que j'ai rencontré était l'allocation dynamique. Le but d' allouer dynamiquement la mémoire est pour gérer des lignes plus longues qu'une limite statique. Ceci est possible avec l'aide de la fonction realloc, qui va allouer de la mémoire quelque part dans le tas avec les mêmes données et va renvoyer le pointeur à cette place.

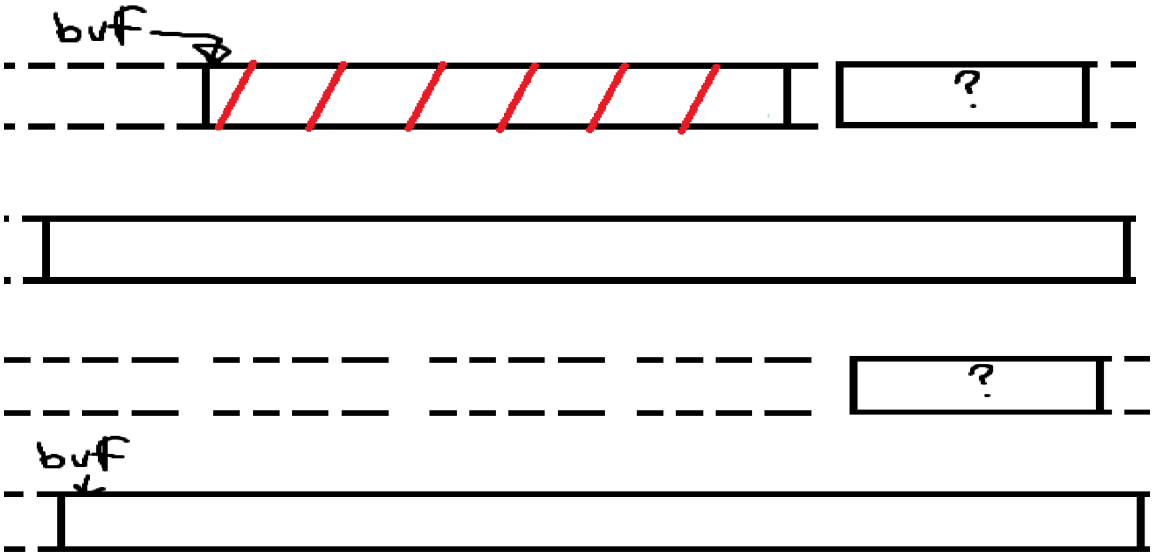
Dans mes recherche j'ai trouvé un vidéo par CodeVault [1], j'ai appris qu'il y avait deux cas:

Quand il y a de l'espace à côté du buf. Qui est très simple.



Et le deuxième quand il n'y a pas assez d'espace à côté du buf. Et dans ce cas, il va trouver de la mémoire quelque part, il va allouer de l'espace puis copier le contenu du buf dedans. Et après il va dealloc la mémoire du buf originale pour finalement renvoyer le pointeur vers le nouveau buf. Et donc on peut faire un appel comme *buf = realloc(*buf, bufsize*2); pour avoir le nouveau pointeur.





Donc j'ai écrit un code pour allouer dynamiquement la place quand on a lu un nombre de caractères approcher à la taille du buf.

```
if(total>allocdynami-200){
                        allocdynami= 2*allocdynami;
                        char* buf= realloc(buf,sizeof(char)*allocdynami);
  }
```

J'ai décidé de faire quelque chose similaire partout dans mon code pour qu'il n'y ait aucun problème à chaque étape du programme, quand on gère des grandes lignes. Mais je n'arrivais pas à faire marcher l'allocation dynamique dans read_until_nl. J'ai trouvé en ligne [2] que si on utilise une fonction pour realloc notre buffer, on va perdre notre pointeur car il va être écrasé. J'ai fait une dernière tentative:

```
int read_until_nl(int fd, char *buf){
   int allocdynamique = 1024;
   int total = 0;
   int n;
   while((n = read(fd, buf+total, 1)) > 0){
        if(buf[total] == '\n')
             break;
        total++;
        while(total>allocdynamique-200){
             allocdynamique = reallocation(&buf,allocdynamique); //approfondissement non fini
        }
   }
   if(n<0)
        return -1;
   return total;
}
```

```
int reallocation(char **buf, int max){
   max= 2*max;
   *buf = realloc(*buf, max);
   return max;
}
```

Pour tester mes l'allocation dynamique j'ai créé à la main un fichier comptine ,fatman.cpt, comme une bombe qui contient plus de 1000 caractères dans sa première ligne. Après plusieurs essais pour faire marcher l'allocation dynamique, je pense que j'étais sur une bonne piste mais à cause de contraintes de temps j'ai décidé d'arrêter de poursuivre une solution à ce problèmes.

Ressources Externes

Youtube:

1. https://youtu.be/2ckt3sP3XvE?si=zwGjZN31yelLIudg
   J'ai regardé cette vidéo pour comprendre comment faire realloc, pour l'allocation dynamique.

2. https://stackoverflow.com/questions/16169939/using-realloc-inside-a-function
   Où j'ai appris qu'on a besoin d'envoyer un double pointeur à la fonction read_until_nl.

3. https://youtu.be/j9yL30R6npk?si=DzkxhjTIFnVCMFaP
   J'ai regardé cette vidéo, car, à mon goût, le TP0 n'était pas suffisant pour avoir une bonne connaissance pour utiliser correctement dirent. Je trouve qu'il n'y avait pas assez d'exemples pour montrer son utilité.

4. https://stackoverflow.com/questions/108183/how-to-prevent-sigpipes-or-handle-them-properly
   Où j'ai appris que mon serveur ferme la connexion à cause de SIG_PIPE.

5. http://sekrit.de/webdocs/c/beginners-guide-away-from-scanf.html
   Pour apprendre à utiliser fgets pour prendre en compte les espace, et arrêter quand le client envoie un retour à la ligne.

Aide par un étudiant:

```
/* Code pour tester comptine_utils.c par Khun Daniphkay
struct catalogue *log;
log = creer_catalogue("comptines");
printf("taille %d\n", log->nb);
struct comptine *cpt;
for (int i = 0; i < log->nb; i++){
   cpt = log->tab[i];
   printf("%s", cpt->titre);
   printf("%s\n", cpt->nom_fichier);
}
 liberer_catalogue(log);*/
```