Programmation des Systèmes

Devoir surveillé final

17 décembre 2014

Durée : 2h30
Documents autorisés

Vous expliquerez votre code de sorte que votre intention (et pas seulement votre implémentation) soit claire. La gestion des erreurs pourra rester très simple, par exemple en interrompant le programme si nécessaire.

Lorsque vos réponses sont de simples modifications de fonctions précédentes, vous vous contenterez d'indiquer les modifications (en particulier en précisant clairement où ces modifications sont placées).

Afin de paralléliser la correction, vous rendrez 3 copies, 1 copie par section.

1 Question de cours et débogage

Répondre aux questions de cette section sur la première copie.

1.1 Question de cours

La commande `ls` écrit sur sa sortie standard la liste des fichiers d'un répertoire (par défaut le répertoire courant). La commande `head` affiche sur sa sortie standard les premières lignes de son entrée standard (par défaut les 10 premières lignes).

Lorsqu'on exécute `ls | head` dans un shell, le tube sert à synchroniser les deux commandes : chacune des deux commandes peut se terminer en premier, l'autre doit alors se terminer peu après.

Q 1. Expliquez ce qui se passe quand `ls` se termine avant `head`.
Expliquez ce qui se passe quand `head` se termine en premier.
Dans quel cas un signal est mis en jeu ? Qu'apporte l'usage d'un signal ?

1.2 Débogage

L'objectif du programme suivant est de faire une copie de son entrée standard sur sa sortie standard.

```c
#include <unistd.h>
#define TAILLE 8

int main(int argc, char *argv[]) {
    char tampon[TAILLE];

    while(read(1, tampon, TAILLE) != -1) {
        write(0, tampon, TAILLE);
    }

    return 0;
}
```

Voici la copie d'une session de test de ce programme, où l'on utilise la notation [1 ↲] pour indiquer ce que l'utilisateur tape (ici, par exemple, les touches « 1 » puis « entrée »), et en texte normal ce que le programme (ou le shell, pour les invites « $ ») affiche. À chaque fois l'utilisateur attend qu'il ne se passe plus rien pour taper une touche.

$ [./copie-std-bug ↲]
[123 ↲]
123
ÿ□ [Ctrl]+[D] 123
ÿ□ [Ctrl]+[D] 123
ÿ□ [Ctrl]+[C]
$ [echo 123 | ./copie-std-bug ↲]
[456 ↲]
(ici rien ne s'affiche, l'utilisateur ne récupère pas la main).

Q 2. Que se passe-t-il quand on tape [Ctrl]+[C] ? Et [Ctrl]+[D] ?

Q 3. Identifiez le ou les bogues de ce programme.

Q 4. Proposez une version correcte.

2 File de messages utilisant un tube

Répondre aux questions de cette section sur la deuxième copie.

L'objectif de cet exercice est d'implémenter une file de messages par un simple tube qui permettra d'échanger des informations entre deux processus, un *producteur* et un *consommateur*.

Nous nous donnons pour cela une structure correspondant à un message. Nous supposerons que le processus producteur dispose d'une fonction générant un message (c'est-à-dire qu'elle remplit une structure avec le contenu du message à envoyer) ; symétriquement, le processus consommateur dispose d'une fonction utilisant un message.

```c
struct message_s { ... } ;

/* Dans le producteur */
int genere(struct message_s *msg);

/* Dans le consommateur */
int utilise(struct message_s *msg);
```

Les fonctions `genere` et `utilise` prennent en paramètre l'adresse du message qu'elles doivent respectivement remplir et utiliser. Elles retournent 0 en cas de réussite et -1 lorsqu'elles ont échoué (par exemple quand il n'y a plus rien à produire).

Nous supposerons tout d'abord que les messages ne prennent que quelques octets.

Q 5. Définissez une fonction `void producteur(int fd)` prenant en argument le descripteur correspondant au tube et effectuant les étapes suivantes en boucle, tant que `genere` retourne 0 :
— il `genere` un message,
— il écrit ce message sur le tube.

Q 6. Définissez une fonction `void consommateur(int fd)` prenant en argument le descripteur correspondant au tube et effectuant les étapes suivantes en boucle, tant que `utilise` retourne 0 :
— il lit un message sur le tube,
— il `utilise` ce message.

Afin de mettre en place le producteur et le consommateur dans une situation simple, nous allons faire une fonction `main` qui crée un processus et déclenche le producteur dans le processus fils et le consommateur dans le processus père.

Q 7. Quand doit-on créer le tube qui permet de connecter le producteur au consommateur ?
Définissez la fonction `main`. Les processus se termineront à la fin respectivement de la fonction `producteur` ou de la fonction `consommateur`.
Vous penserez à fermer les morceaux de tube correctement.

Afin de gérer la terminaison de ces deux processus, nous allons utiliser un mécanisme de signaux : lorsque le consommateur aura reçu tous les messages qu'il attendait, il enverra un signal `SIGTERM` au producteur pour qu'il puisse s'arrêter proprement et au plus tôt.

Q 8. Modifiez vos fonctions précédentes de sorte que :
— le consommateur envoie le signal `SIGTERM` à son processus père, le producteur, quand il se termine,
— le producteur déclenche une fonction `termine_production` quand il reçoit ce signal. Vous préciserez le prototype de `termine_production` qui vous arrange et vous pourrez supposer que cette fonction termine le processus complet.

Prenons un peu de recul sur cette implémentation.

Q 9. Dans quelles limites de taille des messages votre implémentation fonctionnera correctement ? Expliquez.

Q 10. Quelles sont les erreurs de lecture et écriture envisageables dans votre implémentation ? Pouvez-vous indiquer comment vous les géreriez ?

3 Compteur de lignes

Répondre aux questions de cette section sur la troisième copie.

L'objectif de cet exercice est de faire un programme permettant de compter le nombre de lignes apparaissant dans des fichiers, en utilisant plusieurs approches possibles.

Commençons par une fonction qui va compter tous les sauts de ligne (indiqués par le caractère `'\n'`) d'un fichier. Vous utiliserez pour cela les fonctions d'entrée/sortie de bas niveau (c'est-à-dire `open`, `close`, `read`, `write`, etc.).

Q 11. Proposez une fonction `int lignes(char *nom_fichier)` qui :
- ouvre le fichier indiqué,
- lit octet par octet son contenu, en comptabilisant les sauts de lignes,
- ferme le fichier à la fin,
- retourne le nombre de sauts de ligne trouvés.

Q 12. Modifiez la fonction précédente afin de lire le contenu du fichier par blocs de `TAILLE` octets plutôt qu'octet par octet. (Vous indiquerez juste les différences avec la version précédente.)

Q 13. Expliquez pourquoi cette seconde version est préférable.

Nous allons maintenant utiliser des processus légers pour paralléliser ce calcul et recouvrir les entrées-sorties.

Q 14. Proposez une fonction `void *mt_lignes(void *)` qui permette de lancer `lignes` dans un thread. Expliquez en détail quel argument vous passez à cette fonction.

Q 15. Définissez une fonction `main` qui crée un thread par argument et lance `mt_lignes` dans chacun de ces threads. Comment gérez-vous la terminaison des threads ?

Si un fichier est très long et les autres très courts, l'implémentation précédente va devoir attendre la fin de l'exécution pour le long fichier. Une alternative est de découper les fichiers en morceaux et d'utiliser un thread par morceau. Pour simplifier, nous supposerons désormais que nous traitons un seul fichier et nous définissons quelques données partagées :

```
int fd;
int nb_lignes;
int nb_threads_encours;
```

où :

- `fd` contiendra le descripteur du fichier une fois ouvert,
- `nb_lignes` contiendra à la fin le nombre de lignes total du fichier,
- `nb_threads_encours` contiendra à chaque instant le nombre de threads encore en cours de calcul.

Q 16. Que doit-on faire puisque plusieurs threads vont accéder de façon concurrente à ces données ?

Nous découperons le fichier en morceaux de `TAILLE` octets. Le i^e thread va traiter le i^e morceau, c'est-à-dire les octets de `i*TAILLE` à `(i+1)*TAILLE-1`. Comme les accès des différents threads sont concurrents, vous utiliserez l'appel système `pread` plutôt que `read` (cf. le mini-man), afin d'indiquer explicitement à quelle position vous voulez lire des données.

Q 17. Définissez une fonction `void *mt_lignes_morceau(void *)` qui prendra en argument son indice `i` (pour le i^e thread) et qui lira son morceau par blocs.
Avant de se terminer, chaque thread mettra à jour le `nb_lignes` en ajoutant les lignes qu'il aura comptées et décrémentera `nb_threads_encours`.
Le dernier thread à terminer (celui pour lequel `nb_threads_encours` devient nul), affichera le décompte final.

Q 18. Définissez la fonction `main` d'une commande qui prend en premier argument le nom du fichier dont on va calculer le nombre de lignes et :
- récupère la longueur du fichier et en déduit le nombre de morceaux,
- crée un thread par morceau hormis 0 en lançant `mt_lignes_morceau` dans chacun,
- lance `mt_lignes_morceau` pour le morceau 0 dans le thread principal.

Vous expliciterez comment vous gérez la terminaison des threads.