



1 Fonctions utiles

Description extraite du site http://docs.mandraptor.org/files/Operating_systems/Linux/Man_fr/

Fork

fork crée un processus fils qui diffère du processus parent uniquement par ses valeurs PID et PPID et par le fait que toutes les statistiques d'utilisation des ressources sont remises à zéro. Les verrouillages de fichiers, et les signaux en attente ne sont pas hérités. Sous Linux, fork est implémenté en utilisant une méthode de copie à l'écriture. Ceci consiste à ne faire la véritable duplication d'une page mémoire que lorsqu'un processus en modifie une instance. Tant qu'aucun des deux processus n'écrit dans une page donnée, celle-ci n'est pas vraiment dupliquée. Ainsi les seules pénalisations induites par fork sont le temps et la mémoire nécessaires à la copie de la table des pages du parent ainsi que la création d'une structure de tâche pour le fils.

Sleep

sleep() endort le processus jusqu'à ce que nb_sec secondes se soient écoulées, ou jusqu'à ce qu'un signal non-ignoré soit reçu.

Pipe

pipe crée une paire de descripteurs de fichiers, pointant sur un tube, et les place dans un tableau filedes. filedes[0] est utilisé pour la lecture, et filedes[1] pour l'écriture. En général deux processus (créés par fork) vont se partager le tube, et utiliser les fonctions read et write pour se transmettre des données.

Mkfifo

La fonction mkfifo crée un fichier spécial FIFO (tube nommé) à l'emplacement pathname. Mode indique les permissions d'accès. Ces permissions sont modifiées par la valeur d'umask du processus : les permissions d'accès effectivement adoptées sont (mode & ~umask). Un fichier spécial FIFO est semblable à un tube (pipe), sauf qu'il est créé différemment. Plutôt qu'un canal de communication anonyme, un fichier FIFO est inséré dans le système de fichiers en appelant mkfifo. Une fois qu'un fichier FIFO est créé, n'importe quel processus peut l'ouvrir en lecture ou écriture, comme tout fichier ordinaire. En fait, il faut ouvrir les deux extrémités simultanément avant de pouvoir effectuer une opération d'écriture ou de lecture. L'ouverture d'un FIFO en lecture est généralement bloquante, jusqu'à ce qu'un autre processus ouvre le même FIFO en écriture, et inversement.

Open

L'appel-système open() sert à convertir un chemin d'accès en descripteur de fichier (un petit entier non négatif utilisable pour les opérations d'entrées/sorties ultérieures telles read, write, etc.). Lorsque l'appel-système réussit, le descripteur renvoyé sera le plus petit descripteur de fichier non encore ouvert pour le processus. Cet appel crée un nouveau descripteur, non-partagé avec les autres processus. Toutefois le partage de fichiers ouverts peut se produire avec l'appel-système fork (2).

Read

read lit jusqu'à count octets depuis le descripteur de fichier fd dans le buffer pointé par buf. Si count vaut zéro, read renvoie zéro et n'a pas d'autres effets. Si count est supérieur à SSIZE_MAX, le résultat est indéfini.

Write

write écrit jusqu'à count octets dans le fichier associé au descripteur fd depuis le buffer pointé par buf. POSIX réclame qu'une lecture avec read() effectuée après le retour d'une écriture avec write(), renvoie les nouvelles données. Notez que tous les systèmes de fichiers ne sont pas compatibles avec POSIX.

Close

close ferme le descripteur fd, de manière à ce qu'il ne référence plus aucun fichier, et puisse être réutilisé. Tous les verrouillages sur le fichier qui lui était associé, appartenant au processus, sont supprimés (quelque soit le descripteur qui fut utilisé pour obtenir le verrouillage). Si fd est la dernière copie d'un descripteur de fichier donné, les ressources qui lui sont associées sont libérées. Si le descripteur était la dernière référence sur un fichier supprimé avec unlink (2), le fichier est effectivement effacé.

Unlink

unlink détruit un nom dans le système de fichiers. Si ce nom était le dernier lien sur un fichier, et si aucun processus n'a ouvert ce fichier, ce dernier est effacé, et l'espace qu'il utilisait est rendu disponible. Si le nom était le dernier lien sur un fichier, mais qu'un processus conserve encore le fichier ouvert, celui continue d'exister jusqu'à ce que le dernier descripteur le référençant soit fermé. Si le nom correspond à un lien symbolique, le lien est effacé. Si le nom correspond à une socket, une Fifo, ou un périphérique, le nom est effacé mais les processus qui ont ouvert l'objet peuvent continuer à l'utiliser.

2 Création de processus fils

Exercice 2.1.

Ecrire un programme C (proc_fils_v1.c) qui crée deux processus fils. Le premier fils affiche les lettres de A à Z et le deuxième les nombres de 1 à 30. Afin de ralentir la vitesse d'exécution, un délai d'une demie seconde devra être ajouté après, l'affichage d'une lettre ou d'un nombre.

Exercice 2.2.

Modifier le programme précédent (proc_fils_v2.c) pour que l'affichage soit: A B ... Z 1 2 ... 30 dans cet ordre.

3 Serveur d'impression

Exercice 3.1.

a)

Ecrire un programme C (imprimer_v1.c) simulant un serveur d'impression. Celui-ci attend de la part de l'utilisateur des noms de fichier et les envoie à l'imprimante. Bien sûr, l'on n'imprimera pas réellement les fichiers. Au lieu de cela, le programme attendra 3 secondes puis affichera le message: Le fichier nom_du_fichier a été envoyé pour impression.

b)

Tel que imprimer_v1.c est écrit, l'utilisateur doit attendre qu'un fichier soit imprimé pour pouvoir donner le nom d'un autre fichier. Modifier imprimer_v1.c (sauvegarder sous le nom imprimer_v2.c) de sorte

qu'à chaque fois que l'utilisateur a entré un nom de fichier, un processus enfant est créé pour s'occuper de l'impression de ce fichier.

Exercice 3.2.

Modifier le programme `imprimer_v2.c` (sauvegarder sous le nom `imprimer_v3.c`) de l'exercice 3.1. Cette fois-ci, ce programme commence par créer un enfant à qui il envoie, via un tube (pipe), les noms des fichiers à imprimer.

Exercice 3.3.

Le programme réalisé à l'exercice 3.2 présente l'inconvénient que si on lance plusieurs fois l'application en parallèle, il y aura plusieurs enfants en charge des impressions. Pour pallier cet inconvénient, vous devez écrire un programme serveur (`imprimer_serveur.c`) en charge de créer un tube nommé et d'attendre sur ce tube nommé les noms de fichier à imprimer et un programme client (`imprimer_client.c`) capable de se connecter, via le tube nommé, au serveur et de lui envoyer les noms de fichier à imprimer.

Exercice 3.4.

Le programme `imprimer_v2.c` développé dans l'exercice 3.1 présente l'inconvénient que les enfants peuvent tous accéder à l'imprimante en même temps, ce qui risque de mélanger les impressions. Au lieu d'utiliser un enfant dédié à l'impression comme il a été fait dans l'exercice 3.2, modifier `imprimer_v2.c` (sauvegarder sous le nom `imprimer_v4.c`) pour qu'il utilise un sémaphore le but étant qu'un seul enfant puisse imprimer à la fois.

4 Modèle de producteur – consommateur

Exercice 4.1.

Ecrire un programme `producteur_v1.c` permettant de gérer un producteur, un consommateur et un tampon de capacité égale à un message. Le producteur se chargera de placer dans la zone tampon un message. Le consommateur lit et supprime le message de la zone tampon. Un message est prélevé une et une seule fois. Chaque message produit doit être consommé. Un message non utilisé ne peut pas être effacé par un nouveau message.

Exercice 4.2.

Ecrire un programme `producteur_v2.c` permettant de gérer un producteur, un consommateur et un tampon de capacité égale à N messages. Les contraintes sont identiques à l'exercice 4.1, mis à part que la gestion du tampon se fait de manière circulaire : Quand on arrive à la case $N - 1$ on repasse à la case 0 (les opérations de dépôt et de prélèvement se font modulo N)