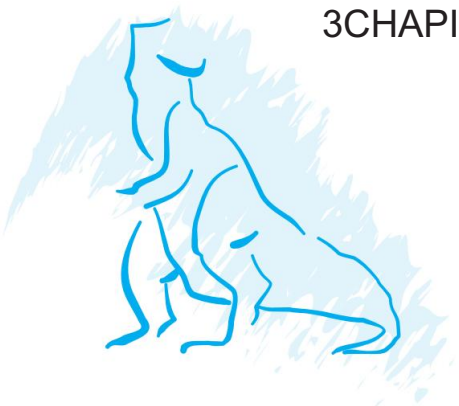
3CHAPITRE

Processus



Les premiers ordinateurs ne permettaient d'exécuter qu'un seul programme à la fois. Ce programme avait le contrôle complet du système et avait accès à toutes les ressources du système. En revanche, les systèmes informatiques contemporains permettent de charger plusieurs programmes en mémoire et de les exécuter simultanément. Cette évolution a nécessité un contrôle plus ferme et un plus grand cloisonnement des différents programmes ; et ces besoins ont abouti à la notion de processus, qui est un programme en exécution. Un processus est l'unité de travail dans un système informatique moderne.

Plus le système d'exploitation est complexe, plus il est censé faire grand chose pour le compte de ses utilisateurs. Bien que sa principale préoccupation soit l'exécution des programmes utilisateur, il doit également s'occuper de diverses tâches système qu'il est préférable d'effectuer dans l'espace utilisateur plutôt que dans le noyau. Un système est donc constitué d'un ensemble de processus, certains exécutant le code utilisateur, d'autres exécutant le code du système d'exploitat Potentiellement, tous ces processus peuvent s'exécuter simultanément, le ou les processeurs étant multiplexés entre eux. Dans ce chapitre, vous découvrirez ce que sont les processus, comment ils sont représentés dans un système d'exploitation et comment ils fonctionnent.

OBJECTIFS DU CHAPITRE

- Identifier les composants distincts d'un processus et illustrer comment ils sont représenté et programmé dans un système d'exploitation.
- Décrire comment les processus sont créés et terminés dans un système d'exploitation, y compris le développement de programmes utilisant les appels système appropriés qui effectuent ces opérations.
- Décrire et contraster la communication interprocessus utilisant la mémoire partagée et la transmission de messages.
- Concevoir des programmes qui utilisent des tubes et de la mémoire partagée POSIX pour effectuer communication interprocessus.
- Décrire la communication client-serveur à l'aide de sockets et de processus distants. appels durs.
- Concevoir des modules de noyau qui interagissent avec le système d'exploitation Linux.

3.1 Concept de processus

Une question qui se pose lors de la discussion sur les systèmes d'exploitation concerne comment appeler toutes les activités du processeur . Les premiers ordinateurs étaient des systèmes par lots qui exécutaient des tâches, suivis par l'émergence de systèmes à temps partagé qui exécutaient des programmes ou des tâches utilisateur . Même sur un système mono-utilisateur, un utilisateur peut être en mesure d'exécuter plusieurs programmes en même temps : un traitement de texte, un navigateur Web et un package de messagerie. Et même si un ordinateur ne peut exécuter qu'un seul programme à la fois, par exemple sur un périphérique intégré qui ne prend pas en charge le multitâche, le système d'exploitation devra peut-être prendre en charge ses propres activités programmées internes, telles que la gestion de la mémoire.
À bien des égards, toutes ces activités sont similaires, c'est pourquoi nous les appelons toutes des processus.

Bien que nous préférions personnellement le terme plus contemporain de processus, le terme travail a une signification historique, car une grande partie de la théorie et de la terminologie des systèmes d'exploitation a été développée à une époque où l'activité principale des systèmes d'exploitation était le traitement des tâches. Par conséquent, dans certains cas appropriés, nous utilisons job pour décrire le rôle du système d'exploitation. À titre d'exemple, il serait trompeur d'éviter l'utilisation de termes communément acceptés incluant le mot travail (tels que planification des tâches) simplement parce que le processus a supplanté le travail.

3.1.1 Le processus

De manière informelle, comme mentionné précédemment, un processus est un programme en exécution. L'état de l'activité en cours d'un processus est représenté par la valeur du compteur de programme et le contenu des registres du processeur. La disposition de la mémoire d'un processus est généralement divisée en plusieurs sections et est illustrée à la figure 3.1.
Ces sections comprennent :

- Section de texte : le code exécutable
- Section Données : variables globales

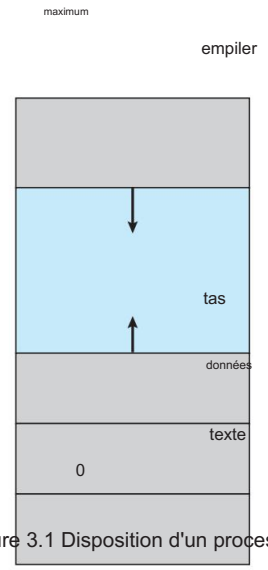


Figure 3.1 Disposition d'un processus en mémoire.

- Section de tas : mémoire allouée dynamiquement pendant l'exécution du programme.
  temps
- Section pile : stockage de données temporaire lors de l'appel de fonctions (telles que les paramètres de fonction, les adresses de retour et les variables locales)

Notez que les tailles des sections de texte et de données sont fixes, car leurs tailles ne changent pas pendant l'exécution du programme. Cependant, les sections de pile et de tas peuvent diminuer et croître de manière dynamique pendant l'exécution du programme. Chaque fois qu'une fonction est appelée, un enregistrement d'activation contenant les paramètres de la fonction, les variables locales et l'adresse de retour est placé sur la pile ; lorsque le contrôle est renvoyé par la fonction, l'enregistrement d'activation est extrait de la pile. De même, le tas augmentera à mesure que la mémoire est allouée dynamiquement et diminuera lorsque la mémoire sera renvoyée au système. Même si les sections de pile et de tas se rapprochent les unes des autres, le système d'exploitation doit s'assurer qu'elles ne se chevauchent pas .

Nous soulignons qu'un programme en soi n'est pas un processus. Un programme est une entité passive , telle qu'un fichier contenant une liste d'instructions stockées sur disque (souvent appelé fichier exécutable). En revanche, un processus est une entité active , avec un compteur de programme spécifiant la prochaine instruction à exécuter et un ensemble de ressources associées. Un programme devient un processus lorsqu'un fichier exécutable est chargé en mémoire. Deux techniques courantes pour charger des fichiers exécutables consistent à double-cliquer sur une icône représentant le fichier exécutable et à saisir le nom du fichier exécutable sur la ligne de commande (comme dans prog.exe ou a.out).

Bien que deux processus puissent être associés à un même programme, ils sont néanmoins considérés comme deux séquences d'exécution distinctes. Par exemple, plusieurs utilisateurs peuvent exécuter différentes copies du programme de messagerie, ou le même utilisateur peut appeler plusieurs copies du programme de navigateur Web. Chacun de ces éléments est un processus distinct ; et bien que les sections de texte soient équivalentes, les sections de données, de tas et de pile varient. Il est également courant qu'un processus génère de nombreux processus au cours de son exécution. Nous discutons de ces questions dans la section 3.4.

Notez qu'un processus peut lui-même être un environnement d'exécution pour un autre code. L'environnement de programmation Java en fournit un bon exemple. Dans la plupart des cas, un programme Java exécutable est exécuté dans la machine virtuelle Java (JVM). La JVM s'exécute comme un processus qui interprète le code Java chargé et prend des actions (via des instructions machine natives) au nom de ce code.

Par exemple, pour exécuter le programme Java compilé Program.class, nous aurions entrer

Programme Java

La commande java exécute la JVM comme un processus ordinaire, qui à son tour exécute le programme Java Program dans la machine virtuelle. Le concept est le même que celui de la simulation, sauf que le code, au lieu d'être écrit pour un jeu d'instructions différent, est écrit en langage Java.

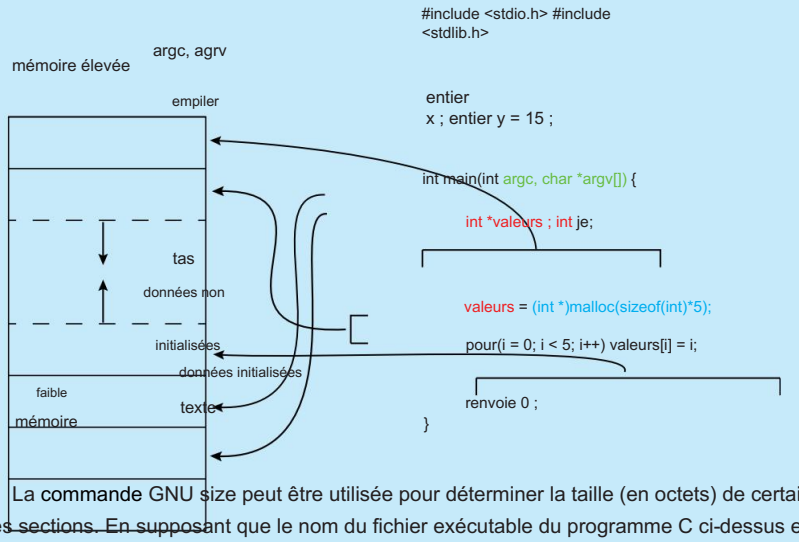
3.1.2 État du processus

Au fur et à mesure qu'un processus s'exécute, il change d'état. L'état d'un processus est défini en partie par l'activité actuelle de ce processus. Un processus peut être dans l'un des cas suivants États:

DISPOSITION DE LA MÉMOIRE DU PROGRAMME AC

La figure ci-dessous illustre la disposition d'un programme C en mémoire, mettant en évidence la manière dont les différentes sections d'un processus sont liées à un programme C réel. Cette figure est similaire au concept général d'un processus en mémoire tel que présenté dans la figure 3.1, avec quelques différences :

- La section des données globales est divisée en différentes sections pour (a) les données initialisées et (b) les données non initialisées.
- Une section distincte est fournie pour les paramètres argc et argv transmis à la fonction main() .



La commande GNU size peut être utilisée pour déterminer la taille (en octets) de certaines de ces sections. En supposant que le nom du fichier exécutable du programme C ci-dessus est mémoire, voici la sortie générée en entrant la taille de mémoire de la commande :

données texte bss		déc	hexadécimal	nom de fichier
1158 284	8	1450	5aa	mémoire

Le champ de données fait référence aux données non initialisées et bss fait référence aux données initialisées. (bss est un terme historique faisant référence au bloc commencé par un symbole.) Les valeurs dec et hex sont la somme des trois sections représentées respectivement en décimal et en hexadécimal.

- Nouveau. Le processus est en train de se créer.
- En cours d'exécution. Les instructions sont en cours d'exécution.
- En attendant. Le processus attend qu'un événement se produise (comme l'achèvement d'une E/S ou la réception d'un signal).
- Prêt. Le processus attend d'être affecté à un processeur.

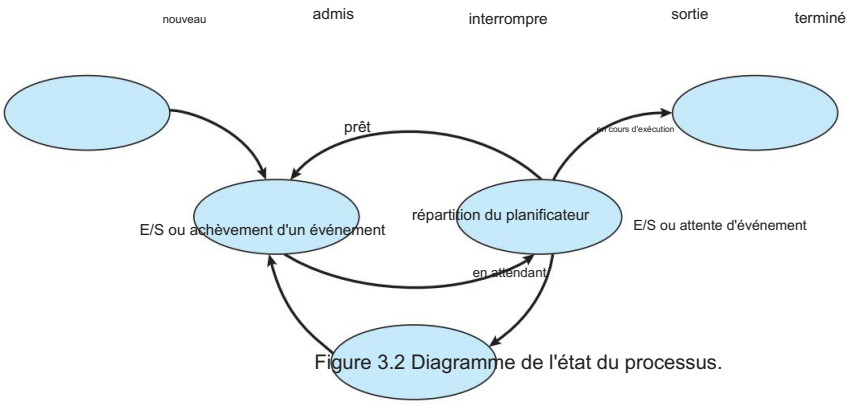


Figure 3.2 Diagramme de l'état du processus.

- Terminé. Le processus est terminé.

Ces noms sont arbitraires et varient selon les systèmes d'exploitation. Les états qu'ils représentent se retrouvent cependant sur tous les systèmes. Certains systèmes d'exploitation délimitent également plus finement les états des processus. Il est important de réaliser qu'un seul processus peut être exécuté sur un cœur de processeur à tout moment. De nombreux processus peuvent cependant être prêts et en attente . Le diagramme d'état correspondant à ces états est présenté sur la figure 3.2.

3.1.3 Bloc de contrôle du processus

Chaque processus est représenté dans le système d'exploitation par un bloc de contrôle de processus (PCB), également appelé bloc de contrôle de tâches. Un PCB est illustré à la figure 3.3. Il contient de nombreuses informations associées à un processus spécifique, notamment celles-ci :

- État du processus. L'État peut être nouveau, prêt, en marche, en attente, arrêté et bientôt.
- Compteur de programme. Le compteur indique l'adresse de la prochaine instruction à exécuter pour ce processus.

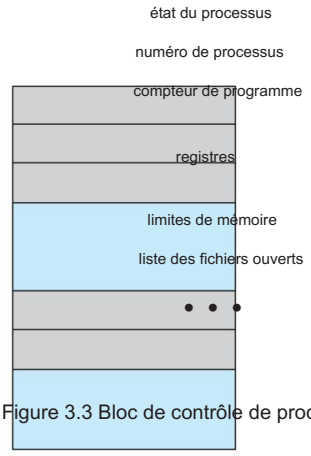


Figure 3.3 Bloc de contrôle de processus (PCB).

- Registres du processeur . Les registres varient en nombre et en type, en fonction de l'architecture informatique. Ils comprennent des accumulateurs, des registres d'index, des pointeurs de pile et des registres à usage général, ainsi que toute information de code de condition. Avec le compteur de programme, ces informations d'état doivent être enregistrées lorsqu'une interruption se produit, pour permettre au processus de se poursuivre correctement par la suite lorsqu'il est reprogrammé pour s'exécuter.
- Informations de planification du processeur. Ces informations incluent une priorité de processus, des pointeurs vers des files d'attente de planification et tout autre paramètre de planification. (Le chapitre 5 décrit la planification des processus.)
- Informations sur la gestion de la mémoire. Ces informations peuvent inclure des éléments tels que la valeur des registres de base et limite et les tables de pages, ou les tables de segments, selon le système de mémoire utilisé par le système d'exploitation (Chapitre 9).
- Information comptable. Ces informations incluent la quantité de CPU et de temps réel utilisée, les limites de temps, les numéros de compte, les numéros de travail ou de processus, etc.
- Informations sur l'état des E/S . Ces informations incluent la liste des périphériques d'E/S alloués au processus, une liste de fichiers ouverts, etc.

En bref, le PCB sert simplement de référentiel pour toutes les données nécessaires au démarrage ou au redémarrage d'un processus, ainsi que certaines données comptables.

3.1.4 Fils de discussion

Le modèle de processus discuté jusqu'à présent impliquait qu'un processus est un programme qui exécute un seul thread d'exécution. Par exemple, lorsqu'un processus exécute un programme de traitement de texte, un seul fil d'instructions est exécuté.
Ce fil de contrôle unique permet au processus d'effectuer une seule tâche à la fois. Ainsi, l'utilisateur ne peut pas simultanément saisir des caractères et exécuter le correcteur orthographique. La plupart des systèmes d'exploitation modernes ont étendu le concept de processus pour permettre à un processus d'avoir plusieurs threads d'exécution et ainsi d'effectuer plus d'une tâche à la fois. Cette fonctionnalité est particulièrement utile sur les systèmes multicœurs, où plusieurs threads peuvent s'exécuter en parallèle. Un traitement de texte multithread pourrait, par exemple, attribuer à un thread la gestion des entrées utilisateur tandis qu'un autre thread exécute le correcteur orthographique. Sur les systèmes prenant en charge les threads, le PCB est étendu pour inclure des informations sur chaque thread. D'autres changements dans l'ensemble du système sont également nécessaires pour prendre en charge les threads. Le chapitre 4 explore les discussions en détail.

3.2 Planification des processus

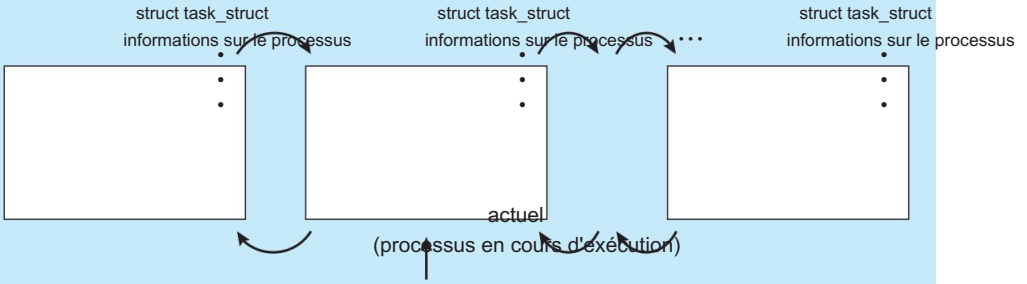
L'objectif de la multiprogrammation est d'avoir un processus en cours d'exécution à tout moment afin de maximiser l'utilisation du processeur . L'objectif du partage du temps est de basculer si fréquemment un cœur de processeur entre les processus que les utilisateurs peuvent interagir avec chaque programme pendant son exécution. Pour atteindre ces objectifs, l' ordonnanceur de processus sélectionne un processus disponible (éventuellement parmi un ensemble de plusieurs processus disponibles) pour l'exécution d'un programme sur un cœur. Chaque cœur de processeur peut exécuter un proc

REPRÉSENTATION DE PROCESSUS SOUS LINUX

Le bloc de contrôle de processus dans le système d'exploitation Linux est représenté par la structure de tâche de structure C, qui se trouve dans le fichier d'inclusion <include/linux/sched.h> dans le répertoire du code source du noyau. Cette structure contient toutes les informations nécessaires pour représenter un processus, y compris l'état du processus, les informations de planification et de gestion de la mémoire, la liste des fichiers ouverts, ainsi que des pointeurs vers le parent du processus et une liste de ses enfants et frères et sœurs. (Le parent d'un processus est le processus qui l'a créé ; ses enfants sont tous les processus qu'il crée.
Ses frères et sœurs sont des enfants avec le même processus parent.) Certains de ces champs incluent :

```
informations de /* état du processus */ long state; /*
parent de ce processus */ struct schedentity se; struct tâche struct *parent; /* le
list head children; /* les enfants de ce processus */ struct files struct *files; /* liste
des fichiers ouverts */ /* espace d'adressage */ struct mm struct *mm;
```

Par exemple, l'état d'un processus est représenté par le champ état long dans cette structure. Dans le noyau Linux, tous les processus actifs sont représentés à l'aide d'une liste doublement chaînée de structures de tâches. Le noyau maintient un pointeur – actuel – vers le processus en cours d'exécution sur le système, comme indiqué ci-dessous :



Pour illustrer la façon dont le noyau pourrait manipuler l'un des champs de la structure de tâche pour un processus spécifié, supposons que le système souhaite changer l'état du processus en cours d'exécution vers la valeur new state.
Si current est un pointeur vers le processus en cours d'exécution, son état est modifié comme suit :

```
état actuel-> = nouvel état ;
```

Pour un système doté d'un seul cœur de processeur , il n'y aura jamais plus d'un processus en cours d'exécution à la fois, alors qu'un système multicœur peut exécuter plusieurs processus en même temps. S'il y a plus de processus que de cœurs, les processus excédentaires auront

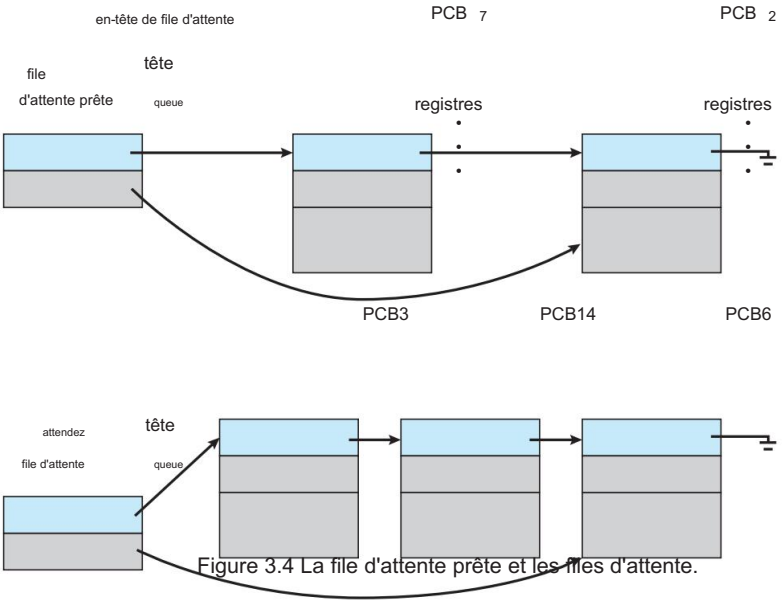


Figure 3.4 La file d'attente prête et les files d'attente.

attendre qu'un noyau soit libre et puisse être reprogrammé. Le nombre de processus actuellement en mémoire est appelé degré de multiprogrammation.

Équilibrer les objectifs de multiprogrammation et de partage de temps nécessite également de prendre en compte le comportement général d'un processus. En général, la plupart des processus peuvent être décrits comme étant liés aux E/S ou au CPU . Un processus lié aux E/S est un processus qui passe plus de temps à effectuer des E/S qu'à effectuer des calculs. En revanche, un processus lié au processeur génère rarement des requêtes d'E/S , consacrant plus de temps à effectuer des calculs.

3.2.1 Planification des files d'attente Lorsque

les processus entrent dans le système, ils sont placés dans une file d'attente prête, où ils sont prêts et en attente d'exécution sur le cœur d'un processeur . Cette file d'attente est généralement stockée sous forme de liste chaînée ; un en-tête de file d'attente prête contient des pointeurs vers le premier PCB de la liste, et chaque PCB comprend un champ de pointeur qui pointe vers le PCB suivant dans la file d'attente prête file d'attente.

Le système comprend également d'autres files d'attente. Lorsqu'un processus se voit attribuer un cœur de processeur , il s'exécute pendant un certain temps et finit par se terminer, est interrompu ou attend l'occurrence d'un événement particulier, tel que l'achèvement d'une requête d'E/S . Supposons que le processus envoie une requête d'E/S à un périphérique tel qu'un disque.
Étant donné que les appareils fonctionnent beaucoup plus lentement que les processeurs, le processus devra attendre que les E/S soient disponibles. Les processus qui attendent qu'un certain événement se produise, comme la fin d' une E/S , sont placés dans une file d'attente (Figure 3.4).

Une représentation courante de la planification des processus est un diagramme de file d'attente, tel que celui de la figure 3.5. Deux types de files d'attente sont présents : la file d'attente prête et un ensemble de files d'attente. Les cercles représentent les ressources qui servent les files d'attente et les flèches indiquent le flux des processus dans le système.

Un nouveau processus est initialement placé dans la file d'attente prête. Il y attend jusqu'à ce qu'il soit sélectionné pour exécution ou expédié. Une fois que le processus se voit attribuer un cœur de processeur et est en cours d'exécution, l'un des événements suivants peut se produire :

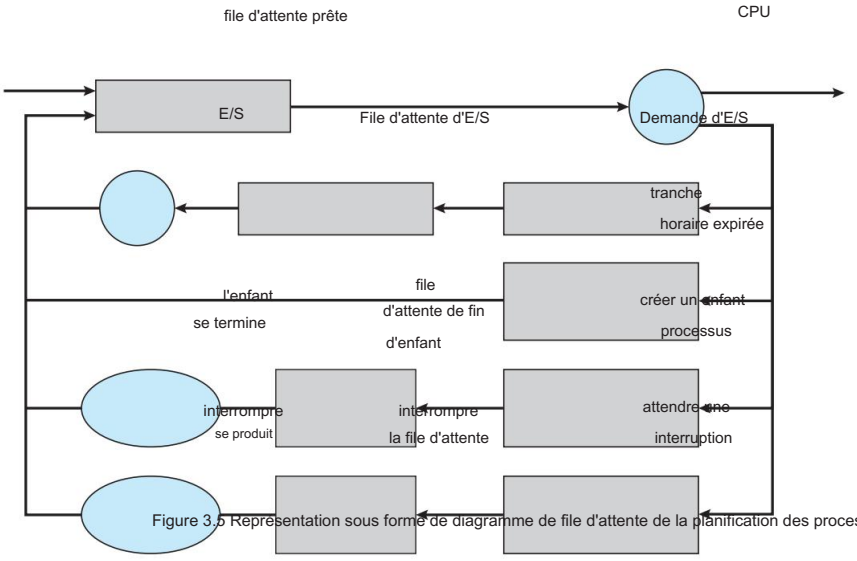


Figure 3.5 Représentation sous forme de diagramme de file d'attente de la planification des processus.

- Le processus peut émettre une requête d'E/S puis être placé en attente d'E/S.

  file d'attente.

- Le processus pourrait créer un nouveau processus enfant, puis être placé dans une file d'attente en attendant la fin du processus enfant.
- Le processus pourrait être supprimé de force du noyau, à la suite d'une interruption ou de l'expiration de sa tranche de temps, et être remis dans la file d'attente prête.

Dans les deux premiers cas, le processus passe finalement de l'état d'attente à l'état prêt puis est remis dans la file d'attente prêt. Un processus continue ce cycle jusqu'à ce qu'il se termine, moment auquel il est supprimé de toutes les files d'attente et son PCB et ses ressources sont désalloués.

3.2.2 Planification du processeur Un

processus migre entre la file d'attente prête et diverses files d'attente tout au long de sa durée de vie. Le rôle du planificateur de CPU est de sélectionner parmi les processus qui se trouvent dans la file d'attente prête et d'attribuer un cœur de CPU à l'un d'entre eux. Le planificateur du processeur doit sélectionner fréquemment un nouveau processus pour le processeur . Un processus lié aux E/S peut s'exécuter pendant quelques millisecondes seulement avant d'attendre une demande d'E/S . Même si un processus lié au processeur nécessitera un cœur de processeur pendant des durées plus longues, il est peu probable que le planificateur accorde le cœur à un processus pendant une période prolongée. Au lieu de cela, il est probablement conçu pour supprimer de force le processeur d'un processus et planifier l'exécution d'un autre processus. Par conséquent, le planificateur du processeur s'exécute au moins une fois toutes les 100 millisecondes, bien que généralement beaucoup p

Certains systèmes d'exploitation disposent d'une forme intermédiaire de planification, appelée swapping, dont l'idée clé est qu'il peut parfois être avantageux de supprimer un processus de la mémoire (et des conflits actifs pour le processeur) et ainsi de réduire le degré de multiprogrammation. Plus tard, le processus peut être réintroduit en mémoire et son exécution peut reprendre là où elle s'était arrêtée.
Ce schéma est connu sous le nom d'échange car un processus peut être « échangé »

de la mémoire au disque, où son état actuel est enregistré, puis « échangé » du disque vers la mémoire, où son état est restauré. L'échange n'est généralement nécessaire que lorsque la mémoire est surchargée et doit être libérée.
L'échange est abordé au chapitre 9.

3.2.3 Changement de contexte

Comme mentionné dans la section 1.2.1, les interruptions amènent le système d'exploitation à modifier un cœur de processeur de sa tâche actuelle et à exécuter une routine du noyau. De telles opérations se produisent fréquemment sur des systèmes à usage général. Lorsqu'une interruption se produit, le système doit enregistrer le contexte actuel du processus exécuté sur le cœur du processeur afin de pouvoir restaurer ce contexte une fois son traitement terminé, essentiellement en suspendant le processus puis en le reprenant. Le contexte est représenté dans le PCB du processus. Il inclut la valeur des registres du processeur , l'état du processus (voir Figure 3.2) et les informations de gestion de la mémoire. De manière générique, nous effectuons une sauvegarde de l'état actuel du cœur du CPU , que ce soit en mode noyau ou utilisateur, puis une restauration d'état pour reprendre les opérations.

Le basculement du cœur du processeur vers un autre processus nécessite d'effectuer une sauvegarde de l'état du processus en cours et une restauration de l'état d'un processus différent. Cette tâche est connue sous le nom de changement de contexte et est illustrée dans la figure 3.6. Lorsqu'un changement de contexte se produit, le noyau enregistre le contexte de l'ancien processus dans son PCB et charge le contexte enregistré du nouveau processus dont l'exécution est programmée. Le temps de changement de contexte est une pure surcharge, car le système n'effectue aucun travail utile lors du changement. La vitesse de commutation varie d'une machine à l'autre,

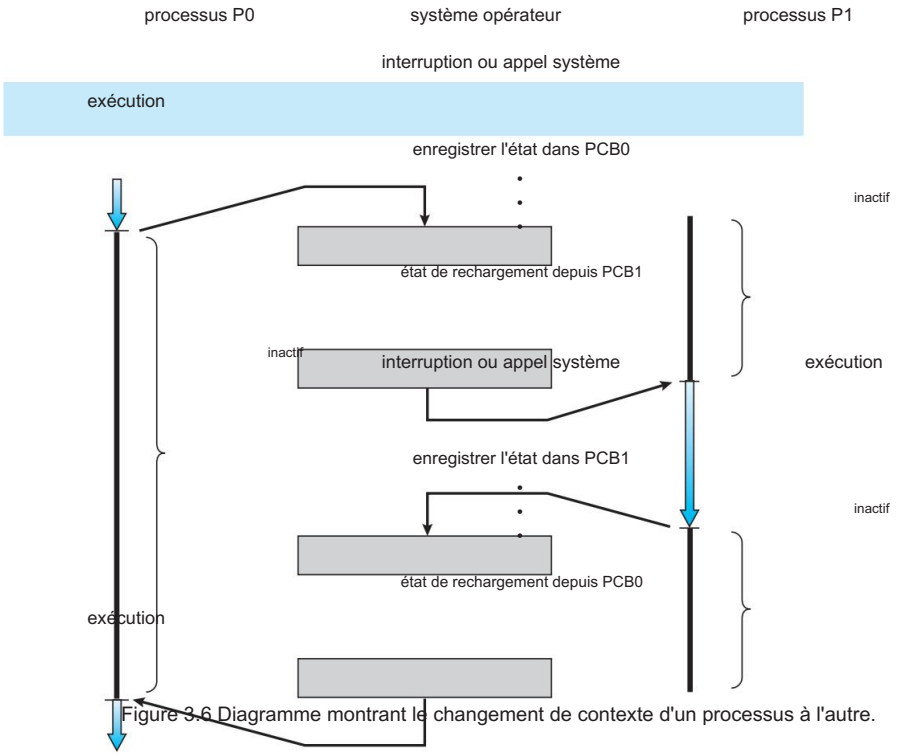


Figure 3.6 Diagramme montrant le changement de contexte d'un processus à l'autre.

MULTITÂCHES DANS LES SYSTÈMES MOBILES

En raison des contraintes imposées aux appareils mobiles, les premières versions d' iOS ne permettaient pas le multitâche utilisateur-application ; une seule application s'exécutait au premier plan tandis que toutes les autres applications utilisateur étaient suspendues. Les tâches du système d'exploitation étaient multitâches car elles étaient écrites par Apple et se comportaient bien. Cependant, à partir d' iOS 4, Apple a fourni une forme limitée de multitâche pour les applications utilisateur, permettant ainsi à une seule application de premier plan de s'exécuter simultanément avec plusieurs applications d'arrière-plan. (Sur un appareil mobile, l' application de premier plan est l'application actuellement ouverte et apparaissant à l'écran. L' application d'arrière -plan reste en mémoire, mais n'occupe pas l'écran d'affichage.) L' API de programmation iOS 4 prenait en charge le multitâche, permettant ainsi un processus à exécuter en arrière-plan sans être suspendu. Cependant, il était limité et disponible uniquement pour quelques types d'applications. Alors que le matériel pour appareils mobiles a commencé à offrir des capacités de mémoire plus importantes, plusieurs cœurs de traitement et une plus grande autonomie de batterie, les versions ultérieures d' iOS ont commencé à prendre en charge des fonctionnalités plus riches pour le multitâche avec moins de restrictions. Par exemple, l'écran plus grand des tablettes iPad permettait d'exécuter deux applications de premier plan en même temps, une technique connue sous le nom d'écran partagé.

Depuis ses origines, Android prend en charge le multitâche et n'impose aucune contrainte sur les types d'applications pouvant s'exécuter en arrière-plan. Si une application nécessite un traitement en arrière-plan, elle doit utiliser un service, un composant d'application distinct qui s'exécute pour le compte du processus en arrière-plan. Prenons l'exemple d'une application de streaming audio : si l'application passe en arrière-plan, le service continue d'envoyer des données audio au pilote du périphérique audio au nom de l'application en arrière-plan. En fait, le service continuera à fonctionner même si l'application en arrière-plan est suspendue. Les services n'ont pas d'interface utilisateur et ont une faible empreinte mémoire, offrant ainsi une technique efficace pour effectuer plusieurs tâches dans un environnement mobile.

la vitesse de la mémoire, le nombre de registres qui doivent être copiés et l'existence d'instructions spéciales (telles qu'une seule instruction pour charger ou stocker tous les registres). Une vitesse typique est de plusieurs microsecondes.

Les temps de changement de contexte dépendent fortement de la prise en charge matérielle. Par exemple, certains processeurs fournissent plusieurs ensembles de registres. Un changement de contexte nécessite ici simplement de changer le pointeur vers l'ensemble de registres actuel. Bien entendu, s'il y a plus de processus actifs qu'il n'y a d'ensembles de registres, le système a recours à la copie des données de registre vers et depuis la mémoire, comme auparavant. De plus, plus le système d'exploitation est complexe, plus la quantité de travail à effectuer lors d'un changement de contexte est importante. Comme nous le verrons au chapitre 9, les techniques avancées de gestion de la mémoire peuvent nécessiter que des données supplémentaires soient commutées avec chaque contexte. Par exemple, l'espace d'adressage du processus en cours doit être préservé pendant que l'espace de la tâche suivante est préparé pour être utilisé. La manière dont l'espace d'adressage est préservé et la quantité de travail nécessaire pour le préserver dépendent de la méthode de gestion de la mémoire du système d'exploitation.

3.3 Opérations sur les processus

Les processus de la plupart des systèmes peuvent s'exécuter simultanément et peuvent être créés et supprimés dynamiquement. Ainsi, ces systèmes doivent fournir un mécanisme de création et de terminaison des processus. Dans cette section, nous explorons les mécanismes impliqués dans la création de processus et illustrons la création de processus sur les systèmes UNIX et Windows.

3.3.1 Création de processus

Au cours de son exécution, un processus peut créer plusieurs nouveaux processus. Comme mentionné précédemment, le processus de création est appelé processus parent et les nouveaux processus sont appelés enfants de ce processus. Chacun de ces nouveaux processus peut à son tour créer d'autres processus, formant ainsi un arbre de processus.

La plupart des systèmes d'exploitation (y compris UNIX, Linux et Windows) identifient les processus selon un identifiant de processus unique (ou pid), qui est généralement un nombre entier. Le pid fournit une valeur unique pour chaque processus du système et peut être utilisé comme index pour accéder à divers attributs d'un processus au sein du noyau.

La figure 3.7 illustre une arborescence de processus typique pour le système d'exploitation Linux, montrant le nom de chaque processus et son pid. (Nous utilisons le terme processus de manière assez vague dans cette situation, car Linux préfère plutôt le terme tâche .) Le processus systemd (qui a toujours un pid de 1) sert de processus parent racine pour tous les processus utilisateur et est le premier processus utilisateur. créé au démarrage du système. Une fois le système démarré, le processus systemd crée des processus qui fournissent des services supplémentaires tels qu'un serveur Web ou d'impression, un serveur ssh , etc. Dans la figure 3.7, nous voyons deux enfants de systemd : login et sshd.
Le processus logind est responsable de la gestion des clients qui se connectent directement au système. Dans cet exemple, un client s'est connecté et utilise le shell bash , auquel le pid 8416 a été attribué. À l'aide de l' interface de ligne de commande bash , cet utilisateur a créé le processus ps ainsi que l' éditeur vim . Le processus sshd est chargé de gérer les clients qui se connectent au système à l'aide de ssh (qui est l'abréviation de Secure Shell).

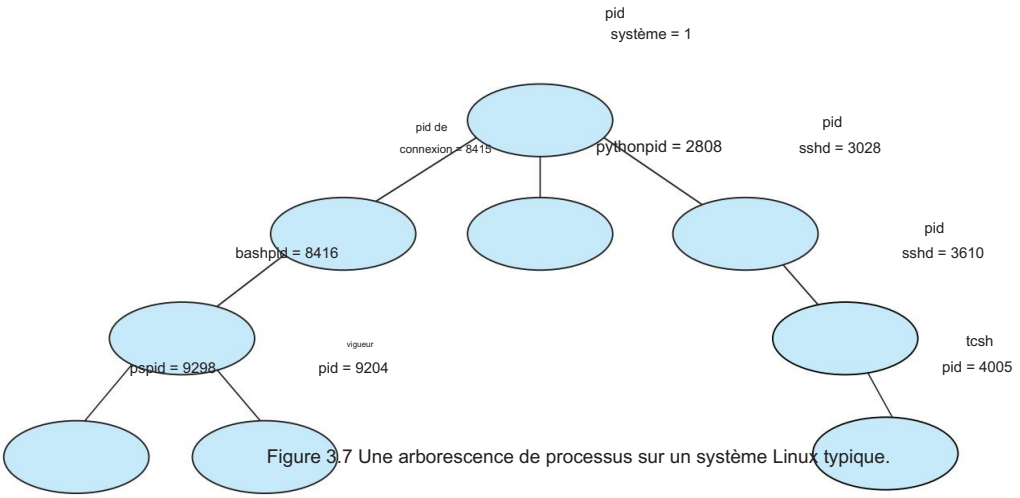


Figure 3.7 Une arborescence de processus sur un système Linux typique.

LES PROCESSUS init ET systemd

Les systèmes UNIX traditionnels identifient le processus init comme racine de tous les processus enfants. init (également connu sous le nom de System V init) se voit attribuer un pid de 1 et est le premier processus créé au démarrage du système. Sur une arborescence de processus similaire à celle illustrée dans la figure 3.7, init se trouve à la racine.

Les systèmes Linux ont initialement adopté l'approche d'initialisation System V , mais les distributions récentes l'ont remplacée par systemd. Comme décrit dans la section 3.3.1, systemd sert de processus initial du système, de la même manière que System V init ; cependant, il est beaucoup plus flexible et peut fournir plus de services que init.

Sur les systèmes UNIX et Linux, nous pouvons obtenir une liste de processus en utilisant la commande ps . Par exemple, la commande

```
ps -el
```

listera les informations complètes sur tous les processus actuellement actifs dans le système. Un arbre de processus similaire à celui illustré à la figure 3.7 peut être construit en traçant de manière récursive les processus parents jusqu'au processus systemd . (De plus, les systèmes Linux fournissent la commande pstree , qui affiche une arborescence de tous les processus du système.)

En général, lorsqu'un processus crée un processus enfant, ce processus enfant aura besoin de certaines ressources (temps CPU , mémoire, fichiers, périphériques d'E/S) pour accomplir sa tâche. Un processus enfant peut être capable d'obtenir ses ressources directement à partir du système d'exploitation, ou il peut être limité à un sous-ensemble des ressources du processus parent. Le parent peut devoir partitionner ses ressources entre ses enfants, ou il peut partager certaines ressources (telles que la mémoire ou les fichiers) entre plusieurs de ses enfants. Restreindre un processus enfant à un sous-ensemble des ressources du parent empêche tout processus de surcharger le système en créant trop de processus enfants.

En plus de fournir diverses ressources physiques et logiques, le processus parent peut transmettre des données d'initialisation (entrée) au processus enfant. Par exemple, considérons un processus dont la fonction est d'afficher le contenu d'un fichier, par exemple hw1.c, sur l'écran d'un terminal. Lorsque le processus est créé, il obtiendra, en entrée de son processus parent, le nom du fichier hw1.c. En utilisant ce nom de fichier, il ouvrira le fichier et en écrira le contenu. Il peut également obtenir le nom du périphérique de sortie. Alternativement, certains systèmes d'exploitation transmettent des ressources aux processus enfants. Sur un tel système, le nouveau processus peut obtenir deux fichiers ouverts, hw1.c et le périphérique terminal, et peut simplement transférer les données entre les deux.

Lorsqu'un processus crée un nouveau processus, deux possibilités d'exécution existent :

1. Le parent continue de s'exécuter en même temps que ses enfants.
2. Le parent attend que certains ou tous ses enfants aient pris fin.

Il existe également deux possibilités d'espace d'adressage pour le nouveau processus :

1. Le processus enfant est un double du processus parent (il a le même programme et les mêmes données que le parent).
2. Le processus enfant contient un nouveau programme chargé.

Pour illustrer ces différences, considérons d'abord le système d'exploitation UNIX . Sous UNIX, comme nous l'avons vu, chaque processus est identifié par son identifiant de processus, qui est un entier unique. Un nouveau processus est créé par l' appel système fork() . Le nouveau processus consiste en une copie de l'espace d'adressage du processus d'orig Ce mécanisme permet au processus parent de communiquer facilement avec son processus enfant. Les deux processus (le parent et l'enfant) continuent l'exécution à l'instruction après fork (), avec une différence : le code de retour de fork() est zéro pour le nouveau processus (enfant), tandis que l'identifiant de processus (différent de zéro) de l'enfant est rendu au parent.

Après un appel système fork() , l'un des deux processus utilise généralement l' appel système exec() pour remplacer l'espace mémoire du processus par un nouveau programme. L' appel système exec() charge un fichier binaire en mémoire (en détruisant l'image mémoire du programme contenant l' appel système exec()) et démarre

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()

{ pid t pid ;

    /* fork un processus enfant */ pid =
    fork();

    if (pid < 0) { /* une erreur s'est produite */
        fprintf(stderr, "Fork Failed"); renvoyer 1 ;


    } else if (pid == 0) { /* processus enfant */ execlp("/bin/
        ls","ls",NULL);

    } else { /* processus parent */ /* le
        parent attendra que l'enfant termine */ wait(NULL); printf("Enfant
        terminé");

    }

    renvoie 0 ;
}
```

Figure 3.8 Création d'un processus distinct à l'aide de l'appel système UNIX fork() .

son exécution. De cette manière, les deux processus sont capables de communiquer puis de se séparer. Le parent peut alors créer plus d'enfants ; ou, s'il n'a rien d'autre à faire pendant que l'enfant s'exécute, il peut émettre un appel système wait() pour se déplacer hors de la file d'attente prête jusqu'à la fin de l'enfant. Étant donné que l'appel à exec() recouvre l'espace d'adressage du processus avec un nouveau programme, exec() ne renvoie pas le contrôle sauf si une erreur se produit.

Le programme C illustré à la figure 3.8 illustre les appels système UNIX décrits précédemment. Nous avons maintenant deux processus différents exécutant des copies du même programme. La seule différence est que la valeur de la variable pid pour le processus enfant est nulle, tandis que celle pour le parent est une valeur entière supérieure à zéro (en fait, c'est le pid réel du processus enfant). Le processus enfant hérite des privilèges et des attributs de planification du parent, ainsi que de certaines ressources, telles que les fichiers ouverts. Le processus enfant superpose ensuite son espace d'adressage avec la commande UNIX /bin/ls (utilisée pour obtenir une liste de répertoires) à l'aide de l' appel système execlp() (execlp() est une version de l' appel système exec()). Le parent attend que le processus enfant se termine avec l' appel système wait() . Lorsque le processus enfant se termine (en appelant implicitement ou explicitement exit()), le processus parent reprend l'appel à wait(), où il se termine à l'aide de l' appel système exit() . Ceci est également illustré dans la figure 3.9.

Bien sûr, rien n'empêche l'enfant de ne pas invoquer exec() et de continuer à s'exécuter en tant que copie du processus parent. Dans ce scénario, le parent et l'enfant sont des processus simultanés exécutant les mêmes instructions de code. L'enfant étant une copie du parent, chaque processus possède sa propre copie de toutes les données.

À titre d'exemple alternatif, nous considérons ensuite la création de processus sous Windows. Les processus sont créés dans l' API Windows à l'aide de la fonction CreateProcess() , qui est similaire à fork() dans la mesure où un parent crée un nouveau processus enfant.
Cependant, alors que fork() fait en sorte que le processus enfant hérite de l'espace d'adressage de son parent, CreateProcess() nécessite de charger un programme spécifié dans l'espace d'adressage du processus enfant lors de la création du processus. De plus, alors que fork() ne reçoit aucun paramètre, CreateProcess() n'attend pas moins de dix paramètres.

Le programme C présenté dans la figure 3.10 illustre la fonction CreateProcess() , qui crée un processus enfant qui charge l'application mspaint.exe.
Nous optons pour la plupart des valeurs par défaut des dix paramètres transmis à Cre-ateProcess(). Les lecteurs intéressés à approfondir les détails de la création et de la gestion des processus dans l' API Windows sont encouragés à consulter les notes bibliographiques à la fin de ce chapitre.

Les deux paramètres passés à la fonction CreateProcess() sont des instances des structures STARTUPINFO et PROCESS INFORMATION .

STARTUPINFO spécifie de nombreuses propriétés du nouveau processus, telles que la fenêtre

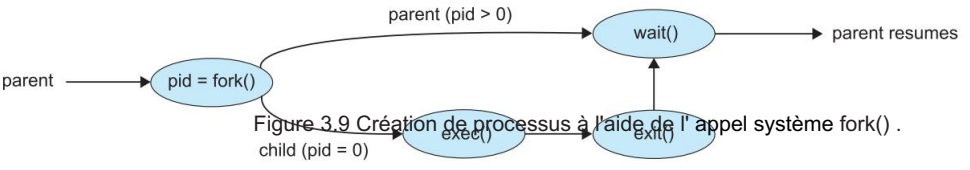


Figure 3.9 Création de processus à l'aide de l' appel système fork() .

```
#include <stdio.h> #include
<windows.h>

int principal (VOID)
{
INFO DEMARRAGE si ;
INFORMATIONS SUR LE PROCESSUS pi ;

    /* alloue de la mémoire */
    ZeroMemory(&si, sizeof(si)); si.cb =
    taillede(si); ZeroMemory(&pi,
    taillede(pi));

    /* crée un processus enfant */ if (!
    CreateProcess(NULL, /* utilise la ligne de commande */
      "C:     WINDOWS     system32     mspaint.exe", /* commande */
      NULL, /* n'hérite pas du handle de processus */ NULL, /*
      n'hérite pas du handle de thread */ FALSE, /* désactiver
      gérer l'héritage */ 0, /* aucun indicateur de création */
      NULL, /* utiliser le bloc d'environnement
      du parent */ NULL, /* utiliser le répertoire existant du parent
      */ &si, &pi)) {



        fprintf(stderr, "Échec de la création du processus"); renvoie
        -1 ;

    } /* le parent attendra que l'enfant termine */
    WaitForSingleObject(pi.hProcess, INFINITE); printf("Enfant
    terminé");

    /* fermer les poignées */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figure 3.10 Création d'un processus distinct à l'aide de l'API Windows.

taille, apparence et gestion des fichiers d'entrée et de sortie standard. La structure PROCESS INFORMATION contient un handle et les identifiants du processus nouvellement créé et de son thread. Nous invoquons la fonction ZeroMemory() pour allouer de la mémoire à chacune de ces structures avant de procéder à CreateProcess().

Les deux premiers paramètres transmis à CreateProcess() sont le nom de l'application et les paramètres de ligne de commande. Si le nom de l'application est NULL (comme c'est le cas dans ce cas), le paramètre de ligne de commande spécifie l'application à charger.

Dans ce cas, nous chargeons l' application Microsoft Windows mspaint.exe . Au-delà de ces deux paramètres initiaux, nous utilisons les paramètres par défaut pour hériter des handles de processus et de thread ainsi que pour spécifier qu'il n'y aura pas d'indicateurs de création. Nous utilisons également le bloc d'environnement et le répertoire de démarrage existants du parent. Enfin, nous fournissons deux pointeurs vers les structures STARTUPINFO et PROCESS - INFORMATION créées au début du programme. Dans la figure 3.8, le processus parent attend que l'enfant termine en appelant l' appel système wait() . L'équivalent dans Windows est WaitForSingleObject(), qui reçoit un handle du processus enfant (pi.hProcess) et attend la fin de ce processus. Une fois le processus enfant terminé, le contrôle revient de la fonction WaitForSingleObject() dans le processus parent.

3.3.2 Fin du processus

Un processus se termine lorsqu'il termine l'exécution de son instruction finale et demande au système d'exploitation de la supprimer à l'aide de l' appel système exit() . À ce stade, le processus peut renvoyer une valeur d'état (généralement un entier) à son processus parent en attente (via l' appel système wait()). Toutes les ressources du processus, y compris la mémoire physique et virtuelle, les fichiers ouverts et les tampons d'E/S , sont libérées et récupérées par le système d'exploitation.

La résiliation peut également survenir dans d'autres circonstances. Un processus peut provoquer l'arrêt d'un autre processus via un appel système approprié (par exemple, TerminateProcess() sous Windows). Habituellement, un tel appel système ne peut être invoqué que par le parent du processus qui doit être terminé. Sinon, un utilisateur (ou une application qui se comporte mal) pourrait arrêter arbitrairement les processus d'un autre utilisateur.
Notez qu'un parent doit connaître l'identité de ses enfants s'il veut les licencier. Ainsi, lorsqu'un processus crée un nouveau processus, l'identité du processus nouvellement créé est transmise au parent.

Un parent peut mettre fin à l'exécution d'un de ses enfants pour diverses raisons, telles que celles-ci :

- L'enfant a dépassé son utilisation de certaines des ressources qui lui ont été allouées. (Pour déterminer si cela s'est produit, le parent doit disposer d'un mécanisme pour inspecter l'état de ses enfants.)
- La tâche assignée à l'enfant n'est plus nécessaire. • Le parent est

en train de se fermer et le système d'exploitation n'autorise pas un enfant à continuer si son parent se termine.

Certains systèmes ne permettent pas à un enfant d'exister si son parent a pris fin.
Dans de tels systèmes, si un processus se termine (normalement ou anormalement), alors tous ses enfants doivent également être terminés. Ce phénomène, appelé terminaison en cascade, est normalement initié par le système d'exploitation.

Pour illustrer l'exécution et l'arrêt du processus, considérons que, sous Linux et Sur les systèmes UNIX , nous pouvons terminer un processus en utilisant l' appel système exit() , en fournissant un statut de sortie en paramètre :

```
/* quitte avec le statut 1 */ exit(1);
```

En fait, en cas de terminaison normale, exit() sera appelé soit directement (comme indiqué ci-dessus), soit indirectement, car la bibliothèque d'exécution C (qui est ajoutée aux fichiers exécutables UNIX) inclura un appel à exit() par défaut.

Un processus parent peut attendre la fin d'un processus enfant en utilisant l' appel système wait() . L' appel système wait() reçoit un paramètre qui permet au parent d'obtenir le statut de sortie de l'enfant. Cet appel système renvoie également l'identifiant de processus de l'enfant terminé afin que le parent puisse savoir lequel de ses enfants a terminé :

```
pid t pid;
statut int ;

_ pid = attendre(&status);
```

Lorsqu'un processus se termine, ses ressources sont libérées par le système d'exploitation. Cependant, son entrée dans la table des processus doit y rester jusqu'à ce que le parent appelle wait(), car la table des processus contient l'état de sortie du processus. Un processus qui s'est terminé, mais dont le parent n'a pas encore appelé wait(), est appelé processus zombie . Tous les processus passent à cet état lorsqu'ils se terminent, mais ils n'existent généralement que brièvement en tant que zombies. Une fois que le parent appelle wait(), l'identifiant du processus zombie et son entrée dans la table des processus sont libérés.

Considérons maintenant ce qui se passerait si un parent n'invoquait pas wait() et se terminait à la place, laissant ainsi ses processus enfants orphelins. Les systèmes UNIX traditionnels ont résolu ce scénario en attribuant le processus init comme nouveau parent aux processus orphelins. (Rappelez-vous de la section 3.3.1 que init sert de racine à la hiérarchie des processus dans les systèmes UNIX .) Le processus init invoque périodiquement wait(), permettant ainsi de collecter l'état de sortie de tout processus orphelin et de libérer l'identifiant du processus orphelin et entrée de la table de processus.

Bien que la plupart des systèmes Linux aient remplacé init par systemd, ce dernier processus peut toujours jouer le même rôle, bien que Linux permette également à des processus autres que systemd d'hériter des processus orphelins et de gérer leur terminaison.

3.3.2.1 Hiérarchie des processus Android

En raison de contraintes de ressources telles qu'une mémoire limitée, les systèmes d'exploitation mobiles peuvent devoir mettre fin aux processus existants pour récupérer des ressources système limitées. Plutôt que de mettre fin à un processus arbitraire, Android a identifié une hiérarchie de processus d'importance, et lorsque le système doit mettre fin à un processus pour rendre des ressources disponibles pour un processus nouveau ou plus important, il met fin aux processus par ordre d'importance croissante. Du plus important au moins important, la hiérarchie des classifications de processus est la suivante :

- Processus de premier plan : le processus en cours visible à l'écran, représentant l'application avec laquelle l'utilisateur interagit actuellement
- Processus visible : un processus qui n'est pas directement visible au premier plan mais qui exécute une activité à laquelle le processus de premier plan fait référence (c'est-à-dire un processus exécutant une activité dont le statut est affiché sur le processus de premier plan).

- Processus de service : un processus similaire à un processus en arrière-plan mais qui exécute une activité visible pour l'utilisateur (telle que la diffusion de musique en streaming).

- Processus d'arrière-plan : un processus qui peut effectuer une activité mais n'est pas visible pour l'utilisateur.
- Processus vide : processus qui ne contient aucun composant actif associé à une application.

Si les ressources système doivent être récupérées, Android mettra d'abord fin aux processus vides, suivis des processus en arrière-plan, et ainsi de suite. Les processus se voient attribuer un classement par importance et Android tente d'attribuer à un processus un classement aussi élevé que possible. Par exemple, si un processus fournit un service et est également visible, la classification visible la plus importante lui sera attribuée.

De plus, les pratiques de développement Android suggèrent de suivre les lignes directrices du cycle de vie des processus. Lorsque ces directives sont suivies, l'état d'un processus sera enregistré avant la fin et repris à son état enregistré si l'utilisateur revient à l'application.

3.4 Communication interprocessus

Les processus s'exécutant simultanément dans le système d'exploitation peuvent être soit des processus indépendants, soit des processus coopérants. Un processus est indépendant s'il ne partage pas de données avec d'autres processus exécutés dans le système. Un processus coopère s'il peut affecter ou être affecté par les autres processus exécutés dans le système. De toute évidence, tout processus partageant des données avec d'autres processus est un processus coopérant.

Il existe plusieurs raisons de fournir un environnement permettant la coopération de processus :

- Partage d'information. Puisque plusieurs applications peuvent être intéressées par la même information (par exemple, copier-coller), nous devons fournir un environnement permettant un accès simultané à cette information.
- Accélération des calculs. Si nous voulons qu'une tâche particulière s'exécute plus rapidement, nous devons la diviser en sous-tâches, chacune s'exécutant en parallèle avec les autres. Notez qu'une telle accélération ne peut être obtenue que si l'ordinateur dispose de plusieurs cœurs de traitement.
- Modularité. Nous pouvons vouloir construire le système de manière modulaire, en divisant les fonctions du système en processus ou threads distincts, comme nous l'avons vu au chapitre 2.

Les processus coopérants nécessitent un mécanisme de communication interprocessus (IPC) qui leur permettra d'échanger des données, c'est-à-dire de s'envoyer et de recevoir des données les uns des autres. Il existe deux modèles fondamentaux de communication interprocessus : la mémoire partagée et la transmission de messages. Dans le modèle de mémoire partagée, une région de mémoire partagée par les processus coopérants est établie. Les processus peuvent ensuite échanger des informations en lisant et en écrivant des données dans la région partagée. Dans le modèle de transmission de messages,

ARCHITECTURE MULTIPROCESSUS – NAVIGATEUR CHROME

De nombreux sites Web contiennent du contenu actif, tel que JavaScript, Flash et HTML5, pour offrir une expérience de navigation Web riche et dynamique. Malheureusement, ces applications Web peuvent également contenir des bogues logiciels, ce qui peut entraîner des temps de réponse lents et même provoquer le blocage du navigateur Web. Ce n'est pas un gros problème dans un navigateur Web qui affiche le contenu d'un seul site Web. Mais la plupart des navigateurs Web contemporains proposent une navigation par onglets, qui permet à une seule instance d'une application de navigateur Web d'ouvrir plusieurs sites Web en même temps, chaque site se trouvant dans un onglet distinct. Pour basculer entre les différents sites, il suffit à l'utilisateur de cliquer sur l'onglet approprié. Cette disposition est illustrée ci-dessous :

Le problème avec cette approche est que si une application Web dans un onglet tombe en panne, l'ensemble du processus, y compris tous les autres onglets affichant des sites Web supplémentaires, se bloque également.

Le navigateur Web Chrome de Google a été conçu pour résoudre ce problème en utilisant une architecture multiprocessus. Chrome identifie trois types de processus différents : navigateur, moteurs de rendu et plug-ins.

- Le processus du navigateur est responsable de la gestion de l'interface utilisateur ainsi que des E/S disque et réseau. Un nouveau processus de navigateur est créé au démarrage de Chrome. Un seul processus de navigateur est créé.
- Les processus de rendu contiennent une logique pour le rendu des pages Web. Ainsi, ils contiennent la logique de gestion du HTML, du Javascript, des images, etc. En règle générale, un nouveau processus de rendu est créé pour chaque site Web ouvert dans un nouvel onglet, et plusieurs processus de rendu peuvent donc être actifs en même temps.
- Un processus de plug-in est créé pour chaque type de plug-in (tel que Flash ou QuickTime) utilisé. Les processus de plug-in contiennent le code du plug-in ainsi que du code supplémentaire qui permet au plug-in de communiquer avec les processus de rendu associés et le processus du navigateur.

L'avantage de l'approche multiprocessus est que les sites Web fonctionnent de manière isolée les uns des autres. Si un site Web tombe en panne, seul son processus de rendu est affecté ; tous les autres processus restent indemnes. De plus, les processus de rendu s'exécutent dans un bac à sable, ce qui signifie que l'accès aux E/S du disque et du réseau est restreint, minimisant ainsi les effets de toute faille de sécurité.

la communication s'effectue au moyen de messages échangés entre les processus coopérants. Les deux modèles de communication sont comparés dans la figure 3.11.

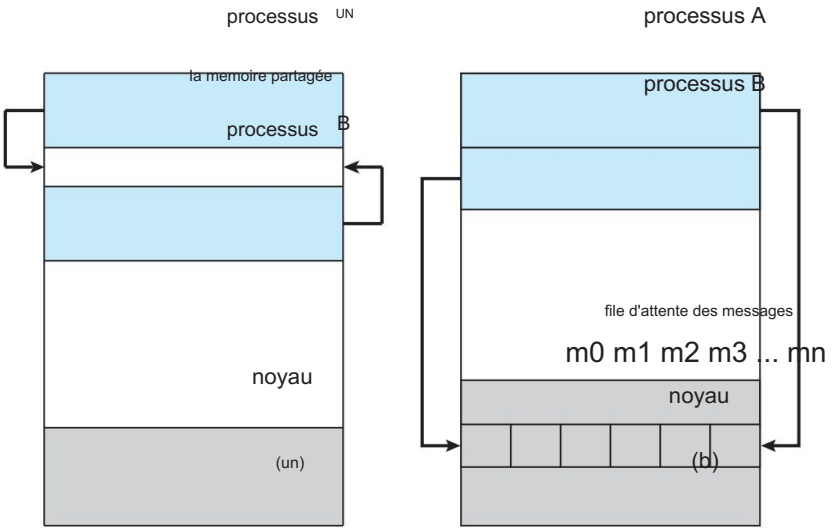


Figure 3.11 Modèles de communication. (a) Mémoire partagée. (b) Passage de messages.

Les deux modèles que nous venons de mentionner sont courants dans les systèmes d'exploitation, et de nombreux systèmes implémentent les deux. Le passage de messages est utile pour échanger de plus petites quantités de données, car aucun conflit ne doit être évité. La transmission de messages est également plus facile à mettre en œuvre dans un système distribué que dans la mémoire partagée. (Bien qu'il existe des systèmes qui fournissent une mémoire partagée distribuée, nous le faisons ne les prend pas en compte dans ce texte.) La mémoire partagée peut être plus rapide que la transmission de messages, car les systèmes de transmission de messages sont généralement implémentés à l'aide du système appels et nécessitent donc la tâche plus longue d'intervention du noyau.
Dans les systèmes à mémoire partagée, les appels système sont requis uniquement pour établir des régions de mémoire partagée. Une fois la mémoire partagée établie, tous les accès sont traités comme les accès de routine à la mémoire, et aucune assistance du noyau n'est requise.

Dans les sections 3.5 et 3.6, nous explorons plus en détail les systèmes de mémoire partagée et de transmission de messages.

3.5 IPC dans les systèmes à mémoire partagée

La communication interprocessus utilisant la mémoire partagée nécessite de communiquer processus pour établir une région de mémoire partagée. Généralement, une mémoire partagée La région réside dans l'espace d'adressage du processus créant la mémoire partagée segment. Autres processus souhaitant communiquer en utilisant cette mémoire partagée Le segment doit l'attacher à son espace d'adressage. Rappelons que, normalement, le système d'exploitation essaie d'empêcher un processus d'accéder au serveur d'un autre processus. mémoire. La mémoire partagée nécessite que deux processus ou plus acceptent de supprimer cette restriction. Ils peuvent alors échanger des informations en lisant et en écrivant données dans les zones partagées. La forme des données et l'emplacement sont déterminés par ces processus et ne sont pas sous le contrôle du système d'exploitation. Les processus sont également chargés de s'assurer qu'ils n'écrivent pas dans le même emplacement simultanément.

Pour illustrer le concept de processus de coopération, considérons le problème producteur-consommateur, qui est un paradigme courant pour les processus de coopération. Un processus producteur produit des informations qui sont consommées par un processus consommateur . Par exemple, un compilateur peut produire du code assembleur consommé par un assembleur. L'assembleur, à son tour, peut produire des modules objets qui sont consommés par le chargeur. Le problème producteur-consommateur fournit également une métaphore utile pour le paradigme client-serveur. Nous considérons généralement un serveur comme un producteur et un client comme un consommateur. Par exemple, un serveur Web produit (c'est-à-dire fournit) du contenu Web tel que des fichiers HTML et des images, qui sont consommés (c'est-à-dire lus) par le navigateur Web client demandant la ressource.

Une solution au problème producteur-consommateur utilise la mémoire partagée. Pour permettre aux processus producteur et consommateur de s'exécuter simultanément, nous devons disposer d'un tampon d'éléments pouvant être rempli par le producteur et vidé par le consommateur. Ce tampon résidera dans une région de mémoire partagée par les processus producteur et consommateur. Un producteur peut produire un article tandis que le consommateur en consomme un autre. Le producteur et le consommateur doivent être synchronisés, afin que le consommateur n'essaye pas de consommer un bien qui n'a pas encore été produit.

Deux types de tampons peuvent être utilisés. Le tampon illimité n'impose aucune limite pratique à la taille du tampon. Le consommateur devra peut-être attendre de nouveaux articles, mais le producteur peut toujours produire de nouveaux articles. Le tampon limité suppose une taille de tampon fixe. Dans ce cas, le consommateur doit attendre si le tampon est vide, et le producteur doit attendre si le tampon est plein.

Examinons de plus près comment le tampon limité illustre la communication interprocessus utilisant la mémoire partagée. Les variables suivantes résident dans une région de mémoire partagée par les processus producteur et consommateur :

```
#define TAILLE DU TAMPON 10

structure typedef {
        ...
} article;

tampon d'élément[TAILLE DU
TAMPON] ; int
dans = 0 ; int sortie = 0 ;
```

Le tampon partagé est implémenté sous la forme d'un tableau circulaire avec deux pointeurs logiques : entrée et sortie. La variable in pointe vers la prochaine position libre dans le tampon ; out pointe vers la première position complète dans le tampon. Le tampon est vide lorsque in == out ; le tampon est plein lorsque ((in + 1) % BUFFER SIZE) == out.

Le code du processus producteur est présenté à la figure 3.12 et le code du processus consommateur est présenté à la figure 3.13. Le processus producteur possède une variable locale suivante produite dans laquelle est stocké le nouvel élément à produire. Le processus de consommation possède une variable locale suivante consommée dans laquelle l'élément à consommer est stocké.

Ce schéma autorise au plus BUFFER SIZE - 1 éléments dans le tampon en même temps. Nous vous laissons comme exercice le soin de fournir une solution dans laquelle les éléments BUFFER SIZE peuvent être dans le tampon en même temps. Dans la section 3.7.1, nous illustrons l' API POSIX pour la mémoire partagée.

article produit ensuite ;

```
while (true) { /*
    produit un élément lors de la prochaine production */

    while (((entrée + 1) % TAILLE DU BUFFER) == sortie)
        ; /* ne fais rien */

    buffer[in] = prochain produit ; en = (en
    + 1) % TAILLE DU TAMPON ;
}
```

Figure 3.12 Le processus producteur utilisant la mémoire partagée.

Un problème que cette illustration ne traite pas concerne la situation dans laquelle le processus producteur et le processus consommateur tentent d'accéder simultanément au tampon partagé. Dans les chapitres 6 et 7, nous discutons de la manière dont la synchronisation entre les processus coopérants peut être mise en œuvre efficacement dans un environnement de mémoire partagée.

3.6 IPC dans les systèmes de transmission de messages

Dans la section 3.5, nous avons montré comment les processus coopérants peuvent communiquer dans un environnement de mémoire partagée. Le schéma nécessite que ces processus partagent une région de mémoire et que le code permettant d'accéder et de manipuler la mémoire partagée soit écrit explicitement par le programmeur de l'application. Une autre façon d'obtenir le même effet consiste pour le système d'exploitation à fournir les moyens permettant aux processus coopérants de communiquer entre eux via une fonction de transmission de messages.

article ensuite consommé ;

```
while (vrai) { while
    (in == out)
        ; /* ne fais rien */

    prochaine consommation = buffer[out];
    sortie = (sortie + 1) % TAILLE DU TAMPON ;

    /* consomme l'objet lors de la prochaine consommation */
}
```

Figure 3.13 Processus consommateur utilisant la mémoire partagée.

La transmission de messages fournit un mécanisme permettant aux processus de communiquer et de synchroniser leurs actions sans partager le même espace d'adressage. Il est particulièrement utile dans un environnement distribué, où les processus communicants peuvent résider sur différents ordinateurs connectés par un réseau. Par exemple, un programme de chat sur Internet pourrait être conçu de manière à ce que les participants au chat communiquent entre eux en échangeant des messages.

Une fonction de transmission de messages fournit au moins deux opérations :

envoyer le message)

et

recevoir (message)

Les messages envoyés par un processus peuvent être de taille fixe ou variable. Si seuls des messages de taille fixe peuvent être envoyés, la mise en œuvre au niveau du système est simple. Cette restriction rend cependant la tâche de programmation plus difficile. À l'inverse, les messages de taille variable nécessitent une implémentation plus complexe au niveau du système, mais la tâche de programmation devient plus simple. Il s'agit d'un type de compromis courant dans la conception des systèmes d'exploitation.

Si les processus P et Q veulent communiquer, ils doivent s'envoyer et recevoir des messages l'un de l'autre : un lien de communication doit exister entre eux. Ce lien peut être mis en œuvre de diverses manières. Nous ne nous intéressons pas ici à l'implémentation physique du lien (comme la mémoire partagée, le bus matériel ou le réseau, qui sont traités au chapitre 19) mais plutôt à son implémentation logique. Voici plusieurs méthodes pour implémenter logiquement un lien et les opérations send()/receive() :

- Communication directe ou indirecte
- Communication synchrone ou asynchrone • Mise en mémoire tampon automatique ou explicite

Nous examinons ensuite les problèmes liés à chacune de ces fonctionnalités.

3.6.1 Nommer Les

processus qui souhaitent communiquer doivent pouvoir se référer les uns aux autres. Ils peuvent utiliser la communication directe ou indirecte.

En communication directe, chaque processus qui souhaite communiquer doit nommer explicitement le destinataire ou l'expéditeur de la communication. Dans ce schéma, les primitives send() et contain() sont définies comme :

- send(P, message) : envoie un message au processus P. • contain(Q, message) : reçoit un message du processus Q.

Un lien de communication dans ce schéma a les propriétés suivantes :

- Un lien est établi automatiquement entre chaque paire de processus souhaitant communiquer. Les processus n'ont besoin que de connaître l'identité de chacun pour communiquer.

• Un lien est associé à exactement deux processus. • Entre chaque paire de processus, il existe exactement un lien.

Ce schéma présente une symétrie dans l'adressage ; c'est-à-dire que le processus émetteur et le processus récepteur doivent tous deux nommer l'autre pour communiquer. Une variante de ce schéma utilise l'asymétrie dans l'adressage. Ici, seul l'expéditeur nomme le destinataire ; le destinataire n'est pas tenu de nommer l'expéditeur. Dans ce schéma, les primitives send() et contain() sont définies comme suit :

• send(P, message) : envoie un message au processus P. • contain(id, message) : reçoit un message de n'importe quel processus. La variable id est définie sur le nom du processus avec lequel la communication a eu lieu.

L'inconvénient de ces deux schémas (symétrique et asymétrique) est la modularité limitée des définitions de processus qui en résultent. Changer l'identifiant d'un processus peut nécessiter l'examen de toutes les autres définitions de processus.
Toutes les références à l'ancien identifiant doivent être retrouvées, afin qu'elles puissent être modifiées vers le nouvel identifiant. En général, de telles techniques de codage en dur , où les identifiants doivent être explicitement indiqués, sont moins souhaitables que les techniques impliquant l'indirection, comme décrit ci-après.

Avec la communication indirecte, les messages sont envoyés et reçus depuis des boîtes aux lettres ou des ports. Une boîte aux lettres peut être considérée de manière abstraite comme un objet dans lequel des messages peuvent être placés par des processus et duquel des messages peuvent être supprimés. Chaque boîte aux lettres possède une identification unique. Par exemple, les files d'attente de messages POSIX utilisent une valeur entière pour identifier une boîte aux lettres. Un processus peut communiquer avec un autre processus via un certain nombre de boîtes aux lettres différentes, mais deux processus ne peuvent communiquer que s'ils disposent d'une boîte aux lettres partagée. Les primitives send() et contain() sont définies comme suit :

• envoyer(A, message) : envoyer un message à la boîte aux lettres A.

• recevoir(A, message) : recevoir un message de la boîte aux lettres A.

Dans ce schéma, un lien de communication a les propriétés suivantes :

• Un lien n'est établi entre une paire de processus que si les deux membres de la paire disposent d'une boîte aux lettres partagée.

• Un lien peut être associé à plus de deux processus. • Entre chaque paire de processus communicants, plusieurs liens différents peuvent exister, chaque lien correspondant à une boîte aux lettres.

Supposons maintenant que les processus P1, P2 et P3 partagent tous la boîte aux lettres A. Le processus P1 envoie un message à A, tandis que P2 et P3 exécutent tous deux une réception () de A. Quel processus recevra le message envoyé par P1 ? La réponse dépend de la méthode que nous choisissons parmi les suivantes :

• Autoriser qu'un lien soit associé à deux processus au maximum.

• Autoriser au plus un processus à la fois à exécuter une opération recevoir() . • Permettre au système de sélectionner arbitrairement quel processus recevra le message (c'est-à-dire que P2 ou P3, mais pas les deux, recevront le message). Le système peut définir un algorithme pour sélectionner le processus qui recevra le message (par exemple, round robin, où les processus reçoivent des messages à tour de rôle). Le système peut identifier le destinataire auprès de l'expéditeur.

Une boîte aux lettres peut appartenir soit à un processus, soit au système d'exploitation. Si la boîte aux lettres appartient à un processus (c'est-à-dire que la boîte aux lettres fait partie de l'espace d'adressage du processus), alors nous distinguons le propriétaire (qui ne peut recevoir des messages que via cette boîte aux lettres) et l'utilisateur (qui ne peut envoyer que des messages à la boîte aux lettres). Étant donné que chaque boîte aux lettres a un propriétaire unique, il ne peut y avoir aucune confusion quant au processus qui doit recevoir un message envoyé à cette boîte aux lettres. Lorsqu'un processus propriétaire d'une boîte aux lettres se termine, la boîte aux lettres disparaît. Tout processus qui envoie ensuite un message à cette boîte aux lettres doit être informé que la boîte aux lettres n'existe plus.

En revanche, une boîte aux lettres appartenant au système d'exploitation a sa propre existence. Il est indépendant et n'est rattaché à aucun processus particulier. Le système d'exploitation doit alors fournir un mécanisme permettant à un processus d'effectuer les opérations suivantes :

• Créez une nouvelle boîte aux lettres.

• Envoyez et recevez des messages via la boîte aux lettres.

• Supprimer une boîte aux lettres.

Le processus qui crée une nouvelle boîte aux lettres est le propriétaire par défaut de cette boîte aux lettres. Initialement, le propriétaire est le seul processus capable de recevoir des messages via cette boîte aux lettres. Cependant, les privilèges de propriété et de réception peuvent être transmis à d'autres processus via des appels système appropriés. Bien entendu, cette disposition pourrait conduire à avoir plusieurs récepteurs pour chaque boîte aux lettres.

3.6.2 Synchronisation

La communication entre les processus s'effectue via des appels aux primitives send() et contain() . Il existe différentes options de conception pour implémenter chaque primitive. La transmission des messages peut être bloquante ou non bloquante, également appelée synchrone et asynchrone. (Tout au long de ce texte, vous rencontrerez les concepts de comportement synchrone et asynchrone en relation avec divers algorithmes du système d'exploitation.)

• Blocage de l'envoi. Le processus d'envoi est bloqué jusqu'à ce que le message soit reçu par le processus de réception ou par la boîte aux lettres.

• Envoi non bloquant. Le processus d'envoi envoie le message et reprend opération.

• Blocage de la réception. Le récepteur bloque jusqu'à ce qu'un message soit disponible.

• Réception non bloquante. Le destinataire récupère soit un message valide, soit un nul.

```
message ensuite produit ;

while (true) { /*
        produit un élément lors de la prochaine production */

        envoyer (prochain produit);
}
```

Figure 3.14 Processus producteur utilisant la transmission de messages.

Différentes combinaisons de send() et contain() sont possibles. Lorsque send() et contain() bloquent, nous avons un rendez-vous entre l'expéditeur et le destinataire. La solution au problème producteur-consommateur devient triviale lorsque nous utilisons les instructions bloquantes send() et contain() . Le producteur appelle simplement l'appel bloquant send() et attend que le message soit remis au destinataire ou à la boîte aux lettres. De même, lorsque le consommateur invoque recevoir(), il bloque jusqu'à ce qu'un message soit disponible. Ceci est illustré dans les figures 3.14 et 3.15.

3.6.3 Mise en mémoire tampon

Que la communication soit directe ou indirecte, les messages échangés par les processus communicants résident dans une file d'attente temporaire. Fondamentalement, de telles files d'attente peuvent être implémentées de trois manières :

- Capacité nulle. La file d'attente a une longueur maximale de zéro ; ainsi, le lien ne peut contenir aucun message en attente. Dans ce cas, l'expéditeur doit bloquer jusqu'à ce que le destinataire reçoive le message.
- Capacité limitée. La file d'attente a une longueur finie n ; ainsi, au plus n messages peuvent y résider. Si la file d'attente n'est pas pleine lorsqu'un nouveau message est envoyé, le message est placé dans la file d'attente (soit le message est copié, soit un pointeur vers le message est conservé) et l'expéditeur peut continuer l'exécution sans

```
message ensuite consommé ;

while (true) { recevoir
        (prochaine consommation);

        /* consomme l'objet lors de la prochaine consommation */
}
```

Figure 3.15 Processus consommateur utilisant la transmission de messages.

en attendant. La capacité du lien est cependant limitée. Si le lien est plein, l'expéditeur doit bloquer jusqu'à ce que de l'espace soit disponible dans la file

d'attente. • Capacité illimitée. La longueur de la file d'attente est potentiellement infinie ; ainsi, n'importe quel nombre de messages peut y attendre. L'expéditeur ne bloque jamais.

Le cas à capacité nulle est parfois appelé système de messagerie sans mise en mémoire tampon. Les autres cas sont appelés systèmes avec mise en mémoire tampon automatique.

3.7 Exemples de systèmes IPC

Dans cette section, nous explorons quatre systèmes IPC différents. Nous abordons d'abord l' API POSIX pour la mémoire partagée, puis discutons du passage de messages dans le système d'exploitation Mach. Ensuite, nous présentons Windows IPC, qui utilise de manière intéressante la mémoire partagée comme mécanisme permettant de fournir certains types de transmission de messages. Nous concluons avec les pipes, l'un des premiers mécanismes IPC sur les systèmes UNIX .

3.7.1 Mémoire partagée POSIX

Plusieurs mécanismes IPC sont disponibles pour les systèmes POSIX , notamment la mémoire partagée et la transmission de messages. Ici, nous explorons l' API POSIX pour la mémoire partagée.

La mémoire partagée POSIX est organisée à l'aide de fichiers mappés en mémoire, qui associent la région de la mémoire partagée à un fichier. Un processus doit d'abord créer un objet de mémoire partagée à l'aide de l' appel système shm open() , comme suit :

```
fd = shm open(nom, O CREAT | O RDWR, 0666);
```

Le premier paramètre spécifie le nom de l'objet de mémoire partagée. Les processus qui souhaitent accéder à cette mémoire partagée doivent faire référence à l'objet par ce nom.
Les paramètres suivants précisent que l'objet de mémoire partagée doit être créé s'il n'existe pas encore (O CREAT) et que l'objet est ouvert en lecture et en écriture (O RDWR). Le dernier paramètre établit les autorisations d'accès aux fichiers de l'objet de mémoire partagée. Un appel réussi à shm open() renvoie un descripteur de fichier entier pour l'objet de mémoire partagée.

Une fois l'objet établi, la fonction ftruncate() est utilisée pour configurer la taille de l'objet en octets. L'appel

```
tronquer(fd, 4096);
```

définit la taille de l'objet à 4 096 octets.

Enfin, la fonction mmap() établit un fichier mappé en mémoire contenant l'objet de mémoire partagée. Il renvoie également un pointeur vers le fichier mappé en mémoire utilisé pour accéder à l'objet de mémoire partagée.

Les programmes présentés dans les figures 3.16 et 3.17 utilisent le modèle producteur-consommateur pour mettre en œuvre la mémoire partagée. Le producteur établit un objet de mémoire partagée et écrit dans la mémoire partagée, tandis que le consommateur lit depuis la mémoire partagée.

```
#include <stdio.h>
#include <stdlib.h> #include
<string.h> #include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>


#include <sys/mman.h>

int main() { /*

la taille (en octets) de l'objet de mémoire partagée */ const int SIZE =
4096; /* nom de l'objet
mémoire partagée */ const char *name = "OS"; /*
chaînes écrites dans la mémoire
partagée */ const char *message 0 = "Bonjour"; const
char *message 1 = "Monde !";


/* Descripteur de fichier de mémoire partagée */
int fd; /*
pointeur vers l'objet de mémoire partagée */ char
*ptr;

    /* crée l'objet de mémoire partagée */ fd = shm
    open(name,O CREAT | O RDWR,0666);

    /* configure la taille de l'objet de mémoire partagée */ ftruncate(fd, SIZE);

    /* mappage mémoire de l'objet mémoire partagée */ ptr =
    (char *)
      mmap(0, TAILLE, PROT READ | PROT WRITE, CARTE PARTAGÉE, fd, 0);

    /* écriture dans l'objet mémoire partagée */
    sprintf(ptr,"%s",message 0); ptr +=
    strlen(message 0);
    sprintf(ptr,"%s",message 1); ptr +=
    strlen(message 1);

    renvoie 0 ;
}
```

Figure 3.16 Processus producteur illustrant l'API de mémoire partagée POSIX.

Le producteur, illustré à la figure 3.16, crée un objet de mémoire partagée nommé OS et écrit la fameuse chaîne « Hello World ! » à la mémoire partagée. Le programme mappe la mémoire d'un objet de mémoire partagée de la taille spécifiée et permet d'écrire sur l'objet. L'indicateur MAP SHARED spécifie que les modifications apportées à l'objet de mémoire partagée seront visibles par tous les processus partageant l'objet. Notez que nous écrivons dans l'objet de mémoire partagée en appelant la fonction sprintf() et en écrivant la chaîne formatée dans le pointeur ptr. Après chaque écriture, il faut incrémenter le pointeur du nombre d'octets écrits.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main() { /

* la taille (en octets) de l'objet de mémoire partagée */ const int
SIZE = 4096; /* nom de l'objet
mémoire partagée */ const char *name = "OS"; /*
Descripteur de fichier de
mémoire partagée */ int fd; /* pointeur vers
l'objet
de mémoire partagée */ char *ptr;


    /* ouvre l'objet de mémoire partagée */ fd =
    shm open(name, O RDONLY, 0666);

    /* mappage mémoire de l'objet mémoire partagée */ ptr
    = (char *) mmap(0,
      SIZE, PROT READ | PROT WRITE, MAP SHARED, fd, 0);

    /* lecture depuis l'objet mémoire partagée */ printf("%s",
    (char *)ptr);

    /* supprime l'objet de mémoire partagée */ shm
    unlink(name);

    renvoie 0 ;
}
```

Figure 3.17 Processus consommateur illustrant l'API de mémoire partagée POSIX.

Le processus consommateur, illustré à la figure 3.17, lit et affiche le contenu de la mémoire partagée. Le consommateur invoque également la fonction shm unlink() , qui supprime le segment de mémoire partagée une fois que le consommateur y a accédé. Nous proposons d'autres exercices utilisant l' API de mémoire partagée POSIX dans les exercices de programmation à la fin de ce chapitre. De plus, nous fournissons une couverture plus détaillée du mappage de mémoire dans la section 13.5.

3.7.2 Transmission de messages Mach Comme

exemple de transmission de messages, nous considérons ensuite le système d'exploitation Mach. Mach a été spécialement conçu pour les systèmes distribués, mais s'est avéré également adapté aux systèmes de bureau et mobiles, comme en témoigne son inclusion dans les systèmes d'exploitation macOS et iOS , comme indiqué au chapitre 2.

Le noyau Mach prend en charge la création et la destruction de plusieurs tâches, similaires aux processus mais comportant plusieurs threads de contrôle et moins de ressources associées. La plupart des communications dans Mach, y compris toutes les communications inter-tâches, s'effectuent par messages. Les messages sont envoyés et reçus depuis des boîtes aux lettres, appelées ports en Mach. Les ports sont de taille limitée et unidirectionnels ; pour une communication bidirectionnelle, un message est envoyé à un port et une réponse est envoyée à un port de réponse distinct . Chaque port peut avoir plusieurs émetteurs, mais un seul récepteur. Mach utilise des ports pour représenter des ressources telles que des tâches, des threads, de la mémoire et des processeurs, tandis que la transmission de messages fournit une approche orientée objet pour interagir avec ces ressources et services système. La transmission de messages peut se produire entre deux ports quelconques sur le même hôte ou sur des hôtes distincts sur un système distribué.

À chaque port est associé un ensemble de droits de port qui identifient les capacités nécessaires à une tâche pour interagir avec le port. Par exemple, pour qu'une tâche reçoive un message d'un port, elle doit avoir la capacité MACH PORT RIGHT RECEIVE pour ce port. La tâche qui crée un port est le propriétaire de ce port, et le propriétaire est la seule tâche autorisée à recevoir des messages de ce port. Le propriétaire d'un port peut également manipuler les capacités d'un port.

Cela se fait le plus souvent lors de l'établissement d'un port de réponse. Par exemple, supposons que la tâche T1 possède le port P1 et qu'elle envoie un message au port P2, qui appartient à la tâche T2. Si T1 s'attend à recevoir une réponse de T2, il doit accorder à T2 le droit MACH PORT RIGHT SEND pour le port P1. La propriété des droits de port se situe au niveau de la tâche, ce qui signifie que tous les threads appartenant à la même tâche partagent les mêmes droits de port. Ainsi, deux threads appartenant à la même tâche peuvent facilement communiquer en échangeant des messages via le port par thread associé à chaque thread.

Lorsqu'une tâche est créée, deux ports spéciaux (le port Task Self et le port Notify) sont également créés. Le noyau dispose de droits de réception sur le port Task Self, qui permet à une tâche d'envoyer des messages au noyau. Le noyau peut envoyer une notification des occurrences d'événements au port Notify d'une tâche (sur lequel, bien entendu, la tâche a des droits de réception).

L' appel de fonction mach port allocate() crée un nouveau port et alloue de l'espace pour sa file d'attente de messages. Il identifie également les droits du port. Chaque droit de port représente un nom pour ce port, et un port n'est accessible que via

un droit. Les noms de ports sont de simples valeurs entières et se comportent un peu comme les descripteurs de fichiers UNIX . L'exemple suivant illustre la création d'un port à l'aide de cette API :

```
port mach port t ; // le nom du port à droite

mach port allocate( mach
    task self(), // une tâche faisant référence à elle-même
    MACH PORT RIGHT RECEIVE, // le droit pour ce port
    &port); // le nom du port à droite
```

Chaque tâche a également accès à un port d'amorçage, ce qui permet à une tâche d'enregistrer un port qu'elle a créé auprès d'un serveur d'amorçage à l'échelle du système. Une fois qu'un port a été enregistré auprès du serveur d'amorçage, d'autres tâches peuvent rechercher le port dans ce registre et obtenir les droits d'envoi de messages vers le port.

La file d'attente associée à chaque port est de taille finie et est initialement vide.
Au fur et à mesure que les messages sont envoyés au port, ils sont copiés dans la file d'attente.
Tous les messages sont transmis de manière fiable et ont la même priorité. Mach garantit que plusieurs messages provenant du même expéditeur sont mis en file d'attente dans l'ordre premier entré, premier sorti (FIFO) , mais ne garantit pas un ordre absolu. Par exemple, les messages provenant de deux expéditeurs peuvent être mis en file d'attente dans n'importe quel ordre.

Les messages Mach contiennent les deux champs suivants :

- Un en-tête de message de taille fixe contenant des métadonnées sur le message, notamment la taille du message ainsi que les ports source et de destination.
  Généralement, le thread expéditeur attend une réponse, de sorte que le nom du port de la source est transmis à la tâche réceptrice, qui peut l'utiliser comme « adresse de retour » lors de l'envoi d'une réponse.
- Un corps

de taille variable contenant des données.

Les messages peuvent être simples ou complexes. Un message simple contient des données utilisateur ordinaires et non structurées qui ne sont pas interprétées par le noyau. Un message complexe peut contenir des pointeurs vers des emplacements mémoire contenant des données (appelées données « hors ligne ») ou peut également être utilisé pour transférer des droits de port vers une autre tâche. Les pointeurs de données hors ligne sont particulièrement utiles lorsqu'un message doit transmettre de grandes quantités de données. Un message simple nécessiterait de copier et de regrouper les données dans le message ; la transmission de données hors ligne nécessite uniquement un pointeur faisant référence à l'emplacement mémoire où les données sont stockées

La fonction mach msg() est l' API standard pour l'envoi et la réception de messages. La valeur de l'un des paramètres de la fonction (MACH SEND MSG ou MACH RCV MSG) indique s'il s'agit d'une opération d'envoi ou de réception.
Nous illustrons maintenant comment il est utilisé lorsqu'une tâche client envoie un message simple à une tâche serveur. Supposons qu'il existe deux ports (client et serveur) associés respectivement aux tâches client et serveur. Le code de la figure 3.18 montre la tâche client construisant un en-tête et envoyant un message au serveur, ainsi que la tâche serveur recevant le message envoyé par le client.

L' appel de fonction mach msg() est invoqué par les programmes utilisateur pour effectuer la transmission de messages. mach msg() invoque ensuite la fonction mach msg trap(), qui est un appel système au noyau Mach. Dans le noyau, mach msg trap() appelle ensuite la fonction mach msg overwrite trap(), qui gère ensuite la transmission réelle du message.

```
#include<mach/mach.h>

struct message { mach
    msg en-tête t en-tête ; données
    entières ; } ;

mach port t client; mach
port t serveur ;

    /* Code client */

message de structure message ;

// construit l'en-tête
message.header.msgh size = sizeof(message);
message.header.msgh port distant = serveur ;
message.header.msgh port local = client ;

// envoie le message mach
msg(&message.header, // en-tête du message MACH SEND
    MSG, // envoi d'un message sizeof(message), //
    taille du message envoyé 0, // taille maximale du message
    reçu - inutile MACH PORT NULL , // nom du port de réception - inutile
    MACH MSG TIMEOUT NONE, // pas de délai d'attente MACH PORT
    NULL // pas de port de notification );

    /* Code du serveur */

message de structure message ;

// réception du message mach
msg(&message.header, // en-tête du message MACH RCV
    MSG, // envoi d'un message 0, // taille du
    message envoyé sizeof(message), //
    taille maximale du serveur de messages reçu, // nom du port de réception
    MACH MSG TIMEOUT NONE, // pas de
    délai d'attente MACH PORT NULL // pas de port de
    notification );
```

Figure 3.18 Exemple de programme illustrant le passage de messages en Mach.

Les opérations d'envoi et de réception elles-mêmes sont flexibles. Par exemple, lorsqu'un message est envoyé vers un port, sa file d'attente peut être pleine. Si la file d'attente n'est pas pleine, le message est copié dans la file d'attente et la tâche d'envoi continu

la file d'attente du port est pleine, l'expéditeur dispose de plusieurs options (spécifiées via les paramètres à mach msg() :

1. Attendez indéfiniment jusqu'à ce qu'il y ait de la place dans la file d'attente.
2. Attendez au maximum n millisecondes.
3. N'attendez pas du tout mais revenez immédiatement.
4. Mettre temporairement en cache un message. Ici, un message est transmis à l'opérateur système à conserver, même si la file d'attente dans laquelle ce message est envoyé envoyé est complet. Lorsque le message peut être mis en file d'attente, une notification le message est renvoyé à l'expéditeur. Un seul message dans une file d'attente pleine peut être en attente à tout moment pour un thread d'envoi donné.

La dernière option est destinée aux tâches serveur. Après avoir terminé une requête, un serveur la tâche peut devoir envoyer une réponse unique à la tâche qui a demandé le service, mais il doit également continuer avec d'autres demandes de service, même si le port de réponse pour un client est rassasié.

Le problème majeur des systèmes de messagerie est généralement la mauvaise performance causée par la copie des messages du port de l'expéditeur vers celui du destinataire. port. Le système de messagerie Mach tente d'éviter les opérations de copie en utilisant techniques de gestion de la mémoire virtuelle (Chapitre 10). Essentiellement, les cartes de Mach l'espace d'adressage contenant le message de l'expéditeur dans l'adresse du destinataire espace. Par conséquent, le message lui-même n'est jamais réellement copié, car l'expéditeur et le récepteur accède à la même mémoire. Cette technique de gestion des messages fournit une amélioration importante des performances mais ne fonctionne que pour les messages intra-système.

3.7.3 Fenêtres

Le système d'exploitation Windows est un exemple de conception moderne qui utilise modularité pour augmenter les fonctionnalités et réduire le temps nécessaire à la mise en œuvre de nouvelles fonctionnalités. Windows prend en charge plusieurs environnements d'exploitation ou sous-systèmes. Les programmes d'application communiquent avec ces sous-systèmes via un mécanisme de transmission de messages. Ainsi, les programmes d'application peuvent être considérés comme des clients d'un serveur de sous-système.

La fonction de transmission de messages dans Windows est appelée fonction ALPC (Advanced Local Procedure Call) . Il est utilisé pour la communication entre deux processus sur la même machine. Il est similaire à l'appel de procédure à distance standard (RPC) mécanisme largement utilisé, mais optimisé et spécifique à Windows. (Les appels de procédure à distance sont traités en détail dans la section 3.8.2.) Comme Mach, Windows utilise un objet port pour établir et maintenir une connexion entre deux processus. Windows utilise deux types de ports : les ports de connexion et les ports de communication.

Les processus serveur publient des objets de port de connexion visibles par tous les processus. Lorsqu'un client souhaite obtenir des services d'un sous-système, il ouvre un handle vers le l'objet port de connexion du serveur et envoie une demande de connexion à ce port. Le serveur crée ensuite un canal et renvoie un handle au client. Le canal se compose d'une paire de ports de communication privés : un pour le client-serveur messages, l'autre pour les messages serveur-client. De plus, la communication les canaux prennent en charge un mécanisme de rappel qui permet au client et au serveur de accepter les demandes alors qu'ils attendraient normalement une réponse.

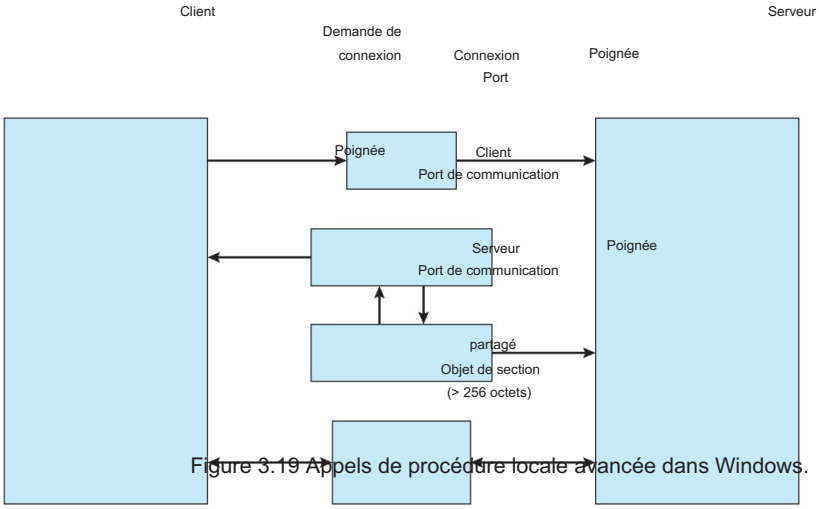


Figure 3.19 Appels de procédure locale avancée dans Windows.

Lorsqu'un canal ALPC est créé, l'une des trois techniques de transmission de messages est choisie :

1. Pour les petits messages (jusqu'à 256 octets), la file d'attente des messages du port est utilisée comme stockage intermédiaire et les messages sont copiés d'un processus à l'autre.

2. Les messages plus volumineux doivent être transmis via un objet section, qui est une région de mémoire partagée associée au canal.

3. Lorsque la quantité de données est trop importante pour tenir dans un objet de section, une API est disponible qui permet aux processus serveur de lire et d'écrire directement dans l'espace d'adressage d'un client.

Le client doit décider, lorsqu'il configure le canal, s'il devra envoyer un message volumineux. Si le client détermine qu'il souhaite envoyer des messages volumineux, il demande la création d'un objet section. De même, si le serveur décide que les réponses seront volumineuses, il crée un objet section. Pour que l'objet section puisse être utilisé, un petit message est envoyé contenant un pointeur et des informations de taille sur l'objet section. Cette méthode est plus compliquée que la première méthode répertoriée ci-dessus, mais elle évite la copie des données. La structure des appels de procédures locales avancées dans Windows est illustrée à la figure 3.19.

Il est important de noter que la fonction ALPC de Windows ne fait pas partie de l' API Windows et n'est donc pas visible pour le programmeur d'application. Au lieu de cela, les applications utilisant l' API Windows invoquent des appels de procédure à distance standard.
Lorsque le RPC est invoqué sur un processus sur le même système, le RPC est géré indirectement via un appel de procédure ALPC . De plus, de nombreux services du noyau utilisent ALPC pour communiquer avec les processus clients.

3.7.4 Tuyaux

Un tuyau agit comme un conduit permettant à deux processus de communiquer. Les tuyaux ont été l'un des premiers mécanismes IPC des premiers systèmes UNIX . Ils constituent généralement l'un des moyens les plus simples permettant aux processus de communiquer entre eux, bien qu'ils présentent également certaines limites. Lors de la mise en œuvre d'un pipeline, quatre problèmes doivent être pris en compte :

1. Le canal permet-il une communication bidirectionnelle ou la communication est-elle unidirectionnelle ?

2. Si la communication bidirectionnelle est autorisée, est-elle en semi-duplex (les données ne peuvent circuler que dans un sens à la fois) ou en duplex intégral (les données peuvent circuler dans les deux sens en même temps) ?

3. Une relation (telle que parent-enfant) doit- elle exister entre la communauté ?
   processus de restauration ?

4. Les canalisations peuvent-elles communiquer sur un réseau ou les canalisations communicantes doivent-elles les processus résident sur la même machine ?

Dans les sections suivantes, nous explorons deux types courants de canaux utilisés sur les systèmes UNIX et Windows : les canaux ordinaires et les canaux nommés.

3.7.4.1 Pipes ordinaires Les pipes

ordinaires permettent à deux processus de communiquer de manière standard producteur-consommateur : le producteur écrit à une extrémité du tube (l' extrémité d'écriture) et le consommateur lit à l'autre extrémité (l' extrémité de lecture). En conséquence, les canalisations ordinaires sont unidirectionnelles, permettant une communication à sens unique. Si une communication bidirectionnelle est requise, deux canaux doivent être utilisés, chaque canal envoyant des données dans une direction différente. Nous illustrons ensuite la construction de canaux ordinaires sur les systèmes UNIX et Windows. Dans les deux exemples de programme, un processus écrit le message Salutations dans le canal, tandis que l'autre processus lit ce message dans le canal.

Sur les systèmes UNIX , les tubes ordinaires sont construits à l'aide de la fonction

tuyau(int fd[])

Cette fonction crée un canal accessible via les descripteurs de fichier int fd[] : fd[0] est la fin de lecture du canal et fd[1] est la fin d'écriture. UNIX traite un tube comme un type spécial de fichier. Ainsi, les canaux sont accessibles à l'aide des appels système read() et write() ordinaires .

Un canal ordinaire n'est pas accessible depuis l'extérieur du processus qui l'a créé. Généralement, un processus parent crée un canal et l'utilise pour communiquer avec un processus enfant qu'il crée via fork(). Rappelez-vous de la section 3.3.1 qu'un processus enfant hérite des fichiers ouverts de son parent. Puisqu'un tube est un type de fichier spécial, l'enfant hérite du tube de son processus parent. La figure 3.20 illustre

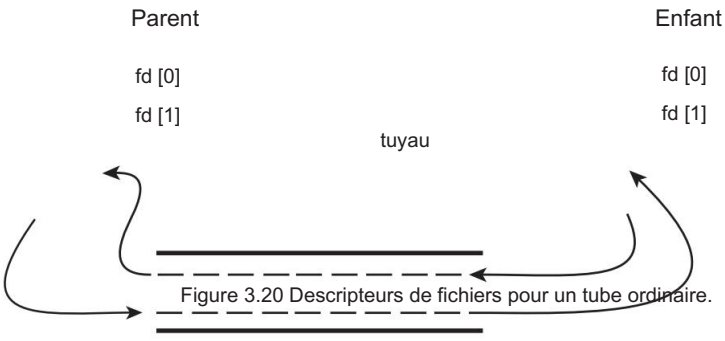


Figure 3.20 Descripteurs de fichiers pour un tube ordinaire.

```
#include <sys/types.h> #include
<stdio.h> #include
<string.h> #include
<unistd.h>

#define TAILLE DU TAMPON
25 #define READ END
0 #define WRITE END 1

int principal (vide) {

    char write msg[BUFFER SIZE] = "Salutations"; char read
    msg[TAILLE DU BUFFER] ; int fd[2];
    pid t pid;

    /* Le programme continue dans la figure 3.22 */
```

Figure 3.21 Tuyau ordinaire sous UNIX.

la relation entre les descripteurs de fichiers du tableau fd et les processus parent et enfant. Comme ceci l'illustre, toute écriture effectuée par le parent vers son extrémité d'écriture du tube (fd[1]) peut être lue par l'enfant à partir de son extrémité de lecture (fd[0]) dans le tube.

Dans le programme UNIX illustré à la figure 3.21, le processus parent crée un tube puis envoie un appel fork() créant le processus enfant. Ce qui se passe après l' appel à fork() dépend de la manière dont les données doivent circuler dans le canal.
Dans ce cas, le parent écrit dans le tube et l'enfant le lit. Il est important de noter que le processus parent et le processus enfant ferment initialement leurs extrémités inutilisées du tube. Bien que le programme illustré à la figure 3.21 ne nécessite pas cette action, il s'agit d'une étape importante pour garantir qu'un processus lisant à partir du tube puisse détecter la fin du fichier (read() renvoie 0) lorsque l'enregistreur a fermé la fin du fichier. tuyau.

Les canaux ordinaires sur les systèmes Windows sont appelés canaux anonymes et ils se comportent de la même manière que leurs homologues UNIX : ils sont unidirectionnels et emploient des relations parent-enfant entre les processus de communication. De plus, la lecture et l'écriture dans le tube peuvent être effectuées avec les fonctions ordinaires ReadFile() et WriteFile() . L' API Windows pour créer des tuyaux est la fonction CreatePipe() , à laquelle quatre paramètres sont transmis. Les paramètres fournissent des handles séparés pour (1) la lecture et (2) l'écriture dans le tube, ainsi que (3) une instance de la structure STARTUPINFO , qui est utilisée pour spécifier que le processus enfant doit hériter des handles du tube. De plus, (4) la taille du canal (en octets) peut être spécifiée.

La figure 3.23 illustre un processus parent créant un canal anonyme pour communiquer avec son enfant. Contrairement aux systèmes UNIX , dans lesquels un processus enfant hérite automatiquement d'un canal créé par son parent, Windows exige que le programmeur spécifie les attributs dont le processus enfant héritera. C'est

```
    /* crée le canal */ if (pipe(fd)
    == -1) { fprintf(stderr,"Pipe
        failed"); renvoyer 1 ;

    }

    /* fork un processus enfant */ pid =
    fork();

    if (pid < 0) { /* une erreur s'est produite */
        fprintf(stderr, "Fork Failed"); renvoyer 1 ;

    }

    if (pid > 0) { /* processus parent */
        /* ferme l'extrémité inutilisée du tuyau */
        close(fd[READ END]);

        /* écrire dans le tube */
        write(fd[WRITE END], write msg, strlen(write msg)+1);

        /* ferme la fin d'écriture du tube */ close(fd[WRITE
        END]);

    } else { /* processus enfant */ /*
        ferme l'extrémité inutilisée du tube */ close(fd[WRITE
        END]);

        /* lecture à partir du tube */
        read(fd[READ END], read msg, BUFFER SIZE);
        printf("lire %s",lire msg);

        /* ferme la fin de lecture du tube */ close(fd[READ
        END]);
    }

    renvoie 0 ;
}
```

Figure 3.22 Figure 3.21, suite.

accompli en initialisant d'abord la structure SECURITY ATTRIBUTES pour permettre l'héritage des handles, puis en redirigeant les handles du processus enfant pour l'entrée standard ou la sortie standard vers le handle de lecture ou d'écriture du tube. Puisque l'enfant lira à partir du tube, le parent doit rediriger l'entrée standard de l'enfant vers le handle de lecture du tube. De plus, comme les tubes sont en semi-duplex, il faut interdire au fils d'hériter de la fin d'écriture du

```
#include <stdio.h> #include
<stdlib.h> #include
<windows.h>

#define TAILLE DU TAMPON 25

int principal (VOID) {

    HANDLE ReadHandle, WriteHandle ;
    INFO DEMARRAGE si ;
    INFORMATIONS SUR LE
    PROCESSUS pi ; char message[BUFFER SIZE] = "Salutations";
    DWORD écrit ;

    /* Le programme continue dans la figure 3.24 */
```

Figure 3.23 Canal anonyme Windows : processus parent.

tuyau. Le programme pour créer le processus enfant est similaire au programme de la figure 3.10, sauf que le cinquième paramètre est défini sur TRUE, indiquant que le processus enfant doit hériter des handles désignés de son parent. Avant d'écrire dans le tube, le parent ferme d'abord l'extrémité de lecture inutilisée du tube. Le processus enfant qui lit à partir du tube est illustré à la figure 3.25. Avant de lire à partir du tube, ce programme obtient le handle de lecture du tube en appelant GetStdHandle().

Notez que les canaux ordinaires nécessitent une relation parent-enfant entre les processus de communication sur les systèmes UNIX et Windows. Cela signifie que ces canaux ne peuvent être utilisés que pour la communication entre les processus sur la même machine.

3.7.4.2 Tubes nommés Les

tubes ordinaires fournissent un mécanisme simple permettant à deux processus de communiquer. Cependant, les canalisations ordinaires n'existent que lorsque les processus communiquent entre eux. Sur les systèmes UNIX et Windows, une fois que les processus ont fini de communiquer et se sont terminés, le canal ordinaire cesse d'exister.

Les canaux nommés fournissent un outil de communication beaucoup plus puissant. La communication peut être bidirectionnelle et aucune relation parent-enfant n'est requise. Une fois qu'un canal nommé est établi, plusieurs processus peuvent l'utiliser pour la communication. En fait, dans un scénario typique, un canal nommé a plusieurs rédacteurs. De plus, les canaux nommés continuent d'exister une fois les processus de communication terminés. Les systèmes UNIX et Windows prennent en charge les canaux nommés, bien que les détails d'implémentation diffèrent considérablement. Ensuite, nous explorons les canaux nommés dans chacun de ces systèmes.

Les canaux nommés sont appelés FIFO dans les systèmes UNIX . Une fois créés, ils apparaissent comme des fichiers typiques dans le système de fichiers. Un FIFO est créé avec l' appel système mkfifo() et manipulé avec les appels système ordinaires open(), read(), write() et close() . Il continuera d'exister jusqu'à ce qu'il soit explicitement supprimé

```
/* configure les attributs de sécurité permettant l'héritage des pipes */
ATTRIBUTS DE SÉCURITÉ sa = {sizeof(ATTRIBUTS DE SÉCURITÉ),NULL,TRUE}; /*
alloue de la mémoire */
ZeroMemory(&pi, taillede(pi));

/* crée le canal */ if (!
CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) { fprintf(stderr, "Create
    Pipe Failed"); renvoyer 1 ;

}

/* établit la structure START INFO pour le processus enfant */ GetStartupInfo(&si);
si.hStdOutput = GetStdHandle
(POIGNÉE DE SORTIE STD);

/* redirige l'entrée standard vers l'extrémité de lecture du tube */ si.hStdInput =
ReadHandle; si.dwFlags = STARTF
USESTDHANDLES ;

/* ne permet pas à l'enfant d'hériter de la fin d'écriture du tube */
SetHandleInformation (WriteHandle, HANDLE FLAG INHERIT, 0);

/* crée le processus enfant */
CreateProcess(NULL, "child.exe", NULL, NULL, TRUE, /* hérite
  des handles */ 0, NULL, NULL, &si,
  &pi);

/* ferme l'extrémité inutilisée du tuyau */
FermerHandle(ReadHandle);

/* le parent écrit dans le tube */ if (!
WriteFile(WriteHandle, message,BUFFER SIZE,&writing,NULL))
    fprintf(stderr, "Erreur d'écriture dans le tube.");

/* ferme la fin d'écriture du tube */
FermerHandle(WriteHandle);

/* attend que l'enfant quitte */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread); renvoie
0 ; }
```

Figure 3.24 Figure 3.23, suite.

```
#include <stdio.h> #include
<windows.h>

#define TAILLE DU TAMPON 25

int principal (VOID) {
                _
POIGNÉE Lecture de la poignée ;
Tampon CHAR[TAILLE DU TAMPON] ;
DWORD lu ;

        /* récupère le handle de lecture du tube */
        ReadHandle = GetStdHandle (POIGNÉE D'ENTRÉE STD);

        /* l'enfant lit à partir du tube */ if (ReadFile(ReadHandle,
        buffer, BUFFER SIZE, &read, NULL))          _
             printf("enfant lu %s",buffer); else fprintf(stderr,

        "Erreur de lecture depuis le tube");                    _

        renvoie 0 ;
}
```

Figure 3.25 Canaux anonymes Windows : processus enfant.

du système de fichiers. Bien que les FIFO permettent une communication bidirectionnelle, seule la transmission semi-duplex est autorisée. Si les données doivent voyager dans les deux sens, deux FIFO sont généralement utilisées. De plus, les processus communicants doivent résider sur la même machine. Si une communication intermachine est requise, des prises (Section 3.8.1) doivent être utilisées.

Les canaux nommés sur les systèmes Windows offrent un mécanisme de communication plus riche que leurs homologues UNIX . La communication en duplex intégral est autorisée et les processus de communication peuvent résider sur la même machine ou sur des machines différentes. De plus, seules les données orientées octets peuvent être transmises via une FIFO UNIX, alors que les systèmes Windows autorisent les données orientées octets ou message. Les canaux nommés sont créés avec la fonction CreateNamedPipe() et un client peut se connecter à un canal nommé à l'aide de ConnectNamedPipe(). La communication via le canal nommé peut être réalisée à l'aide des fonctions ReadFile() et WriteFile() .

3.8 Communication dans les systèmes client-serveur

Dans la section 3.4, nous avons décrit comment les processus peuvent communiquer en utilisant la mémoire partagée et la transmission de messages. Ces techniques peuvent également être utilisées pour la communication dans les systèmes client-serveur (Section 1.10.3). Dans cette section, nous explorons deux autres stratégies de communication dans les systèmes client-serveur : les sockets et

LES TUYAUX EN PRATIQUE

Les tuyaux sont assez souvent utilisés dans l' environnement de ligne de commande UNIX pour les situations dans lesquelles la sortie d'une commande sert d'entrée à une autre. Par exemple, la commande UNIX ls produit une liste de répertoires. Pour les listes de répertoires particulièrement longues, la sortie peut défiler sur plusieurs écrans. La commande less gère la sortie en affichant un seul écran de sortie à la fois où l'utilisateur peut utiliser certaines touches pour avancer ou reculer dans le fichier. La configuration d'un canal entre les commandes ls et less (qui s'exécutent en tant que processus individuels) permet à la sortie de ls d'être transmise en entrée à less, permettant à l'utilisateur d'afficher un grand répertoire répertoriant un écran à la fois. Un tube peut être construit sur la ligne de commande en utilisant le | personnage. La commande complète est

ls | moins

Dans ce scénario, la commande ls sert de producteur et sa sortie est consommée par la commande less .

Les systèmes Windows fournissent davantage de commandes pour le shell DOS avec des fonctionnalités similaires à celles de son homologue UNIX . (Les systèmes UNIX fournissent également une commande more , mais dans le style ironique courant sous UNIX, la commande less fournit en fait plus de fonctionnalités que plus !) Le shell DOS utilise également le | caractère pour établir un tuyau. La seule différence est que pour obtenir une liste de répertoires, DOS utilise la commande dir plutôt que ls, comme indiqué ci-dessous :

rép | plus

appels de procédure à distance (RPC). Comme nous le verrons dans notre étude des RPC, non seulement ils sont utiles pour l'informatique client-serveur, mais Android utilise également des procédures distantes comme forme d' IPC entre les processus exécutés sur le même système.

3.8.1 Prises

Asocket est défini comme un point final pour la communication. Une paire de processus communiquant sur un réseau utilise une paire de sockets, une pour chaque processus.
Un socket est identifié par une adresse IP concaténée avec un numéro de port. En général, les sockets utilisent une architecture client-serveur. Le serveur attend les demandes client entrantes en écoutant un port spécifié. Une fois la demande reçue, le serveur accepte une connexion du socket client pour terminer la connexion.
Les serveurs implémentant des services spécifiques (tels que SSH, FTP et HTTP) écoutent des ports connus (un serveur SSH écoute le port 22 ; un serveur FTP écoute le port 21 ; et un serveur Web ou HTTP écoute le port 80) . Tous les ports inférieurs à 1024 sont considérés comme bien connus et sont utilisés pour mettre en œuvre des services standard.

Lorsqu'un processus client lance une demande de connexion, un port lui est attribué par son ordinateur hôte. Ce port a un nombre arbitraire supérieur à 1024. Par exemple, si un client sur l'hôte X avec l'adresse IP 146.86.5.20 souhaite établir une connexion avec un serveur Web (qui écoute sur le port 80) à

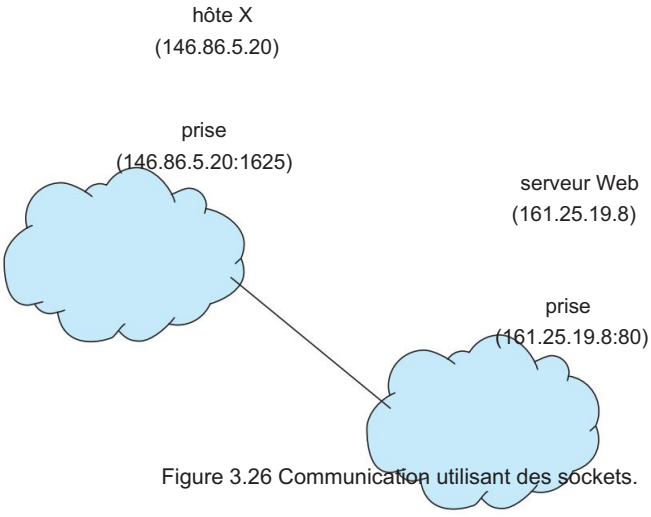


Figure 3.26 Communication utilisant des sockets.

Adresse 161.25.19.8, l'hôte X peut se voir attribuer le port 1625. La connexion consistera en une paire de sockets : (146.86.5.20:1625) sur l'hôte X et (161.25.19.8:80) sur le serveur Web. Cette situation est illustrée dans la figure 3.26. Les paquets circulant entre les hôtes sont transmis au processus approprié en fonction du numéro de port de destination.

Toutes les connexions doivent être uniques. Par conséquent, si un autre processus également sur l'hôte X souhaitait établir une autre connexion avec le même serveur Web, il se verrait attribuer un numéro de port supérieur à 1024 et non égal à 1625. Cela garantit que toutes les connexions sont constituées d'une paire unique de sockets.

Bien que la plupart des exemples de programmes dans ce texte utilisent le C, nous illustrerons les sockets en utilisant Java, car il fournit une interface beaucoup plus simple avec les sockets et possède une riche bibliothèque pour les utilitaires réseau. Les personnes intéressées par la programmation socket en C ou C++ doivent consulter les notes bibliographiques en

Java propose trois types différents de sockets. Les sockets orientés connexion (TCP) sont implémentés avec la classe Socket . Les sockets sans connexion (UDP) utilisent la classe DatagramSocket . Enfin, la classe MulticastSocket est une sous-classe de la classe DatagramSocket . Un socket multicast permet d'envoyer des données à plusieurs destinataires.

Notre exemple décrit un serveur de données qui utilise des sockets TCP orientés connexion . L'opération permet aux clients de demander la date et l'heure actuelles au serveur. Le serveur écoute le port 6013, bien que le port puisse avoir n'importe quel numéro arbitraire et inutilisé supérieur à 1024. Lorsqu'une connexion est reçue, le serveur renvoie la date et l'heure au client.

Le serveur de données est illustré à la figure 3.27. Le serveur crée un ServerSocket qui spécifie qu'il écoutera le port 6013. Le serveur commence alors à écouter le port avec la méthode accept() . Le serveur bloque la méthode accept() en attendant qu'un client demande une connexion. Lorsqu'une demande de connexion est reçue, accept() renvoie un socket que le serveur peut utiliser pour communiquer avec le client.

Les détails de la façon dont le serveur communique avec le socket sont les suivants.

Le serveur établit d'abord un objet PrintWriter qu'il utilisera pour communiquer avec le client. Un objet PrintWriter permet au serveur d'écrire sur le socket en utilisant les méthodes de routine print() et println() pour la sortie. Le

```
importer java.net.* ;
importer java.io.* ;

classe publique DateServer {

    public static void main (String[] arguments) {
        essayer {
            Chaussette ServerSocket = new ServerSocket (6013);

            /* écoute maintenant les connexions */
            while (true)
                { Socket client = sock.accept();

                PrintWriter pout = new
                  PrintWriter(client.getOutputStream(), true);

                /* écrit la date sur le socket */ pout.println(new
                java.util.Date().toString());

                /* ferme le socket et reprend */ /* l'écoute
                des connexions */ client.close();

            }

        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Figure 3.27 Serveur de données.

Le processus serveur envoie la date au client, en appelant la méthode println(). Une fois qu'il a écrit la date sur le socket, le serveur ferme le socket pour le client et recommence à écouter d'autres requêtes.

Un client communique avec le serveur en créant un socket et en se connectant au port sur lequel le serveur écoute. Nous implémentons un tel client dans le programme Java illustré à la figure 3.28. Le client crée un Socket et demande une connexion avec le serveur à l'adresse IP 127.0.0.1 sur le port 6013. Une fois la connexion établie, le client peut lire à partir du socket en utilisant les instructions d'E/S de flux normales . Après avoir reçu la date du serveur, le client ferme le socket et quitte. L' adresse IP 127.0.0.1 est une adresse IP spéciale connue sous le nom de bouclage. Lorsqu'un ordinateur fait référence à l'adresse IP 127.0.0.1, il fait référence à lui Ce mécanisme permet à un client et un serveur sur le même hôte de communiquer en utilisant le protocole TCP/IP . L' adresse IP 127.0.0.1 pourrait être remplacée par l' adresse IP d'un autre hôte exécutant le serveur de données. En plus d'une adresse IP , un nom d'hôte réel, tel que www.westminstercollege.edu, peut également être utilisé.

```
importer java.net.* ;
importer java.io.* ;

classe publique DateClient {

    public static void main (String[] arguments) {
        try { /*
            établir une connexion au socket du serveur */ Socket
            sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            Bac BufferedReader = nouveau
                BufferedReader (nouveau InputStreamReader (dans));

            /* lit la date depuis le socket */ String line; while
            ( (line =
            bin.readLine()) != null)
                System.out.println(ligne);

            /* ferme la connexion socket*/ sock.close();


        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Figure 3.28 Client de données.

La communication utilisant des sockets, bien que courante et efficace, est considérée comme une forme de communication de bas niveau entre des processus distrib

L'une des raisons est que les sockets permettent uniquement l'échange d'un flux d'octets non structuré entre les threads communicants. Il est de la responsabilité de l'application client ou serveur d'imposer une structure aux données. Dans la sous-section suivante, nous examinons une méthode de communication de niveau supérieur : les appels de procédure à distance (RPC).

3.8.2 Appels de procédure à distance

L'une des formes les plus courantes de service à distance est le paradigme RPC , conçu comme un moyen d'abstraire le mécanisme d'appel de procédure à utiliser entre des systèmes dotés de connexions réseau. Il est similaire à bien des égards au mécanisme IPC décrit à la section 3.4, et il est généralement construit au-dessus d'un tel système. Cependant, ici, comme nous avons affaire à un environnement dans lequel les processus s'exécutent sur des systèmes distincts, nous devons utiliser un schéma de communication basé sur les messages pour fournir un service à distance.

Contrairement aux messages IPC , les messages échangés dans la communication RPC sont bien structurés et ne sont donc plus de simples paquets de données. Chaque message est adressé à un démon RPC écoutant un port du système distant, et chacun contient un identifiant spécifiant la fonction à exécuter et les paramètres à transmettre à cette fonction. La fonction est ensuite exécutée comme demandé et toute sortie est renvoyée au demandeur dans un message séparé.

Un port dans ce contexte est simplement un numéro inclus au début d'un paquet de messages. Alors qu'un système possède normalement une adresse réseau, il peut avoir plusieurs ports au sein de cette adresse pour différencier les nombreux services réseau qu'il prend en charge. Si un processus distant a besoin d'un service, il adresse un message au port approprié. Par exemple, si un système souhaitait permettre à d'autres systèmes de répertorier ses utilisateurs actuels, il aurait un démon prenant en charge un tel RPC attaché à un port, par exemple le port 3027. N'importe quel système distant pourrait obtenir les informations nécessaires (c'est-à-dire , la liste des utilisateurs actuels) en envoyant un message RPC au port 3027 du serveur. Les données seraient reçues dans un message de réponse.

La sémantique des RPC permet à un client d'invoquer une procédure sur un hôte distant comme s'il invoquait une procédure localement. Le système RPC masque les détails qui permettent la communication en fournissant un stub côté client.
En règle générale, un stub distinct existe pour chaque procédure distante distincte. Lorsque le client appelle une procédure distante, le système RPC appelle le stub approprié en lui transmettant les paramètres fournis à la procédure distante. Ce stub localise le port sur le serveur et rassemble les paramètres. Le stub transmet ensuite un message au serveur en utilisant la transmission de messages. Un stub similaire côté serveur reçoit ce message et appelle la procédure sur le serveur. Si nécessaire, les valeurs de retour sont renvoyées au client en utilisant la même technique.
Sur les systèmes Windows, le code stub est compilé à partir d'une spécification écrite dans le langage MIDL (Microsoft Interface Definitio Language), qui est utilisé pour définir les interfaces entre les programmes client et serveur.

Le marshaling des paramètres résout le problème concernant les différences de représentation des données sur les machines client et serveur. Considérons la représentation d'entiers de 32 bits. Certains systèmes (appelés big-endian) stockent d'abord l'octet de poids fort, tandis que d'autres systèmes (appelés small-endian) stockent d'abord l'octet de poids faible. Aucun des deux ordres n'est « meilleur » en soi ; le choix est plutôt arbitraire au sein d'une architecture informatique. Pour résoudre de telles différences, de nombreux systèmes RPC définissent une représentation des données indépendante de la machine. Une de ces représentations est connue sous le nom de représentation de données externes (XDR). Côté client, le marshaling des paramètres implique la conversion des données dépendantes de la machine en XDR avant qu'elles ne soient envoyées au serveur. Côté serveur, les données XDR ne sont pas marshalées et converties en représentation dépendante de la machine pour le serveur.

Une autre question importante concerne la sémantique d'un appel. Alors que les appels de procédure locale échouent uniquement dans des circonstances extrêmes, les RPC peuvent échouer ou être dupliqués et exécutés plusieurs fois, en raison d'erreurs réseau courantes. Une façon de résoudre ce problème consiste pour le système d'exploitation à garantir que les messages sont traités exactement une fois, plutôt qu'au maximum une fois. La plupart des appels de procédure locale ont la fonctionnalité « exactement une fois », mais elle est plus difficile à mettre en œuvre.

Tout d'abord, considérons « au plus une fois ». Cette sémantique peut être implémentée en attachant un horodatage à chaque message. Le serveur doit conserver un historique de tous les horodatages des messages qu'il a déjà traités ou un historique suffisamment volumineux

pour garantir que les messages répétés sont détectés. Les messages entrants qui ont déjà un horodatage dans l'historique sont ignorés. Le client peut alors envoyer un message une ou plusieurs fois et être assuré qu'il ne s'exécutera qu'une seule fois.

Pour « exactement une fois », nous devons supprimer le risque que le serveur ne reçoive jamais la requête. Pour ce faire, le serveur doit implémenter le protocole « au plus une fois » décrit ci-dessus mais doit également accuser réception au client que l' appel RPC a été reçu et exécuté. Ces messages ACK sont courants dans tout le réseau. Le client doit renvoyer périodiquement chaque appel RPC jusqu'à ce qu'il reçoive l' ACK pour cet appel.

Un autre problème important concerne la communication entre un serveur et un client. Avec les appels de procédure standard, une certaine forme de liaison a lieu pendant le temps de liaison, de chargement ou d'exécution (Chapitre 9) afin que le nom d'un appel de procédure soit remplacé par l'adresse mémoire de l'appel de procédure. Le schéma RPC nécessite une liaison similaire entre le client et le port du serveur, mais comment un client connaît-il les numéros de port sur le serveur ? Aucun des deux systèmes ne dispose d'informations complètes sur l'autre, car ils ne partagent pas de mémoire.

Deux approches sont courantes. Premièrement, les informations de liaison peuvent être prédéterminées, sous la forme d'adresses de port fixes. Au moment de la compilation, un appel RPC est associé à un numéro de port fixe. Une fois un programme compilé, le serveur ne peut pas modifier le numéro de port du service demandé. Deuxièmement, la liaison peut être effectuée dynamiquement par un mécanisme de rendez-vous. En règle générale, un système d'exploitation fournit un démon de rendez-vous (également appelé matchmaker) sur un port RPC fixe .
Un client envoie ensuite un message contenant le nom du RPC au démon de rendez-vous demandant l'adresse du port du RPC qu'il doit exécuter. Le numéro de port est renvoyé et les appels RPC peuvent être envoyés à ce port jusqu'à la fin du processus (ou jusqu'à ce que le serveur plante). Cette méthode nécessite la surcharge supplémentaire de la requête initiale mais est plus flexible que la première approche. La figure 3.29 montre un exemple d'interaction.

Le schéma RPC est utile pour implémenter un système de fichiers distribué (Chapitre 19). Un tel système peut être implémenté sous la forme d'un ensemble de démons et de clients RPC . Les messages sont adressés au port du système de fichiers distribué sur un serveur sur lequel une opération sur les fichiers doit avoir lieu. Le message contient l'opération de disque à effectuer. L'opération sur le disque peut être read(), write(), rename(), delete() ou status(), correspondant aux appels système habituels liés aux fich
Le message de retour contient toutes les données résultant de cet appel, qui est exécuté par le démon DFS au nom du client. Par exemple, un message peut contenir une demande de transfert d'un fichier entier vers un client ou se limiter à une simple demande de blocage. Dans ce dernier cas, plusieurs requêtes peuvent être nécessaires pour transférer un dossier entier.

3.8.2.1 RPC Android

Bien que les RPC soient généralement associés à l'informatique client-serveur dans un système distribué, ils peuvent également être utilisés comme une forme d' IPC entre des processus exécutés sur le même système. Le système d'exploitation Android dispose d'un riche ensemble de mécanismes IPC contenus dans son cadre de liaison , y compris les RPC qui permettent à un processus de demander des services à un autre processus.

Android définit un composant d'application comme un élément de base qui fournit une utilité à une application Android, et une application peut combiner plusieurs composants d'application pour fournir des fonctionnalités à une application. Une telle application-

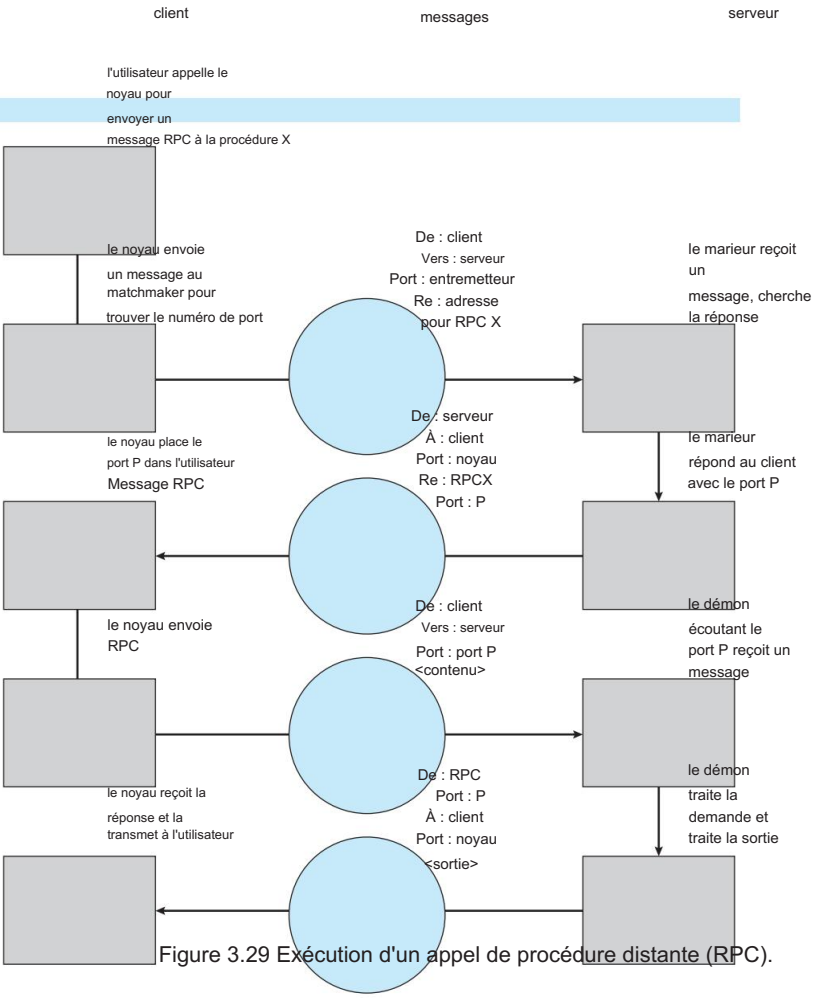


Figure 3.29 Exécution d'un appel de procédure distante (RPC).

Le composant de tion est un service qui n'a pas d'interface utilisateur mais qui s'exécute en arrière-plan lors de l'exécution d'opérations de longue durée ou de l'exécution de travaux pour des processus distants. Des exemples de services incluent la lecture de musique en arrière-plan et la récupération de données via une connexion réseau pour le compte d'un autre processus, empêchant ainsi l'autre processus de se bloquer pendant le téléchargement des données. Lorsqu'une application client appelle la méthode bindService() d'un service, ce service est « lié » et disponible pour fournir une communication client-serveur à l'aide de la transmission de messages ou de RPC.

Un service lié doit étendre la classe Android Service et doit implémenter la méthode onBind(), qui est invoquée lorsqu'un client appelle bindSer-vice(). Dans le cas du passage de messages, la méthode onBind() renvoie un service Mes-senger , qui est utilisé pour envoyer des messages du client au service. Le service Messenger est à sens unique ; si le service doit renvoyer une réponse au client, celui-ci doit également fournir un service Messenger , qui est contenu dans le champ ReplyTo de l' objet Message envoyé au service. Le service peut ensuite renvoyer des messages au client.

Pour fournir des RPC, la méthode onBind() doit renvoyer une interface représentant les méthodes de l'objet distant que les clients utilisent pour interagir avec l'objet distant.

service. Cette interface est écrite dans la syntaxe Java standard et utilise le langage de définition d'interface Android (AIDL) pour créer des fichiers stub, qui servent d'interface client aux services distants.

Ici, nous décrivons brièvement le processus requis pour fournir un service distant générique nommé remoteMethod() à l'aide d'AIDL et du service de liaison. L'interface du service distant apparaît comme suit :

```
/* RemoteService.aidl */
interface RemoteService
{
    boolean remoteMethod(int x, double y);
{
```

Ce fichier est écrit sous le nom RemoteService.aidl. Le kit de développement Android l'utilisera pour générer une interface .java à partir du fichier .aidl , ainsi qu'un stub qui sert d' interface RPC pour ce service. Le serveur doit implémenter l'interface générée par le fichier .aidl , et l'implémentation de cette interface sera appelée lorsque le client invoquera remoteMethod().

Lorsqu'un client appelle bindService(), la méthode onBind() est invoquée sur le serveur et renvoie le stub de l' objet RemoteService au client.
Le client peut alors appeler la méthode distante comme suit :

```
Service de service à distance ;
    ...
service.remoteMethod(3, 0.14);
```

En interne, le framework de classeur Android gère le marshaling des paramètres, le transfert des paramètres marshalés entre les processus et l'invocation de l'implémentation nécessaire du service, ainsi que le renvoi des valeurs de retour au processus client.

3.9 Résumé

- Un processus est un programme en exécution, et l'état de l'activité actuelle d'un processus est représenté par le compteur de programme, ainsi que par d'autres registres.
- L'agencement d'un processus en mémoire est représenté par quatre sections différentes : (1) texte, (2) données, (3) tas et (4) pile.
- Au fur et à mesure qu'un processus s'exécute, il change d'état. Il existe quatre états généraux d'un processus : (1) prêt, (2) en cours d'exécution, (3) en attente et (4) terminé.
- Un bloc de contrôle de processus (PCB) est la structure de données du noyau qui représente un processus dans un système d'exploitation.
- Le rôle du planificateur de processus est de sélectionner un processus disponible sur lequel s'exécuter. un processeur.
- Un système d'exploitation effectue un changement de contexte lorsqu'il passe de exécuter un processus pour en exécuter un autre.

- Les appels système fork() et CreateProcess() sont utilisés pour créer des processus sur les systèmes UNIX et Windows, respectivement.
- Lorsque la mémoire partagée est utilisée pour la communication entre processus, deux processus (ou plus) partagent la même région de mémoire. POSIX fournit une API pour la mémoire partagée.
- Deux processus peuvent communiquer en échangeant des messages entre eux en utilisant la transmission de messages. Le système d'exploitation Mach utilise la transmission de messages comme principale forme de communication interprocessus. Windows fournit également une forme de transmission de messages.
- Un canal fournit un canal permettant à deux processus de communiquer. Il existe deux formes de tuyaux, ordinaires et nommés. Les tuyaux ordinaires sont conçus pour la communication entre les processus ayant une relation parent-enfant.
  Les tubes nommés sont plus généraux et permettent à plusieurs processus de communiquer.
- Les systèmes UNIX fournissent des tubes ordinaires via l' appel système pipe() .
  Les tubes ordinaires ont une fin de lecture et une fin d'écriture. Un processus parent peut, par exemple, envoyer des données au canal en utilisant sa fin d'écriture, et le processus enfant peut les lire à partir de sa fin de lecture. Les canaux nommés sous UNIX sont appelés FIFO.
- Les systèmes Windows fournissent également deux formes de canaux : les canaux anonymes et nommés. Les pipes anonymes sont similaires aux pipes ordinaires UNIX . Ils sont unidirectionnels et emploient des relations parent-enfant entre les processus communicants. Les canaux nommés offrent une forme de communication interprocessus plus riche que leur homologue UNIX , les FIFO.
- Deux formes courantes de communication client-serveur sont les sockets et les appels de procédure distante (RPC). Les sockets permettent à deux processus sur des machines différentes de communiquer sur un réseau. Les RPC résument le concept d'appels de fonction (procédure) de telle manière qu'une fonction peut être invoquée sur un autre processus pouvant résider sur un ordinateur distinct.
- Le système d'exploitation Android utilise les RPC comme forme de communication interprocessus. communication en utilisant son cadre de reliure.

Exercices pratiques

3.1 À l'aide du programme illustré à la figure 3.30, expliquez quel sera le résultat sur la LIGNE A.

3.2 Y compris le processus parent initial, combien de processus sont créés par le programme illustré à la figure 3.31 ?

3.3 Les versions originales du système d'exploitation mobile iOS d'Apple ne prévoyaient aucun moyen de traitement simultané. Discutez de trois complications majeures que le traitement simultané ajoute à un système d'exploitation.

3.4 Certains systèmes informatiques fournissent plusieurs jeux de registres. Décrivez ce qui se passe lorsqu'un changement de contexte se produit si le nouveau contexte est déjà

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

valeur int = 5 ;

int main()

{ pid t pid ;

    pid = fourchette();

    if (pid == 0) { /* processus enfant */ value +=
        15; renvoie 0 ;


    } else if (pid > 0) { /* processus parent */ wait(NULL);
        printf("PARENT :
        valeur = %d",valeur); /* LIGNE A */ return 0;

    }
}
```

Figure 3.30 Quelle sortie aura la ligne A ?

chargé dans l'un des ensembles de registres. Que se passe-t-il si le nouveau contexte est en mémoire plutôt que dans un ensemble de registres et que tous les ensembles de registres sont utilisés ?

3.5 Lorsqu'un processus crée un nouveau processus à l'aide de l' opération fork() , lequel des états suivants est partagé entre le processus parent et le processus enfant ?

un. Empiler

b. Tas c.

Segments de mémoire partagée

3.6 Considérez la sémantique « exactement une fois » par rapport au mécanisme RPC . L'algorithme d'implémentation de cette sémantique s'exécute-t-il correctement même si le message ACK renvoyé au client est perdu en raison d'un problème de réseau ? Décrivez la séquence des messages et discutez si « exactement une fois » est toujours conservé.

3.7 Supposons qu'un système distribué soit susceptible de tomber en panne de serveur. Quels mécanismes seraient nécessaires pour garantir la sémantique « exactement une fois » pour l'exécution des RPC ?

```
#include <stdio.h>
#include <unistd.h>

int main() {

    /* fork un processus enfant */ fork();

    /* fork un autre processus enfant */ fork();

    /* et fork un autre */ fork();

    renvoie 0 ;
}
```

Figure 3.31 Combien de processus sont créés ?

Lectures complémentaires

La création, la gestion et l'IPC de processus dans les systèmes UNIX et Windows, respectivement, sont abordés dans [Robbins et Robbins (2003)] et [Russinovich et al. (2017)]. [Love (2010)] couvre la prise en charge des processus dans le noyau Linux et [Hart (2005)] couvre en détail la programmation des systèmes Windows.

Une couverture du modèle multiprocessus utilisé dans Chrome de Google est disponible sur http://blog.chromium.org/2008/09/multi-process-architecture.html.

La transmission de messages pour les systèmes multicœurs est abordée dans [Holland et Seltzer (2011)]. [Levin (2013)] décrit le passage de messages dans le système Mach, en particulier en ce qui concerne macOS et iOS.

[Harold (2005)] couvre la programmation socket en Java. Des détails sur les RPC Android sont disponibles sur https://developer.android.com/guide/compo nents/aidl.html. [Hart (2005)] et [Robbins et Robbins (2003)] couvrent respectivement les tuyaux dans les systèmes Windows et UNIX .

Les directives pour le développement Android sont disponibles sur https://developer.and roid.com/guide/.

Bibliographie

[Harold (2005)] ER Harold, Programmation réseau Java, troisième édition, O'Reilly & Associés (2005).

[Hart (2005)] JM Hart, Programmation du système Windows, troisième édition, Addison-Wesley (2005).

[Holland et Seltzer (2011)] D. Holland et M. Seltzer, « Multicore OSes: Looking Forward from 1991, er, 2011 », Actes de la 13e conférence USENIX sur Hot sujets dans les systèmes d'exploitation (2011), pages 33-33.

[Lévin (2013)] J. Levin, Mac OS X et iOS internes au noyau d'Apple, Wiley (2013).

[Amour (2010)] R. Love, Développement du noyau Linux, troisième édition, développeur Bibliothèque (2010).

[Robbins et Robbins (2003)] K. Robbins et S. Robbins, Programmation des systèmes Unix : communication, concurrence et threads, deuxième édition, Prentice Salle (2003).

[Russinovitch et al. (2017)] M. Russinovich, DA Solomon et A. Ionescu, Win-Dows Internals – Partie 1, septième édition, Microsoft Press (2017).

Exercices du chapitre 3

3.8 Décrire les actions entreprises par un noyau pour changer de contexte entre les processe.

3.9 Construisez un arbre de processus similaire à celui de la figure 3.7. Pour obtenir des informations sur les processus pour le système UNIX ou Linux, utilisez la commande ps -ae Utilisez la commande man ps pour obtenir plus d'informations sur la commande ps . Le gestionnaire de tâches sur les systèmes Windows ne fournit pas l' ID du processus parent, mais l' outil de surveillance des processus , disponible sur tech-net.microsoft.com, fournit un outil d'arborescence des processus.

3.10 Expliquer le rôle du processus init (ou systemd) sous UNIX et Linux systèmes en ce qui concerne la terminaison du processus.

3.11 En incluant le processus parent initial, combien de processus sont créés par le programme illustré à la figure 3.32 ?

3.12 Expliquez les circonstances dans lesquelles la ligne de code marquée printf("LINE J") dans la figure 3.33 sera atteinte.

3.13 À l'aide du programme de la figure 3.34, identifiez les valeurs du pid aux lignes A, B, C et D. (Supposons que les pid réels du parent et de l'enfant sont respectivement 2600 et 2603.)

3.14 Donnez un exemple d'une situation dans laquelle les canalisations ordinaires conviennent mieux que les canalisations nommées et un exemple de situation dans laquelle les canalisations nommées conviennent mieux que les canalisations ordinaires.

3.15 Considérez le mécanisme RPC . Décrivez les conséquences indésirables qui pourraient découler du fait de ne pas appliquer la sémantique « au plus une fois » ou « exactement une fois ». Décrire les utilisations possibles d'un mécanisme qui n'a aucune de ces garanties.

3.16 À l'aide du programme illustré à la figure 3.35, expliquez quel sera le résultat aux lignes X et Y.

```
#include <stdio.h> #include
<unistd.h>

int main() {

        int je;

        pour (i = 0; i < 4; i++) fork();


        renvoie 0 ;
}
```

Figure 3.21 Combien de processus sont créés ?

3.17 Quels sont les avantages et les inconvénients de chacun des éléments suivants ? Considérez à la fois le niveau du système et celui du programmeur. un. Communication synchrone et asynchrone b. Mise en mémoire tampon automatique et explicite c. Envoyer par copie et envoyer par référence d. Messages de taille fixe et variable

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()

{ pid t pid ;

    /* fork un processus enfant */ pid =
    fork();

    if (pid < 0) { /* une erreur s'est produite */
        fprintf(stderr, "Fork Failed"); renvoyer 1 ;


    } else if (pid == 0) { /* processus enfant */ execlp("/bin/
        ls","ls",NULL); printf("LIGNE J");


    } else { /* processus parent */ /* le
        parent attendra que l'enfant termine */ wait(NULL); printf("Enfant
        terminé");

    }

    renvoie 0 ;
}
```

Figure 3.22 Quand la LIGNE J sera -t-elle atteinte ?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()

{ pid t pid, pid1;

    /* fork un processus enfant */ pid =
    fork();

    if (pid < 0) { /* une erreur s'est produite */
        fprintf(stderr, "Fork Failed"); renvoyer 1 ;


    } else if (pid == 0) { /* processus enfant */ pid1 =
        getpid(); printf("enfant :
        pid = %d",pid); /* A */ printf("child: pid1 = %d",pid1); /
        * B */

    } else { /* processus parent */ pid1 =
        getpid(); printf("parent :
        pid = %d",pid); /* C */ printf("parent: pid1 = %d",pid1); /
        * D */ attendre(NULL);

    }

    renvoie 0 ;
}
```

Figure 3.23 Quelles sont les valeurs pid ?

```
#include <sys/types.h> #include
<stdio.h> #include
<unistd.h>

#define TAILLE 5

nombres int[TAILLE] = {0,1,2,3,4} ;

int main()

{ int je;
pid t pid;

    pid = fourchette();

    if (pid == 0) { for (i =
        0; i < TAILLE; i++) {
            nums[i] *= -i;
            printf("ENFANT : %d ",nums[i]); /* LIGNE X */
        }

    } sinon if (pid > 0)
        { wait(NULL);
        pour (i = 0; i < TAILLE; i++)
            printf("PARENT : %d ",nums[i]); /* LIGNE Y */
    }

    renvoie 0 ;
}
```

Figure 3.24 Quelle sortie sera la ligne X et la ligne Y ?

Problèmes de programmation

3.18 À l'aide d'un système UNIX ou Linux, écrivez un programme C qui crée un processus enfant qui devient finalement un processus zombie. Ce processus zombie doit rester dans le système pendant au moins 10 secondes. Les états du processus peuvent être obtenus à partir de la commande

```
ps-l
```

Les états du processus sont affichés sous la colonne S ; les processus avec un état Z sont des zombies. L'identifiant de processus (pid) du processus enfant est répertorié dans la colonne PID et celui du parent est répertorié dans la colonne PPID .

Le moyen le plus simple de déterminer que le processus enfant est bien un zombie est peut-être d'exécuter le programme que vous avez écrit en arrière-plan (à l'aide du &), puis d'exécuter la commande ps -l pour déterminer si l'enfant est un processus zombie. Parce que vous ne voulez pas qu'il y ait trop de processus zombies dans le système, vous devrez supprimer celui que vous avez créé. Le moyen le plus simple de procéder consiste à terminer le processus parent à l'aide de la commande kill . Par exemple, si le pid du parent est 4884, vous saisirez

```
tuer -9 4884
```

3.19 Écrivez un programme C appelé time.c qui détermine le temps nécessaire pour exécuter une commande à partir de la ligne de commande. Ce programme sera exécuté sous le nom "./time <command>" et indiquera le temps écoulé pour exécuter la commande spécifiée. Cela impliquera d'utiliser les fonctions fork() et exec() , ainsi que la fonction gettimeofday() pour déterminer le temps écoulé. Cela nécessitera également l'utilisation de deux mécanismes IPC différents .

La stratégie générale consiste à créer un processus enfant qui exécutera la commande spécifiée. Cependant, avant que l'enfant n'exécute la commande, il enregistrera un horodatage de l'heure actuelle (que nous appelons « heure de début »). Le processus parent attendra la fin du processus enfant.
Une fois que l'enfant a terminé, le parent enregistrera l'horodatage actuel pour l'heure de fin. La différence entre les heures de début et de fin représente le temps écoulé pour exécuter la commande. L'exemple de sortie ci-dessous indique la durée d'exécution de la commande ls :

```
./heure ls
heure.c
heure

Temps écoulé : 0,25422
```

Comme le parent et l'enfant sont des processus distincts, ils devront organiser la façon dont l'heure de début sera partagée entre eux. Vous écrirez deux versions de ce programme, chacune représentant une méthode différente d' IPC.

La première version demandera au processus enfant d'écrire l'heure de début dans une région de la mémoire partagée avant d'appeler exec(). Une fois le processus enfant terminé, le parent lira l'heure de début dans la mémoire partagée.
Reportez-vous à la section 3.7.1 pour plus de détails sur l'utilisation de la mémoire partagée POSIX . Dans cette section, il existe des programmes distincts pour le producteur et le consommateur. Comme la solution à ce problème ne nécessite qu'un seul programme, la région de mémoire partagée peut être établie avant que le processus enfant ne soit bifurqué, permettant ainsi aux processus parent et enfant d'accéder à la région de mémoire partagée.

La deuxième version utilisera un tuyau. L'enfant écrira l'heure de début dans le tube et le parent la lira après la fin du processus enfant.

Vous utiliserez la fonction gettimeofday() pour enregistrer l'horodatage actuel. Cette fonction reçoit un pointeur vers un objet struct timeval , qui contient deux membres : tv sec et t usec. Ceux-ci représentent le nombre de secondes et de microsecondes écoulées depuis le 1er janvier 1970 (connu sous le nom d' ÉPOCH UNIX). L'exemple de code suivant illustre comment cette fonction peut être utilisée :

```
struct timeval courant ;

gettimeofday(&current,NULL);

// current.tv sec représente les secondes //
current.tv usec représente les microsecondes
```

Pour l'IPC entre les processus enfant et parent, le contenu du pointeur de mémoire partagée peut se voir attribuer la structure timeval représentant l'heure de début. Lorsque des tubes sont utilisés, un pointeur vers une structure timeval peut être écrit et lu depuis le tube.

3.20 Le gestionnaire pid d'un système d'exploitation est responsable de la gestion des identifiants de processus. Lorsqu'un processus est créé pour la première fois, le gestionnaire de pid lui attribue un pid unique. Le pid est renvoyé au gestionnaire de pid lorsque le processus termine son exécution, et le gestionnaire peut ultérieurement réaffecter ce pid. Les identifiants de processus sont abordés plus en détail à la section 3.3.1. Le plus important ici est de reconnaître que les identifiants de processus doivent être uniques ; deux processus actifs ne peuvent pas avoir le même pid.

Utilisez les constantes suivantes pour identifier la plage de valeurs pid possibles :

```
#définir MIN PID 300
#définir MAX PID 5000
```

Vous pouvez utiliser n'importe quelle structure de données de votre choix pour représenter la disponibilité des identifiants de processus. Une stratégie consiste à adopter ce que Linux a fait et à utiliser un bitmap dans lequel une valeur de 0 à la position i indi

un identifiant de processus de valeur i est disponible et une valeur de 1 indique que l'identifiant de processus est actuellement utilisé.

Implémentez l' API suivante pour obtenir et publier un pid :

- int allocate map(void) : crée et initialise une structure de données pour représenter les pids ; renvoie −1 en cas d'échec, 1 en cas de succès
- int allocate pid(void) : alloue et renvoie un pid ; renvoie −1 si impossible d'attribuer un pid (tous les pids sont utilisés)
- void release pid(int pid) : libère un pid

Ce problème de programmation sera modifié ultérieurement au chapitre 4 et au chapitre 6.

3.21 La conjecture de Collatz concerne ce qui se passe lorsque nous prenons un entier positif n et appliquons l'algorithme suivant :

$$n = \begin{cases} n/2, & \text{si n est pair} \\ 3 \times n + 1, & \text{si n est impair} \end{cases}$$

La conjecture stipule que lorsque cet algorithme est appliqué continuellement, tous les entiers positifs finiront par atteindre 1. Par exemple, si n = 35, la séquence est

35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Écrivez un programme C en utilisant l' appel système fork() qui génère cette séquence dans le processus enfant. Le numéro de départ sera fourni à partir de la ligne de commande. Par exemple, si 8 est passé en paramètre sur la ligne de commande, le processus enfant affichera 8, 4, 2, 1. Étant donné que les processus parent et enfant ont leurs propres copies des données, il sera nécessaire pour l'enfant de sortir la séquence. Demandez au parent d'invoquer l' appel wait() pour attendre la fin du processus enfant avant de quitter le programme. Effectuez la vérification des erreurs nécessaire pour vous assurer qu'un entier positif est transmis sur la ligne de commande.

3.22 Dans l'exercice 3.21, le processus enfant doit produire la séquence de nombres générée à partir de l'algorithme spécifié par la conjecture de Collatz car le parent et l'enfant ont leurs propres copies des données. Une autre approche pour concevoir ce programme consiste à établir un objet de mémoire partagée entre les processus parent et enfant. Cette technique permet à l'enfant d'écrire le contenu de la séquence dans l'objet de mémoire partagée. Le parent peut ensuite afficher la séquence lorsque l'enfant a terminé. La mémoire étant partagée, tous les changements apportés par l'enfant seront également reflétés dans le processus parent.

Ce programme sera structuré en utilisant la mémoire partagée POSIX comme décrit dans la section 3.7.1. Le processus parent progressera à travers les étapes suivantes :

un. Établissez l'objet de mémoire partagée (shm open(), ftruncate() et mmap()).

b. Créez le processus enfant et attendez qu'il se termine. c. Afficher le contenu de la mémoire partagée. d. Supprimez l'objet de mémoire partagée.

Un domaine de préoccupation concernant les processus de coopération concerne les problèmes de synchronisation. Dans cet exercice, les processus parent et enfant doivent être coordonnés afin que le parent ne génère pas la séquence tant que l'enfant n'a pas terminé l'exécution. Ces deux processus seront synchronisés à l'aide de l'appel système wait() : le processus parent invoquera wait(), ce qui le suspendra jusqu'à la sortie du processus enfant.

3.23 La section 3.8.1 décrit certains numéros de port comme étant bien connus, c'est-à-dire qu'ils fournissent des services standard. Le port 17 est connu comme le service de cotation du jour . Lorsqu'un client se connecte au port 17 sur un serveur, le serveur répond avec un devis pour ce jour.

Modifiez le serveur de date illustré dans la figure 3.27 afin qu'il fournisse une cotation du jour plutôt que la date actuelle. Les guillemets doivent être des caractères ASCII imprimables et doivent contenir moins de 512 caractères, bien que plusieurs lignes soient autorisées. Puisque ces ports bien connus sont réservés et donc indisponibles, demandez à votre serveur d'écouter le port 6017. Le client de date illustré dans la figure 3.28 peut être utilisé pour lire les guillemets renvoyés par votre serveur.

3.24 Ahaiku est un poème de trois vers dans lequel le premier vers contient cinq syllabes, le deuxième vers contient sept syllabes et le troisième vers contient cinq syllabes. Écrivez un serveur haïku qui écoute le port 5575. Lorsqu'un client se connecte à ce port, le serveur répond avec un haïku. Le client de date présenté dans la figure 3.28 peut être utilisé pour lire les citations renvoyées par votre serveur haïku.

3.25 Un serveur d'écho renvoie tout ce qu'il reçoit d'un client. Par exemple, si un client envoie au serveur la chaîne Bonjour là-bas !, le serveur répondra par Bonjour là-bas !

Écrivez un serveur d'écho à l'aide de l' API réseau Java décrite dans la section 3.8.1. Ce serveur attendra une connexion client en utilisant la méthode accept() . Lorsqu'une connexion client est reçue, le serveur effectue une boucle en effectuant les étapes suivantes :

- Lire les données du socket dans un tampon.
- Écrivez le contenu du tampon au client.

Le serveur ne sortira de la boucle que lorsqu'il aura déterminé que le client a fermé la connexion.

Le serveur de données de la figure 3.27 utilise la classe java.io.BufferedReader . BufferedReader étend la classe java.io.Reader , qui est utilisée pour lire les flux de caractères. Cependant, le serveur echo ne peut pas garantir qu'il lira les caractères des clients ; il peut également recevoir des données binaires. La classe java.io.InputStream traite les données au niveau des octets plutôt qu'au niveau des caractères. Ainsi, votre serveur d'écho doit utiliser un objet qui étend java.io.InputStream. La méthode read() dans le

La classe java.io.InputStream renvoie -1 lorsque le client a fermé l'extrémité de la connexion socket.

3.26 Concevoir un programme utilisant des canaux ordinaires dans lequel un processus envoie un message de chaîne à un deuxième processus, et le deuxième processus inverse la casse de chaque caractère du message et le renvoie au premier processus. Par exemple, si le premier processus envoie le message Hi There, le deuxième processus renverra hI tHERE. Cela nécessitera l'utilisation de deux canaux, l'un pour envoyer le message original du premier au deuxième processus et l'autre pour envoyer le message modifié du second au premier processus. Vous pouvez écrire ce programme en utilisant des canaux UNIX ou Windows.

3.27 Concevoir un programme de copie de fichiers nommé filecopy.c en utilisant des tubes ordinaires. Ce programme recevra deux paramètres : le nom du fichier à copier et le nom du fichier de destination. Le programme créera ensuite un tube ordinaire et écrira le contenu du fichier à copier dans le tube. Le processus enfant lira ce fichier à partir du tube et l'écrira dans le fichier de destination. Par exemple, si nous invoquons le programme comme suit :

```
./filecopy input.txt copie.txt
```

le fichier input.txt sera écrit dans le tube. Le processus enfant lira le contenu de ce fichier et l'écrira dans le fichier de destination copy.txt. Vous pouvez écrire ce programme en utilisant des canaux UNIX ou Windows.

Projets de programmation

Projet 1—UNIX Shell Ce projet

consiste à concevoir un programme C pour servir d'interface shell qui accepte les commandes utilisateur, puis exécute chaque commande dans un processus distinct. Votre implémentation prendra en charge la redirection d'entrée et de sortie, ainsi que les canaux comme forme d' IPC entre une paire de commandes. La réalisation de ce projet impliquera l'utilisation des appels système UNIX fork(), exec(), wait(), dup2() et pipe() et peut être réalisée sur n'importe quel système Linux, UNIX ou macOS .

I. Aperçu

Une interface shell donne à l'utilisateur une invite, après quoi la commande suivante est saisie. L'exemple ci-dessous illustre l'invite osh> et la commande suivante de l'utilisateur : cat prog.c. (Cette commande affiche le fichier prog.c sur le terminal à l'aide de la commande UNIX cat .)

```
osh>cat prog.c
```

Une technique pour implémenter une interface shell consiste à demander au processus parent de lire d'abord ce que l'utilisateur saisit sur la ligne de commande (dans ce cas, cat prog.c) , puis de créer un processus enfant distinct qui exécute la commande. Sauf indication contraire, le processus parent attend que l'enfant se termine avant de continuer. Cette fonctionnalité est similaire à la création de nouveaux processus illustrée dans la figure 3.9. Cependant, les shells UNIX permettent généralement également au processus enfant de s'exécuter en arrière-plan ou simultanément. Pour ce faire, nous ajoutons une esperluette (&) à la fin de la commande. Ainsi, si nous réécrivons la commande ci-dessus comme

```
osh>cat prog.c &
```

les processus parent et enfant s'exécuteront simultanément.

Le processus enfant séparé est créé à l'aide de l' appel système fork() et la commande de l'utilisateur est exécutée à l'aide de l'un des appels système de la famille exec() (comme décrit dans la section 3.3.1).

Le programme AC qui fournit les opérations générales d'un shell de ligne de commande est fourni dans la figure 3.36. La fonction main() présente l'invite osh-> et décrit les étapes à suivre après la lecture des entrées de l'utilisateur. La fonction main() boucle continuellement aussi longtemps que son exécution est égale à 1 ; lorsque l'utilisateur entre exit à l'invite, votre programme doit s'exécuter à 0 et se terminer.

```
#include <stdio.h>
#include <unistd.h>

#define MAX LINE 80 /* La commande de longueur maximale */

int main(void)
{ char *args[MAX LINE/2 + 1]; /* arguments de ligne de commande */ int
devrait run = 1; /* indicateur pour déterminer quand quitter le programme */

    while (devrait s'exécuter)
    { printf("osh>");
      fflush(sortie standard);
      /**
       * Après avoir lu les entrées de l'utilisateur, les étapes
       sont : * (1) créer un processus enfant en utilisant fork()
       * (2) le processus enfant invoquera execvp() * (3) le parent
       invoquera wait() sauf si la commande est incluse & */

    }

    renvoie 0 ;
}
```

Figure 3.36 Aperçu d'une coque simple.

Ce projet est organisé en plusieurs parties :

1. Création du processus enfant et exécution de la commande dans l'enfant 2. Fourniture d'une fonctionnalité d'historique 3. Ajout de la prise en charge de la redirection d'entrée et de sortie 4. Permettre aux processus parent et enfant de communiquer via un canal

II. Exécution d'une commande dans un processus enfant

La première tâche consiste à modifier la fonction main() de la figure 3.36 afin qu'un processus enfant soit forké et exécute la commande spécifiée par l'utilisateur. Cela nécessitera d'analyser ce que l'utilisateur a entré dans des jetons séparés et de stocker les jetons dans un tableau de chaînes de caractères (arguments dans la figure 3.36). Par exemple, si l'utilisateur saisit la commande ps -ael à l'invite osh> , les valeurs stockées dans le tableau args sont :

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

Ce tableau args sera passé à la fonction execvp() , qui possède le prototype suivant :

```
execvp(char *commande, char *params[])
```

Ici, command représente la commande à exécuter et params stocke les paramètres de cette commande. Pour ce projet, la fonction execvp() doit être invoquée en tant que execvp(args[0], args). Assurez-vous de vérifier si l'utilisateur a inclus & pour déterminer si le processus parent doit attendre ou non la sortie de l'enfant.

III. Création d'une fonctionnalité d'historique

La tâche suivante consiste à modifier le programme d'interface shell afin qu'il fournisse une fonction d'historique permettant à un utilisateur d'exécuter la commande la plus récente en entrant !!. Par exemple, si un utilisateur entre la commande ls -l, il peut alors exécuter à nouveau cette commande en entrant !! à l'invite. Toute commande exécutée de cette manière doit être répercutée sur l'écran de l'utilisateur et la commande doit également être placée dans le tampon d'historique en tant que commande suivante.

Votre programme doit également gérer la gestion des erreurs de base. S'il n'y a pas de commande récente dans l'historique, entrez !! devrait entraîner un message « Aucune commande dans l'historique ».

IV. Redirection de l'entrée et de la sortie

Votre shell doit alors être modifié pour prendre en charge la redirection '>' et '<'

opérateurs, où '>' redirige la sortie d'une commande vers un fichier et '<' redirige l'entrée vers une commande à partir d'un fichier. Par exemple, si un utilisateur saisit

```
osh>ls > out.txt
```

la sortie de la commande ls sera redirigée vers le fichier out.txt. De la même manière, les entrées peuvent également être redirigées. Par exemple, si l'utilisateur saisit

```
osh>trier <in.txt
```

le fichier in.txt servira d'entrée à la commande de tri .

La gestion de la redirection des entrées et des sorties impliquera l'utilisation de la fonction dup2() , qui duplique un descripteur de fichier existant vers un autre descripteur de fichier. Par exemple, si fd est un descripteur de fichier vers le fichier out.txt, l'appel

```
dup2(fd, STDOUT FILENO);
```

duplique fd sur la sortie standard (le terminal). Cela signifie que toute écriture sur la sortie standard sera en fait envoyée vers le fichier out.txt .

Vous pouvez supposer que les commandes contiendront soit une redirection d'entrée, soit une redirection de sortie et ne contiendront pas les deux. En d'autres termes, vous n'avez pas à vous soucier des séquences de commandes telles que sort < in.txt > out.txt.

V. Communication via un canal

La dernière modification de votre shell consiste à permettre à la sortie d'une commande de servir d'entrée à une autre à l'aide d'un tube. Par exemple, la séquence de commandes suivante

```
osh>ls -l | moins
```

a la sortie de la commande ls -l servir d'entrée à la commande less . Les commandes ls et less s'exécuteront comme des processus séparés et communiqueront en utilisant la fonction UNIX pipe() décrite dans la section 3.7.4.

Le moyen le plus simple de créer ces processus séparés est peut-être de demander au processus parent de créer le processus enfant (qui exécutera ls -l). Cet enfant créera également un autre processus enfant (qui s'exécutera moins) et établira un canal entre lui et le processus enfant qu'il crée. L'implémentation de la fonctionnalité pipe nécessitera également l'utilisation de la fonction dup2() comme décrit dans la section précédente. Enfin, bien que plusieurs commandes puissent être chaînées à l'aide de plusieurs tubes, vous pouvez supposer que les commandes ne contiendront qu'un seul caractère de barre verticale et ne seront combinées avec aucun opérateur de redirection.

Projet 2 — Module du noyau Linux pour les informations sur les tâches Dans ce projet,

vous écrirez un module du noyau Linux qui utilise le système de fichiers /proc pour afficher les informations d'une tâche en fonction de sa valeur d'identifiant de processus pid. Avant de commencer ce projet, assurez-vous d'avoir terminé le projet de programmation du module du noyau Linux du chapitre 2, qui implique la création d'une entrée dans le système de fichiers /proc . Ce projet impliquera l'écriture d'un identifiant de processus pour

le fichier /proc/pid. Une fois qu'un pid a été écrit dans le fichier /proc , les lectures ultérieures à partir de /proc/pid rapporteront (1) la commande exécutée par la tâche, (2) la valeur du pid de la tâche et (3) l'état actuel du tâche. Un exemple de la façon dont votre module noyau sera accessible une fois chargé dans le système est le suivant

```
echo "1395" > /proc/pid cat /
proc/pid command
= [bash] pid = [1395] state = [1]
```

La commande echo écrit les caractères « 1395 » dans le fichier /proc/pid . Votre module noyau lira cette valeur et stockera son équivalent entier car il représente un identifiant de processus. La commande cat lit depuis /proc/pid, où votre module noyau récupérera les trois champs de la structure de tâche associée à la tâche dont la valeur pid est 1395.

```
ssize t proc write (struct file *file, char user *usr buf, size t count, loff t *pos)
{
    int RV = 0 ;
    char *k mem;

    /* alloue de la mémoire au noyau */ k
    mem = kmalloc(count, GFP KERNEL);

    /* copie l'espace utilisateur usr buf dans la mémoire du noyau */
    copy from user(k mem, usr buf, count);

    printk(KERN INFO "%s n", k mem);

    /* renvoie la mémoire du noyau */
    kfree(k mem);

    nombre de retours ;
}
```

Figure 3.37 La fonction proc write() .

I. Écriture dans le système de fichiers /proc

Dans le projet de module de noyau du chapitre 2, vous avez appris à lire à partir du système de fichiers /proc . Nous expliquons maintenant comment écrire dans /proc. Définition du champ .write dans les opérations sur le fichier struct sur

```
.write = proc écriture
```

provoque l'appel de la fonction proc write() de la figure 3.37 lorsqu'une opération d'écriture est effectuée dans /proc/pid

La fonction kmalloc() est l'équivalent noyau de la fonction mal-loc() au niveau utilisateur pour l'allocation de mémoire, sauf que la mémoire noyau est allouée. L' indicateur GFP KERNEL indique une allocation de mémoire de routine du noyau.
La fonction copy from user() copie le contenu de usr buf (qui contient ce qui a été écrit dans / proc/pid) dans la mémoire du noyau récemment allouée. Votre module noyau devra obtenir l'équivalent entier de cette valeur en utilisant la fonction noyau kstrtol(), qui a la signature

int kstrtol (const char *str, base int non signée, long *res)

Ceci stocke le caractère équivalent de str, qui est exprimé sous forme de base dans rés.

Enfin, notez que nous renvoyons la mémoire précédemment allouée avec kmalloc() au noyau avec l'appel à kfree(). Une gestion minutieuse de la mémoire, qui inclut la libération de mémoire pour éviter les fuites de mémoire, est cruciale lors du développement de code au niveau du noyau.

II. Lecture à partir du système de fichiers /proc

Une fois l'identifiant du processus stocké, toute lecture depuis /proc/pid renverra le nom de la commande, son identifiant de processus et son état.
Comme illustré dans la section 3.1, le PCB sous Linux est représenté par la structure task struct, qui se trouve dans le fichier d'inclusion <linux/sched.h> . Étant donné un identifiant de processus, la fonction pid task() renvoie la structure de tâche associée. La signature de cette fonction apparaît comme suit :

tâche struct tâche struct pid (struct pid *pid, type enum pid type)

La fonction du noyau find vpid(int pid) peut être utilisée pour obtenir la structure pid, et PIDTYPE PID peut être utilisé comme type de pid.

Pour un pid valide dans le système, la tâche pid renverra sa structure de tâche. Vous pouvez ensuite afficher les valeurs de la commande, du pid et de l'état. (Vous devrez probablement lire la structure de la tâche dans <linux/sched.h> pour obtenir les noms de ces champs.)

Si pid task() ne reçoit pas de pid valide, il renvoie NULL. Assurez-vous d'effectuer une vérification d'erreur appropriée pour vérifier cette condition. Si cette situation se produit, la fonction du module noyau associée à la lecture depuis /proc/pid doit renvoie 0.

Dans le téléchargement du code source, nous donnons le programme C pid.c, qui produit fournit certains des éléments de base pour démarrer ce projet.

Projet 3—Module du noyau Linux pour lister les tâches Dans ce projet, vous écrirez un module de noyau qui répertorie toutes les tâches en cours dans un système Linux. Vous parcourirez d'abord les tâches de manière linéaire et approfondie.

Partie I : Itérer linéairement sur les tâches

Dans le noyau Linux, la macro for each process() permet facilement d'itérer sur toutes les tâches en cours dans le système :

```
#include <linux/sched.h>

struct tâche struct *tâche ;

pour chaque processus (tâche) {
    /* à chaque tâche itérative pointe vers la tâche suivante */
}
```

Les différents champs de la structure de tâche peuvent ensuite être affichés pendant que le programme parcourt la macro for each process() .

Affectation

Concevez un module de noyau qui parcourt toutes les tâches du système à l'aide de la macro for each process() . En particulier, affichez la commande de tâche, l'état et l'identifiant de processus de chaque tâche. (Vous devrez probablement lire la structure de la tâche dans <linux/sched.h> pour obtenir les noms de ces champs.) Écrivez ce code dans le point d'entrée du module afin que son contenu apparaisse dans le tampon du journal du noyau, qui peut être visualisé à l'aide de la commande dmesg . Pour vérifier que votre code fonctionne correctement, comparez le contenu du tampon du journal du noyau avec le résultat de la commande suivante, qui répertorie toutes les tâches du système :

```
ps -el
```

Les deux valeurs doivent être très similaires. Toutefois, les tâches étant dynamiques, il est possible que quelques tâches apparaissent dans une liste mais pas dans l'autre.

Partie II : Itérer sur les tâches avec un arbre de recherche en profondeur

La deuxième partie de ce projet consiste à parcourir toutes les tâches du système à l'aide d'un arbre de recherche en profondeur (DFS) . (À titre d'exemple : l' itération DFS des processus de la figure 3.7 est 1, 8415, 8416, 9298, 9204, 2808, 3028, 3610, 4005.)

Linux gère son arborescence de processus sous la forme d'une série de listes. Examiner le task struct dans <linux/sched.h>, nous voyons deux objets de tête de liste struct :

```
enfants
```

et

```
frère et sœur
```

Ces objets sont des pointeurs vers une liste des enfants de la tâche, ainsi que ses frères et sœurs. Linux conserve également une référence à la tâche initiale du système – init task – qui est de type task struct. En utilisant ces informations ainsi que des opérations de macro sur les listes, nous pouvons parcourir les enfants de la tâche init comme suit :

```
struct tâche struct *tâche ; struct
tête de liste *liste ;
```

```
liste pour chacun (liste, &init tâche-> enfants) { tâche =
    entrée de liste (liste, struct tâche struct, frère); /* la tâche pointe vers
    l'enfant suivant dans la liste */
} –
```

La liste de la macro each() reçoit deux paramètres, tous deux de type struct tête de liste :

- Un pointeur vers la tête de la liste à parcourir
- Un pointeur vers le nœud de tête de la liste à parcourir

A chaque itération de list pour each(), le premier paramètre est défini sur la structure de liste du prochain enfant. Nous utilisons ensuite cette valeur pour obtenir chaque structure de la liste à l'aide de la macro list Entry() .

Affectation

En commençant par la tâche d'initialisation , concevez un module de noyau qui parcourt toutes les tâches du système à l'aide d'une arborescence DFS . Tout comme dans la première partie de ce projet, affichez le nom, l'état et le pid de chaque tâche. Effectuez cette itération dans le module d'entrée du noyau afin que sa sortie apparaisse dans le tampon du journal du noyau.

Si vous affichez toutes les tâches du système, vous verrez peut-être beaucoup plus de tâches qu'avec la commande ps -ael . En effet, certains threads apparaissent comme des enfants mais n'apparaissent pas comme des processus ordinaires. Par conséquent, pour vérifier la sortie de l' arborescence DFS , utilisez la commande

```
ps-eLf
```

Cette commande répertorie toutes les tâches, y compris les threads, du système. Pour vérifier que vous avez bien effectué une itération DFS appropriée , vous devrez examiner les relations entre les différentes tâches générées par la commande ps .

Projet 4 : Structures de données du noyau

Dans la section 1.9, nous avons couvert diverses structures de données courantes dans les systèmes d'exploitation. Le noyau Linux fournit plusieurs de ces structures. Ici, nous explorons en utilisant la liste circulaire à double chaînage disponible pour les développeurs du noyau. Une grande partie de ce dont nous discutons est disponible dans le code source de Linux (dans ce cas, le fichier d'inclusion <linux/list.h>) et nous vous recommandons d'examiner ce fichier au fur et à mesure que vous suivez les étapes suivantes.

Dans un premier temps, vous devez définir une structure contenant les éléments qui doivent être insérés dans la liste chaînée. La structure C suivante définit une couleur comme un mélange de rouge, de bleu et de vert :

```
struct color { int
    rouge ;
    bleu int;
    vert int;
```

```
    liste de têtes de liste de structures ;
} ;
```

Notez la liste d'en-tête de la liste de structures des membres. La structure d'en-tête de liste est définie dans le fichier d'inclusion <linux/types.h>, et son intention est d'intégrer la liste chaînée dans les nœuds qui composent la liste. Cette structure de tête de liste est assez simple : elle contient simplement deux membres, next et prev, qui pointent vers les entrées suivantes et précédentes de la liste. En intégrant la liste chaînée dans la structure, Linux permet de gérer la structure des données avec une série de fonctions macro .

I. Insertion d'éléments dans la liste liée

Nous pouvons déclarer un objet tête de liste , que nous utilisons comme référence à la tête de liste en utilisant la macro LIST HEAD() :

```
statique LIST HEAD (liste de couleurs);
```

Cette macro définit et initialise la liste de couleurs variables, qui est de type tête de liste de structures.

Nous créons et initialisons des instances de struct color comme suit :

```
couleur de structure *violet ;

violet = kmalloc(sizeof(*violet), GFP KERNEL); violet->rouge =
138 ; violet->bleu = 43 ;
violet->vert = 226 ;

INIT LIST HEAD(&violet->liste);
```

La fonction kmalloc() est l'équivalent noyau de la fonction malloc() au niveau utilisateur pour l'allocation de mémoire, sauf que la mémoire noyau est allouée.
L' indicateur GFP KERNEL indique une allocation de mémoire de routine du noyau. La macro INIT LIST HEAD() initialise le membre de la liste en couleur de structure. Nous pouvons ensuite ajouter cette instance à la fin de la liste chaînée en utilisant la macro list add tail() :

```
list add tail(&violet->list, &color list);
```

II. Parcourir la liste chaînée

Parcourir la liste implique d'utiliser la liste pour chaque macro Entry(), qui accepte trois paramètres :

- Un pointeur vers la structure sur laquelle l'itération est effectuée
- Un pointeur vers l'en-tête de la liste sur laquelle l'itération est effectuée
- Le nom de la variable contenant la structure de l'en-tête de la liste

Le code suivant illustre cette macro :

```
couleur de structure *ptr;

list pour chaque entrée (ptr, &color list, list) { /* à chaque itération
    ptr pointe */ /* vers la couleur de structure suivante
    */
} _ _ _ _
```

III. Suppression d'éléments de la liste liée

La suppression d'éléments de la liste implique l'utilisation de la macro list del() , à laquelle est transmis un pointeur vers la tête de liste struct :

```
list del(struct list head *élément);
```

Cela supprime un élément de la liste tout en conservant la structure du reste de la liste.

L'approche la plus simple pour supprimer tous les éléments d'une liste chaînée consiste peut-être à supprimer chaque élément lorsque vous parcourez la liste. La liste de macros pour chaque entrée safe() se comporte un peu comme la liste pour chaque entrée() sauf qu'elle reçoit un argument supplémentaire qui conserve la valeur du pointeur suivant de l'élément en cours de suppression. (Cela est nécessaire pour préserver la structure de la liste.) L'exemple de code suivant illustre cette macro :

```
struct couleur *ptr, *suivant ;

liste pour chaque entrée sécurisée (ptr, suivant, & liste de couleurs, liste) {
    /* à chaque itération ptr pointe */ /* vers la couleur
    de structure suivante */ list del(&ptr->list);
_ _ kfree(ptr); _ _

}
```

Notez qu'après avoir supprimé chaque élément, nous renvoyons au noyau la mémoire précédemment allouée avec kmalloc() avec l'appel à kfree().

Partie I – Devoir

Dans le point d'entrée du module, créez une liste chaînée contenant quatre éléments de couleur struct . Parcourez la liste chaînée et affichez son contenu dans le tampon du journal du noyau. Invoquez la commande dmesg pour vous assurer que la liste est correctement construite une fois le module du noyau chargé.

Au point de sortie du module, supprimez les éléments de la liste chaînée et renvoyez la mémoire libre au noyau. Encore une fois, appelez la commande dmesg pour vérifier que la liste a été supprimée une fois le module du noyau déchargé.

Partie II – Passage de paramètres

Cette partie du projet impliquera de transmettre un paramètre à un module du noyau. Le module utilisera ce paramètre comme valeur initiale et générera la séquence Collatz comme décrit dans l'exercice 3.21.

Passer un paramètre à un module du noyau

Les paramètres peuvent être transmis aux modules du noyau lors de leur chargement. Par exemple, si le nom du module du noyau est collatz, on peut passer la valeur initiale de 15 au paramètre du noyau start comme suit :

```
sudo insmod collatz.ko start=15
```

Dans le module noyau, nous déclarons start en paramètre en utilisant le code suivant :

```
#include<linux/moduleparam.h>

début int statique = 25 ;

paramètre de module (début, int, 0);
```

La macro module param() est utilisée pour établir des variables comme paramètres des modules du noyau. module param() reçoit trois arguments : (1) le nom du paramètre, (2) son type et (3) les autorisations de fichier. Puisque nous n'utilisons pas de système de fichiers pour accéder au paramètre, nous ne nous soucions pas des autorisations et utilisons une valeur par défaut de 0. Notez que le nom du paramètre utilisé avec la commande insmod doit correspondre au nom du paramètre du noyau assoc Enfin, si nous ne fournissons pas de valeur au paramètre module lors du chargement avec insmod, la valeur par défaut (qui dans ce cas est 25) est utilisée.

Partie II—Affectation

Concevez un module de noyau nommé collatz auquel est transmise une valeur initiale en tant que paramètre de module. Votre module générera et stockera ensuite la séquence dans une liste chaînée du noyau lorsque le module sera chargé. Une fois la séquence stockée, votre module parcourra la liste et affichera son contenu dans le tampon du journal du noyau. Utilisez la commande dmesg pour vous assurer que la séquence est correctement générée une fois le module chargé.

Au point de sortie du module, supprimez le contenu de la liste et restituez la mémoire libre au noyau. Encore une fois, utilisez dmesg pour vérifier que la liste a été supprimée une fois le module du noyau déchargé.