Services et Interfaces d'un Système d'exploitation

Concept des appels système

Ilias TOUGUI

L'Ecole Supérieure d'Informatique et du Numérique

INF2132 - Systèmes d'Exploitation

PLAN DU COURS

1. Introduction
2. Les services d'un système d'exploitation
3. Interfaces utilisateurs et système d'exploitation
4. Les appels Système

Introduction

Un système d'exploitation fournit l'environnement dans lequel les programmes sont exécutés. En interne, les systèmes d'exploitation varient considérablement dans leur composition, car ils sont organisés selon de nombreuses orientations différentes.

Nous pouvons considérer un système d'exploitation sous plusieurs angles:

- Les services que le système fournit
- L'interface qu'il met à disposition des utilisateurs et des programmeurs
- Ses composants et leurs interconnexions
- Les mécanismes d'interaction entre les programmes et le système

Objectifs du chapitre

Dans ce chapitre, nous explorons ces aspects des systèmes d'exploitation, en montrant les points de vue des utilisateurs, des programmeurs et des concepteurs de systèmes d'exploitation.

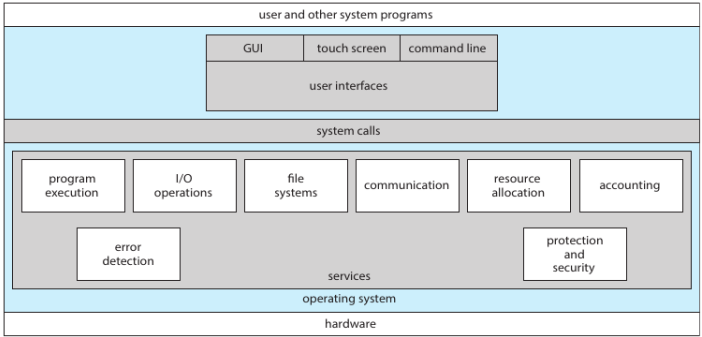
À la fin de ce chapitre, vous serez capable de:

- Identifier les services fournis par un système d'exploitation
- Distinguer les différents types d'interfaces utilisateur (CLI, GUI, tactile)
- Comprendre comment les appels système sont utilisés pour fournir les services du SE
- Expliquer la relation entre API, interface d'appels système et noyau
- Classifier les six catégories d'appels système avec des exemples concrets
- Identifier les différents types de programmes utilitaires système

Partie I: Les services d'un système d'exploitation

Les services d'un SE

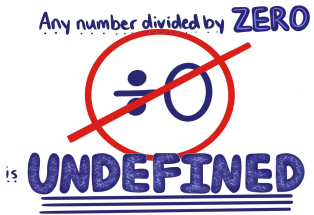
Un système d'exploitation fournit un environnement pour l'exécution des programmes. Il met certains services à la disposition des programmes et de leurs utilisateurs. Bien sûr, les services spécifiques fournis diffèrent d'un système d'exploitation à l'autre, mais on peut identifier des classes communes.



Une vue des services du système d'exploitation

1. Exécution du programme

Le système doit pouvoir charger un programme en mémoire et l'exécuter. Le pragramme doit pouvoir terminer son exécution, normalement ou anormalement (en indiquant une erreur).



Exemple d'exécution anormale

2. Opérations d'E/S

Un programme en cours d'exécution peut nécessiter des E/S, qui peuvent impliquer un fichier ou un périphérique d'E/S. Pour certains périphériques, des fonctions spécifiques peuvent être requises (comme la lecture depuis une interface réseau ou l'écriture dans un système de fichiers).

Pour des raisons d'efficacité et de sécurité, les utilisateurs ne peuvent généralement pas contrôler directement les périphériques d'E/S. Par conséquent, le système d'exploitation doit fournir un moyen d'effectuer des opérations d'E/S.

Exemple:

Une application de messagerie veut envoyer un email. Elle ne peut pas contrôler directement la carte réseau. Elle utilise les services du système d'exploitation via des sockets pour établir une connexion TCP/IP. Le système d'exploitation gère les protocoles réseau, l'encapsulation des données, et la communication avec la carte réseau.

3. Manipulation du système de fichiers

Le système de fichiers présente un intérêt particulier. De toute évidence, les programmes doivent:

- Lire et écrire des fichiers et des répertoires.
- Créer et les supprimer par nom
- Rechercher un fichier donné
- Gestion des permissions pour autoriser ou refuser l'accès aux fichiers ou aux répertoires en fonction de leur propriétaire.

Exemple:

Un étudiant crée un nouveau dossier Images_Projet pour organiser ses ressources, puis supprime un ancien fichier brouillon_v1.txt devenu inutile. Le système met à jour la table des fichiers pour refléter ces changements

4. Communication entre processus

Les communications entre processus permettent à différents programmes d'échanger des informations, que ce soit sur le même ordinateur ou via un réseau. La communication entre processus se divise en deux categories:

1. Communication par mémoire partagée
2. Communication par passage de messages

4. Communication entre processus

Exemples:

1. Communication par mémoire partagée

Le navigateur Chrome et le plugin Flash Player partagent une zone mémoire commune. Quand vous regardez une vidéo, le navigateur écrit les données vidéo dans cette zone mémoire partagée, et le plugin Flash lit ces mêmes données pour afficher la vidéo. Les deux processus accèdent simultanément à la même région mémoire.

2. Communication par passage de messages

L'antivirus envoie des messages formatés au système d'exploitation pour signaler la détection d'un virus. Le message contient des informations structurées : "ALERTE|virus_trojan.exe|`C:\Users\Documents`*|QUARANTAINE_REQUISE"*

5. Detection des erreurs

La détection d'erreurs permet au système d'exploitation d'identifier et de corriger les problèmes pour maintenir la stabilité du système. La détection d'erreurs se divise en trois categories:

1. Erreurs matérielles - CPU et mémoire
2. Erreurs des périphériques d'E/S
3. Erreurs des programmes utilisateur

5. Detection des erreurs

Exemples:

1.Erreurs matérielles - CPU et mémoire
Lors d'une coupure électrique soudaine, l'onduleur signale l'événement au système d'exploitation qui sauvegarde immédiatement les données critiques, ferme proprement les applications ouvertes, et met le système en veille sécurisée.

2. Erreurs des périphériques d'E/S
L'imprimante signale qu'elle n'a plus de papier. Le système suspend la tâche d'impression et affiche une notification "Ajouter du papier dans l'imprimante Canon LP150"

3. Erreurs des programmes utilisateur
Un programme de calcul scientifique tente de calculer 10^{500}*, dépassant la capacité de stockage. Le système génère une exception, termine le processus défaillant, et affiche "Erreur: Dépassement de capacité arithmétique"*

6. Allocation des ressources

L'allocation des ressources permet au système d'exploitation de distribuer équitablement les ressources matérielles entre plusieurs processus qui s'exécutent simultanément. L'allocation des ressources permet une:

1. Allocation du processeur (CPU)
2. Allocation de la mémoire
3. Allocation du stockage
4. Allocation des périphériques d'E/S

6. Allocation des ressources

Exemple: Multitâche sur un ordinateur

1. Allocation du processeur (CPU)

Votre ordinateur exécute simultanément Chrome, Spotify, et un antivirus. L'ordonnanceur du système alloue des tranches de temps CPU :

- Chrome reçoit 60% du temps CPU (processus prioritaire car interaction utilisateur)
- Spotify obtient 25% (lecture audio continue)
- L'antivirus reçoit 15% (tâche de fond)

Le système ajuste dynamiquement ces allocations selon l'activité. Quand vous ouvrez un jeu vidéo, l'ordonnanceur lui donne la priorité en réduisant les parts des autres processus.

6. Allocation des ressources

Exemple: Gestion RAM

2. Allocation de la mémoire

Avec 8 GB de RAM disponible, le système alloue :

- 2 GB pour le système d'exploitation
- 1 GB pour votre navigateur avec 20 onglets ouverts
- 3 GB pour un logiciel de montage vidéo
- 2 GB restent disponibles pour de nouveaux programmes

Si un nouveau processus demande 3 GB mais seulement 2 GB sont libres, le système utilise la mémoire virtuelle (swap) ou demande à fermer des applications.

6. Allocation des ressources

Exemple: Espace disque

3. Allocation du stockage

Plusieurs applications tentent d'écrire simultanément :

- Un téléchargement de film nécessite 4 GB
- Une sauvegarde système demande 10 GB
- Un jeu s'installe et requiert 25 GB

Le système vérifie l'espace disponible, alloue les blocs de stockage nécessaires, et gère les accès concurrents pour éviter la corruption des données.

6. Allocation des ressources

Exemple: Gestion d'imprimante

4. Allocation des périphériques d'E/S

Trois utilisateurs envoient des documents à imprimer simultanément :

- Rapport_projet.pdf (5 pages) - Utilisateur A
- Presentation_cours.pptx (50 pages) - Utilisateur B
- Facture_urgente.pdf (1 page) - Utilisateur C

Le système crée une file d'attente, peut prioriser la facture urgente, et s'assure qu'un seul processus utilise l'imprimante à la fois.

7. L'accounting

L'accounting permet au système d'exploitation d'enregistrer et de mesurer l'utilisation des ressources (CPU, mémoire, disque, réseau) par chaque utilisateur et processus pour la facturation, l'audit ou l'optimisation du système.

Exemple: Université - laboratoire informatique

Le système suit l'usage de chaque étudiant :

- Étudiant A : 8 heures CPU, 2 GB impression, 50 GB stockage
- Étudiant B : 15 heures CPU, 500 MB impression, 25 GB stockage

L'université peut facturer les dépassements ou allouer des quotas équitables en fonction des statistiques collectées.

8. La protection et sécurité

La protection et sécurité permettent au système d'exploitation de contrôler l'accès aux ressources et de protéger le système contre les menaces internes et externes.

Exemple: Éditeur de texte et programme malveillant

Le système d'exploitation isole complètement les deux processus. Le malware ne peut pas :

- Lire le contenu de votre document Word en cours d'édition
- Modifier votre texte pour y insérer du contenu malveillant
- Accéder aux fichiers récemment ouverts dans Word
- Intercepter les frappes clavier destinées à Word

Chaque processus fonctionne dans son propre espace d'adressage virtuel, rendant impossible l'accès croisé aux données.

Partie II: Interfaces utilisateurs et système d'exploitation

Introduction

Nous avons mentionné précédemment qu'il existe plusieurs façons pour les utilisateurs d'interagir avec le système d'exploitation. Nous abordons ici trois approches fondamentales. La première propose une interface en ligne de commande, ou interpréteur de commandes, qui permet aux utilisateurs de saisir directement les commandes à exécuter par le système d'exploitation. Les deux autres permettent aux utilisateurs d'interagir avec le système d'exploitation via une interface utilisateur graphique, ou GUI.

Interpréteur de commandes

La plupart des systèmes d'exploitation (Linux, UNIX, Windows) traitent l'interpréteur de commandes comme un programme spécial qui s'exécute lors de l'initialisation d'un processus ou de la première connexion d'un utilisateur.

Sur les systèmes avec plusieurs interpréteurs disponibles, ceux-ci sont appelés **shells**.

Exemples de shells UNIX/Linux:

- C shell
- Bourne-Again shell (bash)
- Korn shell
- Shells tiers et gratuits développés par les utilisateurs

Le choix du shell est généralement basé sur les préférences personnelles.

Fonction principale de l'interpréteur

La fonction principale de l'interpréteur de commandes est d'obtenir et d'exécuter la prochaine commande spécifiée par l'utilisateur.

Commandes courantes de manipulation de fichiers:

- Créer
- Supprimer
- Lister
- Imprimer
- Copier
- Exécuter

Ces commandes peuvent être implémentées de deux manières générales.

Approche 1: Code intégré (built-in)

Principe: Dans cette approche, l'interpréteur de commandes contient directement le code de chaque commande à l'intérieur de son propre programme.

Fonctionnement:

- L'utilisateur tape une commande (ex: `rm fichier.txt`)
- L'interpréteur identifie la commande demandée
- Il exécute le code correspondant qui est déjà écrit dans son programme
- Aucun programme externe n'est appelé

Analogie: Un couteau suisse avec tous les outils intégrés dans un seul objet (tournevis, ciseaux, ouvre-bouteille...). Chaque outil fait partie du couteau.

Approche 1: Exemple de code

Code simplifié de l'interpréteur:

Structure de l'interpréteur avec code intégré

```
// Boucle principale de l'interpréteur
while (1) {
 lire_commande(cmd, args);  // Lire l'entrée utilisateur

 if (strcmp(cmd, "rm") == 0) {
  // Code intégré pour supprimer
  unlink(args[0]);

 } else if (strcmp(cmd, "ls") == 0) {
  // Code intégré pour lister
  DIR *d = opendir(".");
  // ... afficher les fichiers
 }
 // ... et ainsi de suite pour chaque commande
}
```

Approche 1: Analyse détaillée

Que se passe-t-il dans le code?

1. **Lecture:** L'interpréteur lit la commande tapée par l'utilisateur
2. **Comparaison:** `strcmp(cmd, "rm")` compare la commande avec "rm", "ls", "cp"...
3. **Exécution directe:** Si correspondance, le code de cette commande (déjà écrit dans l'interpréteur) s'exécute
4. **Appel système:** Le code fait directement l'appel système approprié (`unlink()`, `opendir()`...)

Remarque importante: Toutes les commandes sont codées à l'intérieur du même fichier exécutable de l'interpréteur.

Approche 1: Avantages et inconvénients

Avantages

- **Rapidité:** Pas besoin de charger un programme externe
- **Simplicité conceptuelle:** Tout est au même endroit

Inconvénients majeurs

- **Taille énorme:** L'interpréteur devient très volumineux
  - 10 commandes = beaucoup de code
  - 100 commandes = fichier exécutable gigantesque!
- **Maintenance difficile:** Pour ajouter/modifier une commande, il faut:
  - Modifier le code source de l'interpréteur
  - Recompiler tout le programme
  - Redistribuer l'interpréteur complet
- **Pas extensible:** Impossible d'ajouter des commandes personnalisées facilement

Approche 1: Exemple concret

Scénario: Un interpréteur avec code intégré contenant 50 commandes (rm, ls, cp, mv, cat, grep, sed, awk...).

Résultat:

- Taille du fichier exécutable: ~5 MB
- Mémoire utilisée au démarrage: ~5 MB (tout le code est chargé)
- Pour ajouter la commande "find": modifier l'interpréteur, recompiler, redistribuer

Question: Peut-on faire mieux?

Réponse: Oui! C'est l'approche 2 avec les programmes externes séparés (que nous verrons ensuite).

Approche 1: Exemples réels

Où trouve-t-on cette approche aujourd'hui?

- **Commandes built-in de bash:**
  - `cd` (changer de répertoire)
  - `pwd` (afficher le répertoire courant)
  - `exit` (quitter le shell)
  - `export` (définir des variables d'environnement)

- **Pourquoi ces commandes sont built-in?**
  - Elles modifient l'état interne du shell
  - Elles sont très fréquemment utilisées
  - Elles doivent s'exécuter rapidement

Note: Les shells modernes utilisent une **approche hybride**: quelques commandes intégrées + la majorité en programmes externes.

Approche 2: Programmes système (UNIX)

Approche utilisée par UNIX: la plupart des commandes sont implémentées via des programmes système.

L'interpréteur ne comprend pas la commande — il utilise simplement la commande pour identifier **un fichier** à charger en mémoire et à exécuter.

Exemple:

```
rm fichier.txt
```

1. Recherche un fichier appelé `rm`
2. Charge le fichier en mémoire
3. L'exécute avec le paramètre `fichier.txt`

La logique associée à la commande `rm` est définie dans le fichier `rm.c`.

Approche 2: Voir le code source de rm

Peut-on voir le code source de rm?

Oui! Le programme `rm` est open-source et fait partie des **GNU coreutils**.

Méthodes pour accéder au code source

```
# Télécharger les sources depuis GNU
$ wget https://ftp.gnu.org/gnu/coreutils/coreutils-9.1.tar.gz
$ tar -xzf coreutils-9.1.tar.gz
$ cd coreutils-9.1/src
$ cat rm.c
$ wc -l rm.c
```

Caractéristiques du code rm.c:

- Environ 373 lignes de code C
- Gère les options: `-r` (récursif), `-f` (force), `-i` (interactif)
- Utilise les appels système: `unlink()` et `rmdir()`

Avantages de l'approche par programmes système

Extensibilité facile:
Les programmeurs peuvent ajouter de nouvelles commandes facilement en créant de nouveaux fichiers avec la logique de programme appropriée.

Taille réduite:
Le programme d'interpréteur de commandes peut être petit et n'a pas besoin d'être modifié pour ajouter de nouvelles commandes.

Modularité:
Chaque commande est un programme indépendant, ce qui facilite la maintenance et les mises à jour.

Interface graphique (GUI)

Une deuxième stratégie pour interagir avec le système d'exploitation est l'interface graphique utilisateur (**GUI**).

Caractéristiques:

- Système de fenêtres et de menus basé sur la souris
- Métaphore du bureau (desktop)
- Utilisation d'icônes représentant des programmes, fichiers, répertoires et fonctions système

L'utilisateur déplace la souris pour positionner le pointeur sur des images ou icônes à l'écran. En cliquant, il peut:

- Invoquer un programme
- Sélectionner un fichier ou dossier
- Dérouler un menu contenant des commandes

Historique des interfaces graphiques

Origines (années 1970):
Les interfaces graphiques sont apparues grâce aux recherches menées au centre de recherche Xerox PARC au début des années 1970.

Chronologie:

- **1973:** Première GUI sur l'ordinateur Xerox Alto
- **Années 1980:** Généralisation avec les ordinateurs Apple Macintosh
- **macOS:** Adoption de l'interface Aqua (changement majeur)
- **Windows 1.0:** Ajout d'une interface GUI à MS-DOS
- **Versions ultérieures de Windows:** Changements significatifs d'apparence et améliorations fonctionnelles

GUI sur les systèmes UNIX/Linux

Traditionnellement, les systèmes UNIX ont été dominés par les interfaces en ligne de commande. Récemment diverses interfaces GUI sont disponibles grâce aux projets open-source:

- **KDE** (K Desktop Environment)
- **GNOME** (par le projet GNU)

Identifier son environnement de bureau:

```
$ echo $XDG_CURRENT_DESKTOP
```

Caractéristiques:

- Fonctionnent sur Linux et divers systèmes UNIX
- Disponibles sous licences open-source
- Code source accessible pour lecture et modification

Interface tactile pour appareils mobiles

Les interfaces en ligne de commande ou souris-clavier étant peu pratiques pour les systèmes mobiles, les smartphones et tablettes utilisent typiquement une **interface à écran tactile**. Les utilisateurs interagissent en effectuant des gestes sur l'écran tactile:

- Appuyer avec les doigts
- Balayer (swiper) sur l'écran
- Pincer pour zoomer
- Glisser pour faire défiler

Évolution: Bien que les premiers smartphones incluaient un clavier physique, la plupart des smartphones et tablettes modernes simulent maintenant un clavier directement sur l'écran tactile.

Exemple: L'iPad et l'iPhone d'Apple utilisent l'interface tactile Springboard.

Choix de l'interface: CLI vs GUI

Le choix entre interface en ligne de commande (CLI) et interface graphique (GUI) est principalement une question de préférence personnelle.

Utilisateurs privilégiant la CLI:

- Informaticiens
- Administrateurs système
- Utilisateurs experts avec connaissance approfondie du système

Avantages de la CLI:

- Plus efficace pour les experts
- Accès plus rapide aux activités à effectuer
- Certaines fonctions système uniquement disponibles via CLI
- Facilite les tâches répétitives grâce à la programmabilité

Scripts shell

Les interfaces en ligne de commande permettent l'automatisation via des **scripts shell**. Si une tâche fréquente nécessite plusieurs étapes en ligne de commande, ces étapes peuvent être:

- Enregistrées dans un fichier
- Exécutées comme un programme

Le programme n'est pas compilé en code exécutable, mais interprété par l'interface en ligne de commande.

Usage: Les scripts shell sont très courants sur les systèmes orientés ligne de commande comme UNIX et Linux.

Exemple de Script de création de dossiers

Fichier: create_dirs.sh

```
#!/bin/bash
# Script pour créer des dossiers dans le répertoire utilisateur

echo "Déplacement vers le répertoire utilisateur..."
cd ~

echo "Création de la structure de dossiers..."
mkdir -p projet/docs
mkdir -p projet/tests

echo "Vérification des dossiers créés:"
ls -R projet/
```

Exécution du script:

```
$ chmod +x create_dirs.sh
$ ./create_dirs.sh
```

Préférences selon les systèmes

Windows:

- La plupart des utilisateurs utilisent l'environnement GUI
- L'interface shell est rarement utilisée
- Versions récentes: GUI standard pour ordinateurs de bureau/portables + interface tactile pour tablettes

macOS (évolution intéressante):

- Historiquement: uniquement GUI, aucune CLI
- Depuis macOS (noyau UNIX): interface Aqua GUI + CLI

Systèmes mobiles (iOS/Android):

- Applications CLI existent mais rarement utilisées
- Presque tous les utilisateurs interagissent via l'interface tactile

Interface utilisateur et structure du SE

L'interface utilisateur peut varier d'un système à l'autre et même d'un utilisateur à l'autre au sein d'un même système.

Point important: L'interface utilisateur est généralement substantiellement éloignée de la structure réelle du système.

Conséquence: La conception d'une interface utilisateur utile et intuitive n'est pas une fonction directe du système d'exploitation.

Perspective du SE: Du point de vue du système d'exploitation, il n'y a pas de distinction entre programmes utilisateur et programmes système. L'accent est mis sur la fourniture de services adéquats aux programmes utilisateur.

Partie III: Les appels Système

Introduction

Les **appels système** fournissent une interface aux services mis à disposition par un système d'exploitation. Ces appels sont généralement disponibles sous forme de fonctions écrites en C et C++, bien que certaines tâches de bas niveau (par exemple, celles nécessitant un accès direct au matériel) puissent nécessiter l'utilisation d'instructions en langage assembleur.

Exemple: Copie de fichier

Avant de discuter comment un système d'exploitation rend les appels système disponibles, considérons un exemple pratique.

Question: Comment écrire un programme simple qui lit des données d'un fichier et les copie dans un autre fichier?

Le programme a besoin des noms des deux fichiers:

- Le fichier source (input file)
- Le fichier destination (output file)

Ces noms peuvent être spécifiés de plusieurs manières selon la conception du système d'exploitation.

Approche 1: Ligne de commande

Principe: Passer les noms des fichiers comme paramètres de la commande

Exemple UNIX:

```
cp in.txt out.txt
```

Cette commande copie le fichier `in.txt` vers `out.txt`.

Avantages:

- Simple et direct
- Rapide pour les utilisateurs expérimentés
- Facilement automatisable dans des scripts

Appels système: Les noms sont récupérés directement depuis les arguments de la commande.

Approche 2: Mode interactif

Principe: Le programme demande les noms à l'utilisateur avec plusieurs séquences d'interaction:

Étape 1: Demander le fichier source

- Écrire un message d'invite à l'écran (appel système)
- Exemple: "Entrez le nom du fichier source:"
- Lire la réponse depuis le clavier (appel système)

Étape 2: Demander le fichier destination

- Écrire un message d'invite à l'écran (appel système)
- Exemple: "Entrez le nom du fichier destination:"
- Lire la réponse depuis le clavier (appel système)

Appels système nécessaires: Au minimum 4 (2 pour affichage + 2 pour lecture)

Approche 3: Interface graphique (GUI)

Principe: Utiliser un système de fenêtres et de menus basé sur la souris

Interface typique:
Un menu de noms de fichiers est affiché dans une fenêtre (métaphore du bureau).

Actions de l'utilisateur:
Étape 1: Sélectionner le fichier source

- Utiliser la souris pour cliquer sur le fichier

Étape 2: Spécifier le fichier destination

- Une fenêtre s'ouvre pour entrer ou sélectionner le nom

Appels système nécessaires: Nombreux appels d'E/S pour gérer l'affichage des fenêtres, capturer les événements souris, et lire les sélections.

Opérations de copie

Une fois les noms de fichiers obtenus (quelle que soit l'approche), le programme effectue les opérations suivantes:

Étape 1: Ouvrir le fichier source (appel système)

- Vérifier que le fichier existe
- Vérifier les permissions d'accès
- Gérer les erreurs (fichier inexistant ou protégé)

Étape 2: Créer/ouvrir le fichier destination (appel système)

- Si le fichier existe: demander confirmation ou remplacer
- Créer un nouveau fichier

Étape 3: Copier les données en boucle

- Lire depuis le fichier source (appel système)
- Écrire dans le fichier destination (appel système)
- Répéter jusqu'à la fin du fichier

Gestion des erreurs et terminaison

Durant la copie, gérer les erreurs possibles:

Erreurs en lecture:

- Fin du fichier atteinte (condition normale)
- Défaillance matérielle (erreur de parité)

Erreurs en écriture:

- Plus d'espace disque disponible
- Problèmes matériels du périphérique

Après la copie complète:

Étape 4: Fermer les fichiers

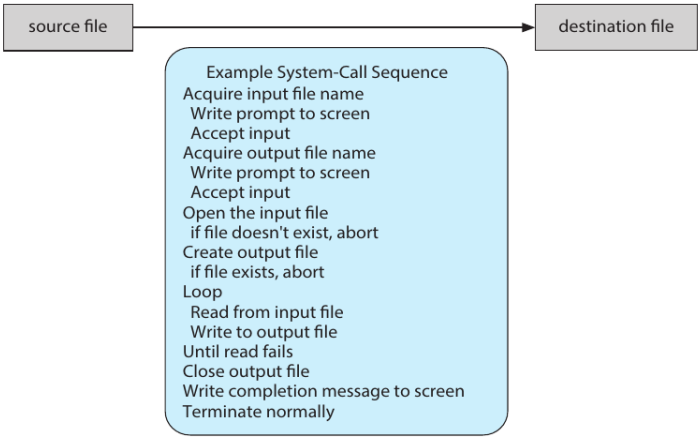
- Fermer le fichier source (appel système)
- Fermer le fichier destination (appel système)

Étape 5: Informer l'utilisateur et terminer

- Afficher un message de succès (appels système)
- Terminer normalement (appel système)

Récapitulatif des appels système

Séquence complète d'appels système pour copier un fichier:



Exemple d'utilisation des appels système

Utilisation intensive des appels système

Comme nous l'avons vu, même les programmes simples peuvent faire un usage intensif du système d'exploitation.

Volume d'appels système:
Les systèmes exécutent fréquemment des milliers d'appels système par seconde.

Abstraction pour les programmeurs:
La plupart des programmeurs ne voient jamais ce niveau de détail.

Pourquoi?
Les développeurs d'applications conçoivent généralement des programmes selon une **interface de programmation d'applications (API)**, qui cache la complexité des appels système.

Interface de programmation (API)

Définition:
L'API spécifie un ensemble de fonctions disponibles pour un programmeur d'applications.

Informations fournies par l'API:

- Les fonctions disponibles
- Les paramètres à passer à chaque fonction
- Les valeurs de retour attendues par le programmeur

Avantages de l'API:

- Abstraction des détails du système d'exploitation
- Interface stable et documentée
- Portabilité entre différentes versions du système
- Simplification du développement

APIs courantes

Les trois APIs les plus courantes:

1. **Windows API**
   - Pour les systèmes Windows
   - Aussi appelée Win32 API ou WinAPI
2. **POSIX API**
   - Pour les systèmes basés sur POSIX
   - Inclut pratiquement toutes les versions d'UNIX, Linux et macOS
   - Standard de portabilité entre systèmes
3. **Java API**
   - Pour les programmes qui s'exécutent sur la machine virtuelle Java (JVM)
   - Indépendante de la plateforme

Accès aux APIs via bibliothèques

Un programmeur accède à une API via une bibliothèque de code fournie par le système d'exploitation.

Exemple: UNIX et Linux
Pour les programmes écrits en langage C, la bibliothèque s'appelle `libc` (C library).

Rôle de la bibliothèque:

- Fournit les fonctions de l'API
- Fait le lien entre l'API et les appels système
- Gère les conversions de paramètres
- Traduit les appels API en appels système appropriés

Avantage:
Le programmeur utilise des fonctions API simples sans connaître les détails des appels système sous-jacents.

Le Manuel Linux : man

Le manuel Linux est organisé en sections numérotées. Pour consulter les pages du manuel, utilisez la commande :

```
$ man [section] commande
```

Sections principales :

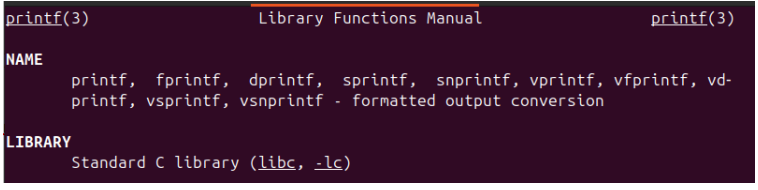
- Section 1 : Commandes utilisateur
- Section 2 : Appels système (fournis par le noyau)
- Section 3 : Fonctions de bibliothèque (API)
- Section 5 : Formats de fichiers
- Section 8 : Commandes d'administration

Exemple D'API Standard

Prenons comme exemple d'API standard la fonction printf(), disponible dans la bibliothèque C standard. L'API de cette fonction est accessible depuis la page de manuel en invoquant la commande :

```
$ man 3 printf
```

Une description de cette API est présentée ci-dessous :



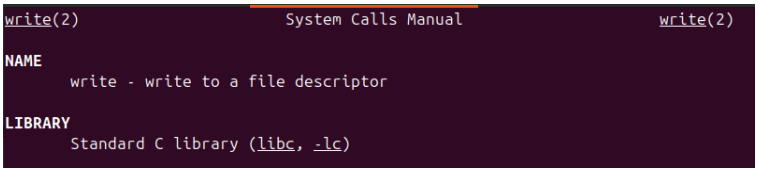
Exemple d'API de la fonction printf()

Exemple D'Appel Système

Prenons comme exemple d'appel système la fonction write(), qui est invoquée en interne par printf(). L'interface de cet appel système est accessible avec :

```
$ man 2 write
```

Une description de cet appel système est présentée ci-dessous :



Exemple d'appel système write()

Pourquoi programmer avec des APIs?

Question:
Pourquoi un programmeur d'applications préférerait-il programmer selon une API plutôt qu'invoquer directement les appels système?

Raison 1: Portabilité du programme
Un programmeur concevant un programme avec une API peut s'attendre à ce que son programme compile et s'exécute sur tout système supportant la même API.

Exemple:
Un programme écrit avec l'API POSIX peut fonctionner sur Linux, macOS, et divers systèmes UNIX sans modification majeure.

Remarque: En réalité, les différences architecturales rendent parfois cela plus difficile qu'il n'y paraît.

Pourquoi programmer avec des APIs? (suite)

Raison 2: Simplicité d'utilisation

Les appels système réels peuvent souvent être:

- Plus détaillés et complexes
- Plus difficiles à utiliser
- Nécessitant une connaissance approfondie du système

En comparaison, les APIs disponibles aux programmeurs d'applications sont:

- Plus abstraites et intuitives
- Mieux documentées
- Plus faciles à apprendre et à utiliser
- Conçues pour la productivité du développeur

Résultat: Les APIs permettent aux programmeurs de se concentrer sur la logique de l'application plutôt que sur les détails du système.

Comment l'API accède aux appels système?

Un programmeur accède à une API via une **bibliothèque de code** fournie par le système d'exploitation.

Exemple: Pour UNIX et Linux, les programmes écrits en C utilisent la bibliothèque appelée `libc`.

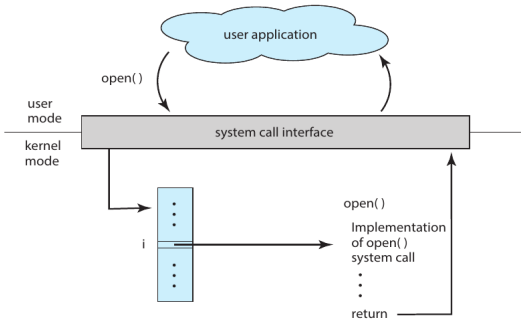
Mécanisme interne:
L'environnement d'exécution fournit une **interface d'appels système** qui:

- Intercepte les appels de fonctions dans l'API
- Utilise une table de numéros pour mapper chaque fonction API à un appel système
- Invoque l'appel système correspondant dans le noyau
- Retourne le résultat au programme

Avantage: Le programmeur n'a pas besoin de connaître ces détails. Il suffit de respecter la signature des fonctions API.

Relation entre API et appels système

Il existe souvent une forte corrélation entre une fonction dans l'API et son appel système associé dans le noyau.



La gestion d'une application utilisateur appelant l'appel système open()

L'API agit comme une couche d'abstraction offrant simplicité et portabilité, tout en cachant la complexité des appels système.

Types d'appels système

Les appels système peuvent être regroupés approximativement en six catégories principales:

1. **Contrôle des processus** (Process Control)
2. **Gestion des fichiers** (File Management)
3. **Gestion des périphériques** (Device Management)
4. **Maintenance de l'information** (Information Maintenance)
5. **Communications** (Communications)
6. **Protection** (Protection)

La plupart de ces appels système supportent, ou sont supportés par, des concepts et fonctions qui seront discutés dans les chapitres suivants.

Outil de traçage: strace

strace est un outil de diagnostic qui permet de tracer les appels système effectués par un programme.

Utilité:

- Visualiser les appels système utilisés par les commandes
- Comprendre le lien entre programmes utilisateur et noyau
- Déboguer et analyser le comportement des programmes

Syntaxe de base:

- `strace commande` - Tracer tous les appels système
- `strace -e appel1,appel2 commande` - Filtrer des appels spécifiques
- `strace -c commande` - Afficher un résumé statistique
- `strace -o fichier.txt commande` - Sauvegarder le résultat

1. Contrôle des processus

Les appels système de contrôle des processus permettent de gérer le cycle de vie et les propriétés des processus.

Appels système principaux:

- `fork()`, `clone()` - Créer un processus
- `exit()`, `exit_group()` - Terminer un processus
- `execve()` - Charger et exécuter un programme
- `getpid()`, `nice()` - Obtenir/définir les attributs
- `wait()`, `waitpid()` - Attendre un événement
- `kill()`, `signal()` - Signaler un événement
- `brk()`, `mmap()` - Allouer/libérer mémoire

Exemple: Un shell qui lance une commande utilise `fork()` pour créer un processus enfant, puis `execve()` pour charger et exécuter le programme.

Exemple strace: Contrôle des processus

Commande: Résumé statistique des appels système

```
$ strace -c sh -c "echo Bonjour"
```

Sortie (extrait):

```
% time     seconds  calls  syscall
------ ----------- ----- ----------
 18.75    0.0001       2  clone
 15.63    0.0001       1  execve
 12.50    0.0001       3  wait4
 9.38    0.0001        1  exit_group
------ ----------- ----- ----------
100.00    0.0008      45  total
```

Observation: Création (`clone`), exécution (`execve`), attente (`wait4`), et terminaison (`exit_group`) des processus.

2. Gestion des fichiers

Les appels système de gestion des fichiers permettent de manipuler les fichiers sur le système de stockage.

Appels système principaux:

- `creat()`, `open()` - Créer/ouvrir un fichier
- `unlink()`, `remove()` - Supprimer un fichier
- `close()` - Fermer un fichier
- `read()`, `write()` - Lire/écrire
- `lseek()` - Repositionner le pointeur
- `stat()`, `chmod()` - Obtenir/définir attributs

Exemple: Un éditeur de texte utilise `open()` pour ouvrir un fichier, `read()` pour lire son contenu, `write()` pour sauvegarder les modifications, et `close()` pour fermer le fichier.

Exemple strace: Gestion des fichiers

Démonstration: Tracer les opérations fichier avec `cat`

```
$ echo "Bonjour" > fichier.txt
$ strace -e openat,read,close cat fichier.txt
```

Sortie:

```
openat(AT_FDCWD, "fichier.txt", O_RDONLY) = 3
read(3, "Bonjour\n", 131072) = 8
Bonjour
close(3) = 0
+++ exited with 0 +++
```

Observation: `openat()` ouvre le fichier (descripteur 3), `read()` lit 8 octets, `close()` ferme le fichier.

3. Gestion des périphériques

Les appels système de gestion des périphériques permettent d'interagir avec les dispositifs matériels.

Appels système principaux:

- `open()` - Demander un périphérique
- `close()` - Libérer un périphérique
- `read()`, `write()` - Lire/écrire sur périphériques
- `lseek()` - Repositionner sur périphériques
- `ioctl()` - Obtenir/définir attributs périphérique
- `mount()`, `umount()` - Attacher/détacher logiquement

Exemple: Un programme d'impression demande accès à l'imprimante via `open("/dev/lp0")`, écrit les données avec `write()`, puis libère l'imprimante avec `close()`.

Exemple strace: Gestion des périphériques

Démonstration: Tracer l'accès aux périphériques

```
$ strace -e openat,ioctl ls /dev/null
```

Sortie pertinente:

```
openat(AT_FDCWD, "/dev/null", O_RDONLY|...) = 3
ioctl(1, TCGETS, {...}) = 0
ioctl(1, TIOCGWINSZ, {...}) = 0
```

Observation: `openat()` accède au périphérique `/dev/null`, `ioctl()` contrôle les attributs du terminal. Les périphériques sont traités comme des fichiers spéciaux sous Linux.

4. Maintenance de l'information

Les appels système de maintenance permettent d'obtenir et de modifier des informations système.

Appels système principaux:

- `time()`, `gettimeofday()` - Obtenir date/heure
- `uname()`, `sysinfo()` - Obtenir/définir données système
- `getpid()`, `getppid()` - Obtenir attributs processus
- `stat()`, `fstat()` - Obtenir attributs fichier
- `ioctl()` - Obtenir attributs périphérique
- `chmod()`, `chown()` - Définir attributs

Exemple: La commande `ls -l` utilise `stat()` pour obtenir les informations détaillées (taille, permissions, date de modification) de chaque fichier avant de les afficher.

Exemple strace: Maintenance de l'information

Démonstration: Tracer `uname -a` (informations système)

```
$ strace -e uname uname -a
```

Sortie:

```
uname({sysname="Linux", nodename="pc",
 release="5.15.0", version="#1 SMP",
 machine="x86_64"}) = 0
Linux pc 5.15.0 #1 SMP x86_64 GNU/Linux
+++ exited with 0 +++
```

Observation: Un seul appel `uname()` retourne toutes les informations du système: nom du noyau, nom de la machine, version, et architecture.

5. Communications

Les appels système de communication permettent l'échange d'informations entre processus.

Appels système principaux:

- `socket()`, `pipe()` - Créer connexion
- `close()` - Supprimer connexion
- `send()`, `write()`, `sendto()` - Envoyer messages
- `recv()`, `read()`, `recvfrom()` - Recevoir messages
- `getsockopt()`, `setsockopt()` - Transférer statut
- `connect()`, `bind()` - Attacher/détacher périphériques distants

Exemple: Un serveur web utilise `socket()` pour créer un socket, `bind()` pour l'associer à un port, `listen()` pour écouter, `accept()` pour accepter des connexions, et `send()`/`recv()` pour échanger des données HTTP.

Exemple strace: Communications

Démonstration: Tracer une résolution DNS

```
$ strace -e socket,connect getent hosts www.google.com
```

Sortie (extrait):

```
socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC, 0) = 3
connect(3, {sa_family=AF_UNIX,
 sun_path="/var/run/nscd/socket"}, 110) = 0
142.250.200.132  www.google.com
+++ exited with 0 +++
```

Observation: `socket()` crée un socket Unix, `connect()` se connecte au service DNS local pour résoudre le nom de domaine.

6. Protection

Les appels système de protection permettent de contrôler l'accès aux ressources système.

Appels système principaux:

- `stat()`, `access()` - Obtenir permissions fichier
- `chmod()`, `chown()`, `chgrp()` - Définir permissions fichier

Permissions sous UNIX/Linux:

- Lecture (r), écriture (w), exécution (x)
- Pour propriétaire, groupe, et autres
- Exemple: `chmod 755 fichier.txt`

Exemple: Lorsqu'un utilisateur exécute `chmod 600 document.txt`, le système utilise l'appel `chmod()` pour définir les permissions (rw——-), protégeant ainsi le fichier.

Exemple strace: Protection

Démonstration: Tracer les modifications de permissions

```
$ touch fichier_test.txt
$ strace -e chmod,fchmodat chmod 600 fichier_test.txt
```

Sortie:

```
fchmodat(AT_FDCWD, "fichier_test.txt", 0600) = 0
+++ exited with 0 +++
```

Observation: `fchmodat()` modifie les permissions du fichier. Le mode 0600 signifie: lecture/écriture pour le propriétaire uniquement (rw——-).

Ressources et Références

Documentation complète :

- Liste des appels système Linux :
  https://man7.org/linux/man-pages/man2/syscalls.2.html
- Documentation des API POSIX :
  https://man7.org/linux/man-pages/dir_section_3.html
- Pages de manuel Linux (man7.org) :
  https://man7.org/linux/man-pages/

Ouvrage de référence :

- Ce cours est basé sur l'ouvrage :
  Operating System Concepts (10^{e} édition)
  Abraham Silberschatz, Peter B. Galvin, Greg Gagne
- Site web du livre : https://os-book.com