



université
virtuelle
Burkina ★ Faso

Conception et programmation Système

— Niveau : Licence 2-Génie Logiciel —

Réalisé par: Abdoul Aziz BONKOUNGOU

Informations générales

1. Durée

- Cours théorique: 24 heures
- Travaux dirigés: 6 heures
- Travaux pratiques: 6 heures

2. Prérequis

- Connaissances de base en programmation et architecture des systèmes informatiques

3. Ressources:

<https://github.com/azizYaaba/Programmation-Syst-me.git>

Plan

1. Introduction à la programmation système

- Concepts de base de la programmation système
- Distinction entre programmation système et programmation applicative
- Environnements de développement pour la programmation système

2. Architecture des systèmes d'exploitation et interaction

- Structure d'un système d'exploitation
- Appels système et interface avec le noyau
- Gestion des interruptions et du temps

3. Gestion des processus

- Création et terminaison de processus
- Communication entre processus (IPC) : pipes, files de messages,
- mémoire partagée, sockets
- Gestion de la synchronisation des processus (sémaphores, mutex)

4. Gestion de la mémoire

- Allocation et libération dynamique de mémoire
- Segmentation et pagination
- Gestion de la mémoire virtuelle

5. Programmation multi-thread

- Création et gestion de threads
- Synchronisation des threads
- Problèmes courants : deadlock, starvation

6. Gestion des fichiers et systèmes de fichiers

- Manipulation des fichiers (ouverture, lecture, écriture, fermeture)
- Organisation des systèmes de fichiers
- Permissions et gestion de la sécurité des fichiers

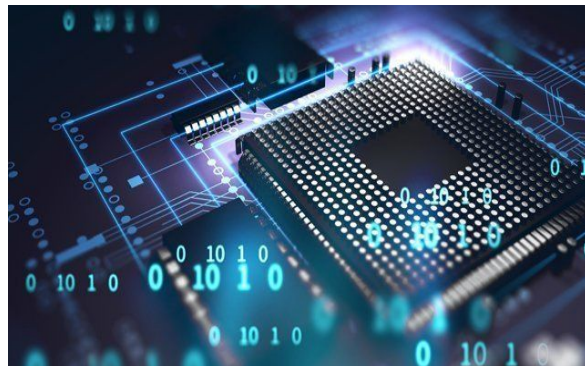
7. Optimisation et performances en programmation système

- Techniques d'optimisation de code
- Utilisation des buffers et caches
- Introduction au benchmarking

Introduction à la programmation système

Qu'est-ce que la programmation système ?

- Définition : La programmation système est l'ensemble des techniques et outils qui permettent d'écrire des logiciels interagissant directement avec le système d'exploitation (OS) et, par son intermédiaire, avec le matériel.
- Elle vise à fournir des services fondamentaux aux applications et à gérer efficacement les ressources matérielles (CPU, mémoire, périphériques).

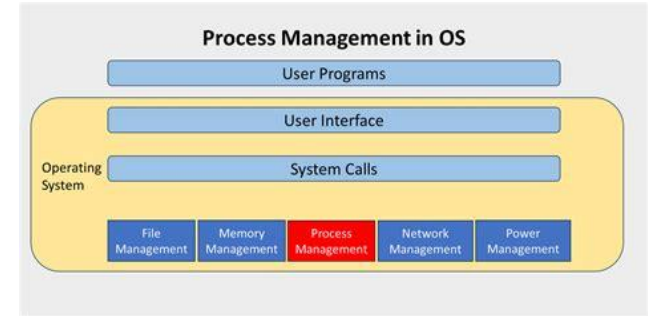


Caractéristiques de la programmation système

- **Bas niveau** : proche du matériel, accès direct aux ressources
- **Dépendante de l'OS** : utilisation des API et appels système spécifiques (POSIX, Win32)
- **Performance** : recherche d'efficacité, minimisation des surcoûts
- **Précision** : nécessite une bonne compréhension de l'architecture et du fonctionnement de l'OS

Rôle du système d'exploitation

- Le système d'exploitation est un logiciel de contrôle qui gère :
 - Le processeur : planification des tâches
 - La mémoire : allocation et protection
 - Les périphériques : abstraction via des pilotes
 - Les fichiers : gestion et sécurité des accès
- La programmation système permet d'interagir avec ces services.



Les appels système (syscalls)

- Définition : Mécanisme par lequel un programme utilisateur demande un service au noyau de l'OS.
- Ils constituent l'interface entre l'espace utilisateur et l'espace noyau.
- Exemples d'appels système POSIX :
 - `fork()` → création de processus
 - `read()` / `write()` → accès aux fichiers ou périphériques
 - `exec()` → exécution d'un nouveau programme

Exemple d'appel système

```
#include <unistd.h>
int main() {
    write(1, "Bonjour système\n", 16);
    return 0;
}
```

- Ici, write() envoie directement la chaîne de caractères vers la sortie standard (console) via le noyau.
- Contrairement à printf, aucune bibliothèque de haut niveau n'est utilisée.

Concept fondamental : Processus

- Définition : Un processus est un programme en cours d'exécution, identifié par un PID (Process Identifier).
- Il possède :
 - Son espace mémoire
 - Son contexte d'exécution (registres, pile, variables)
 - Ses ressources (fichiers ouverts, périphériques)
 - Importance : isolation, multitâche, sécurité.

Concept fondamental : Threads

- Définition : Un thread est une unité légère d'exécution à l'intérieur d'un processus.
- Plusieurs threads peuvent exécuter des parties différentes du même programme en parallèle.
- Ils partagent :
 - La mémoire du processus parent
 - Les fichiers ouverts
- Avantage : rapidité, parallélisme → utile pour serveurs, calculs intensifs.

Gestion de la mémoire

- La mémoire est une ressource critique, gérée par l'OS.
- Mécanismes :
 - Allocation dynamique (malloc/free en C)
 - Segmentation : séparation en zones logiques
 - Pagination : division en pages de taille fixe, avec translation par la MMU
- Protection : chaque processus a un espace mémoire isolé pour éviter les corruptions.

Entrées/Sorties (I/O)

- En programmation système, tous les périphériques (disques, claviers, écrans) sont vus comme des fichiers.
- Exemple :
 - `open("fichier.txt", O_RDONLY);`
 - `read(fd, buffer, size);`
 - `write(fd, buffer, size);`
- Avantage : interface unifiée pour l'accès aux ressources.

Programmation système vs applicative

Programmation Système	Programmation Applicative
Proche du matériel	Proche de l'utilisateur
Manipule processus, mémoire, fichiers, périphériques	Manipule interfaces, données métier, logique applicative
Langages : C, C++, Rust, assembleur	Langages : Java, Python, C#, etc.
Objectif : efficacité, contrôle	Objectif : productivité, fonctionnalités

Exemple comparatif

- Programme applicatif : éditeur de texte écrit en Python.
- Programme système : routines `read()` et `write()` utilisées par Python pour ouvrir et écrire dans des fichiers.
- L'applicatif repose entièrement sur le système.

Pourquoi apprendre la programmation système ?

- Comprendre les fondements du fonctionnement des logiciels
- Développer des logiciels optimisés et fiables
- Pouvoir manipuler et analyser des problèmes système complexes (débogage, performance)
- Indispensable pour : systèmes embarqués, développement d'OS, logiciels critiques

Langages utilisés

- C : langage de référence, norme POSIX, accès direct aux syscalls
- C++ : ajoute abstractions, orienté objets, mais garde la performance
- Assembleur : utilisé pour écrire des parties critiques (pilotes, bootloader)
- Rust : moderne, évite les erreurs mémoire grâce à son système de propriété

Environnements de développement : Linux

- Compilateurs : GCC, Clang
- Débogueur : GDB
- Analyse : strace (suivi des syscalls), perf (profilage)
- Automatisation : Make, CMake
- Linux est l'environnement privilégié en enseignement et recherche.

Environnements de développement : Windows

- Visual Studio : IDE complet pour C/C++
- MinGW / Cygwin : outils POSIX sous Windows
- Win32 API : interface native pour interagir avec le système
- Moins utilisé académiquement que Linux, mais incontournable en industrie.

Exemple pratique sous Linux

- Compilation et exécution :

```
gcc prog.c -o prog  
./prog
```

- Analyse des appels système :

```
strace ./prog
```

strace montre chaque syscall effectué par le programme.

Outils essentiels

- Éditeur de code : Vim, Emacs, VS Code
- Gestion de versions : Git
- Débogueurs/Profilers : GDB, Valgrind, perf
- Shells : bash, zsh → pour exécuter et tester les programmes système

Défis de la programmation système

- Complexité : manipulation directe des ressources → erreurs fréquentes
- Portabilité : dépendance forte à l'OS
- Sécurité : bugs peuvent exposer tout le système
- Courbe d'apprentissage : nécessite de solides bases en architecture et OS

Récapitulatif

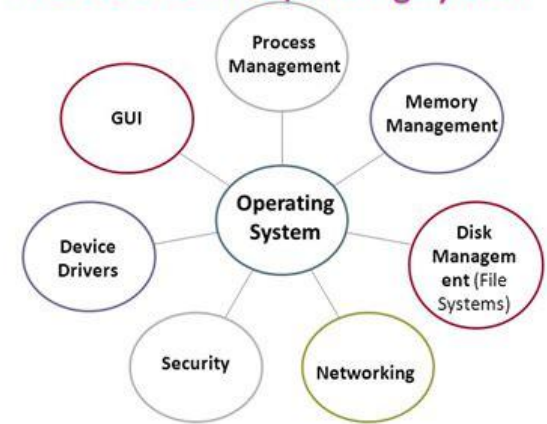
- La programmation système = interaction directe avec l'OS et le matériel
- Concepts clés : processus, threads, mémoire, I/O, appels système
- Différence majeure avec la programmation applicative
- Environnements principaux : Linux, Windows
- Outils : compilateurs, débogueurs, analyseurs, shells

Architecture des systèmes d'exploitation et interaction

Structure d'un système d'exploitation

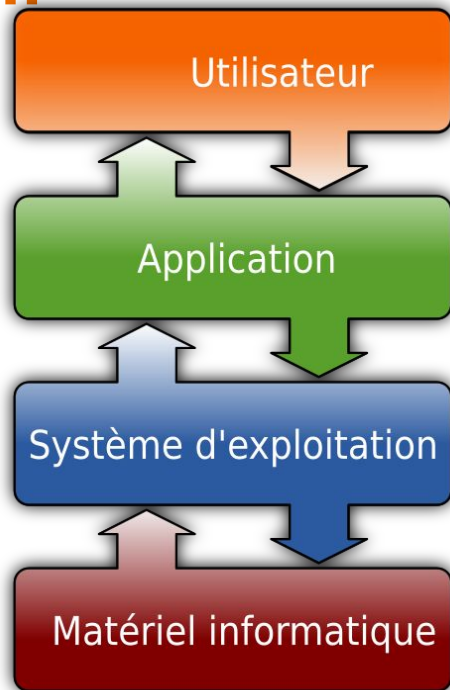
- **Définition** : Un système d'exploitation (OS) est un logiciel qui agit comme un intermédiaire entre le matériel et les applications.
- **Fonctions principales** :
 - Gestion des ressources (CPU, mémoire, périphériques)
 - Fourniture de services aux applications
 - Isolation et sécurité

Functions of an Operating System



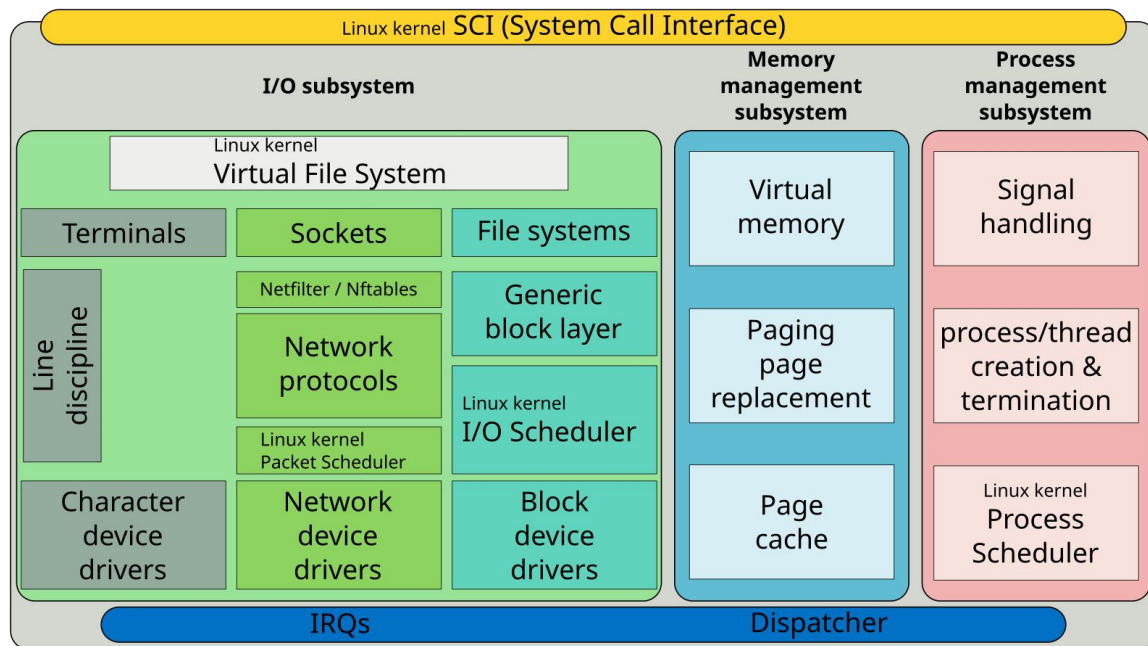
Les couches d'un système d'exploitation

1. Matériel : CPU, mémoire, périphériques
2. Noyau (kernel) : gestion directe du matériel
3. Appels système (syscalls) : interface entre noyau et programmes
4. Bibliothèques système : abstraction des syscalls
5. Applications : logiciels utilisateur



Organisation en couches

- Un OS est souvent structuré en couches hiérarchiques :
 - Bas niveau → accès direct au matériel
 - Haut niveau → services pour l'utilisateur
- Avantage : modularité et maintenance facilitée



Types d'architectures de noyaux

1. Monolithique :

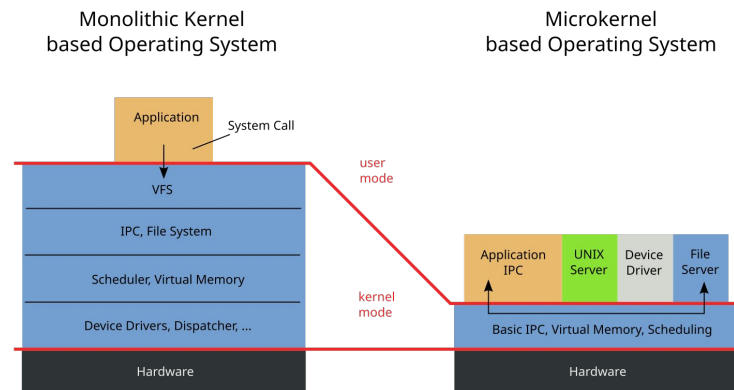
- Tout le noyau fonctionne en espace noyau
- Ex. : Linux, Unix traditionnel

2. Micro-noyau :

- Fonctions minimales dans le noyau (IPC, gestion mémoire)
- Services en espace utilisateur
- Ex. : Minix, QNX

3. Hybride :

- Combinaison des deux
- Ex. : Windows NT, macOS



Comparaison des architectures

Monolithique	Micro-noyau	Hybride
Rapide	Plus sûr, modulable	Compromis
Complexe à maintenir	Surcoût de performance	Plus flexible que monolithique
Exemple : Linux	Exemple : Minix	Exemple : Windows NT

Appels système et interface avec le noyau

Définition des appels système

- Appel système (syscall) : fonction fournie par le noyau permettant à un programme utilisateur d'accéder à une ressource matérielle ou à un service système.
- Interface essentielle entre espace utilisateur et espace noyau.

Exemple d'appel système

Lecture d'un fichier en C :

```
int fd = open("fichier.txt", O_RDONLY);  
read(fd, buffer, taille);  
close(fd);
```

Ces fonctions → appels système POSIX traduits en instructions noyau.

Mécanisme d'un syscall

1. Programme appelle une fonction de la bibliothèque C (libc)
2. Instruction spéciale (trap, interrupt) pour passer en mode noyau
3. Le noyau exécute la fonction correspondante
4. Résultat renvoyé au programme en mode utilisateur

Espace utilisateur vs espace noyau

- Espace utilisateur : applications → pas d'accès direct au matériel
- Espace noyau : OS → accès complet au matériel
- Les syscalls assurent la communication entre les deux espaces

Gestion des interruptions

Définition d'une interruption

- **Définition** : signal envoyé au processeur pour indiquer un événement nécessitant une réaction immédiate.
- **Types** :
 - Interruptions matérielles : clavier, disque, carte réseau
 - Interruptions logicielles : générées par un programme (ex. appel système)

Cycle d'une interruption

1. Périphérique déclenche une interruption
2. CPU sauvegarde le contexte du programme en cours
3. Le noyau exécute la routine de traitement d'interruption (ISR)
4. Retour au programme initial

Interruptions matérielles

Exemples :

- Appui sur une touche → signal du clavier
- Données disponibles sur le disque → signal du contrôleur
- Avantage : évite le polling (attente active du CPU)

Interruptions logicielles

- Déclenchées par un programme, souvent via une instruction spéciale
- Exemple : l'instruction `int 0x80` en assembleur sur Linux (anciens systèmes) pour exécuter un `syscall`
- Permet la transition en mode noyau

Gestion du temps

Rôle de la gestion du temps

- L'OS doit gérer le temps pour :
- Ordonnancement des processus
- Mesure de la consommation CPU
- Mise en veille et temporisation
- Suivi de l'horloge système

Timer matériel

- Un timer génère des interruptions à intervalles réguliers
- Permet à l'OS de reprendre la main et de basculer entre processus
- Exemple : toutes les 10 ms, interruption → scheduler appelé

Gestion du temps dans l'OS

- Structures utilisées :
 - Horloge système : temps réel
 - Compteurs de ticks : nombre d'interruptions timer
- Utilisé pour :
 - sleep() → mise en pause d'un processus
 - Mesure de la performance

Exemple en C (temporisation)

```
#include <unistd.h>
int main() {
    write(1, "Début\n", 6);
    sleep(2); // pause de 2 secondes
    write(1, "Fin\n", 4);
    return 0;
}
```

Utilisation du timer pour suspendre l'exécution

Récapitulatif

- Un OS est structuré en couches : matériel, noyau, syscalls, bibliothèques, applications
- Les appels système assurent la communication entre programmes et noyau
- Les interruptions permettent de réagir aux événements internes et externes
- La gestion du temps est essentielle pour l'ordonnancement et la synchronisation

Gestion des processus

Définition d'un processus

Un processus = un programme en cours d'exécution, identifié par un PID (Process Identifier). C'est l'unité de base de l'ordonnancement dans un système d'exploitation.

Composition d'un processus

Un processus possède :

1. **Un espace mémoire**

- Code, pile, tas, variables globales.

2. **Un contexte d'exécution**

- Registres CPU, pointeur d'instruction, pointeur de pile.

3. **Des ressources**

- Fichiers ouverts, périphériques, sockets.

Cycle de vie d'un processus

Un processus passe par plusieurs états :

- **Création** → via `fork()`.
- **Exécution** (Running) → quand le CPU lui est alloué.
- **Suspendu** (Waiting/Blocked) → en attente d'une ressource (I/O, signal).
- **Prêt** (Ready) → en attente du CPU.
- **Terminé** (Zombie) → fini, mais pas encore "ramassé" par le père (`wait()`).

Commandes utiles :

- `ps -o pid,ppid,stat,cmd` → voir les états.
- `top` → suivi en temps réel

États principaux

1. **Nouveau (New)** → processus créé (via fork()).
2. **Prêt (Ready)** → en attente d'un CPU.
3. **En cours d'exécution (Running)** → instructions exécutées par le CPU.
4. **Bloqué (Waiting/Blocked)** → en attente d'une ressource (I/O, signal).
5. **Terminé (Terminated)** → fini, mémoire libérée.
6. **Zombie** → fini mais pas encore nettoyé par wait().

Transition :

- fork() → New → Ready.
- L'ordonnanceur choisit → Running.
- Si attente d'I/O → Waiting.
- Quand terminé → Zombie → cleanup avec wait().

Création de processus avec fork()

- En C sous Unix/Linux, un processus est créé avec :

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Je suis le processus fils, PID=%d\n", getpid());
    } else {
        printf("Je suis le processus père, PID=%d, fils=%d\n", getpid(), pid);
    }
    return 0;
}
```

- fork() crée un clone du processus courant.

Exécution avec exec()

- La famille exec remplace l'image mémoire du processus par un nouveau programme.

```
#include <unistd.h>
int main() {
    execl("/bin/ls", "ls", "-l", NULL);
    return 0; // exécuté uniquement si exec échoue
}
```

- Exemple : un shell utilise fork() + exec() pour exécuter des commandes.

Terminaison et attente d'un processus

- Un processus se termine par :
 - `exit(status)` → termine immédiatement
 - Retour de `main()`
- Le père récupère le code de retour avec `wait()` ou `waitpid()`

```
#include <sys/wait.h>
```

```
int status;
```

```
wait(&status); // bloquant, attend la fin du fils
```

Terminaison et attente d'un processus

`wait(&status)`

- Bloque le père jusqu'à ce qu'un fils se termine.
- Met le code de retour dans `status`.

`waitpid(pid, &status, 0)`

- Variante plus fine : on attend un fils précis (`pid`).
- Très utile quand le père a plusieurs enfants.

`WIFEXITED(status)` → vrai si le fils s'est terminé normalement.

`WEXITSTATUS(status)` → récupère la valeur passée à `exit()` → code de retour

Commande utiles

Visualisation avec ps et top : Lister les processus actifs

```
ps -o pid,ppid,stat,cmd
```

- PID = identifiant du processus.
- PPID = identifiant du père.
- STAT = état du processus (R, S, T, Z).
- CMD = commande associée.

Exemple :

PID	PPID	STAT	CMD
1234	1220	S	./monprog

Surveiller en temps réel

- **Suivi en temps réel**

- **Top**

- %CPU → charge processeur du processus.
 - %MEM → mémoire utilisée.
 - STAT → état actuel (R = running, S = sleeping, Z = zombie...).

- **Suspendre et reprendre avec signaux :**

- `kill -STOP <PID>` # processus passe à l'état T (stopped)
 - `kill -CONT <PID>` # processus reprend

- **Terminer**

- `kill -KILL <PID>`

Importance

- Permet le multitâche.
- Garantit l'isolation (mémoire protégée).
- Supporte la sécurité (droits, permissions).
- Base pour la communication inter-processus (IPC)

À vous de jouer

- TP 1:À la découverte des processus(1H)
- TP 2: Etat d'ordonnancement(1H)
- TD1 – Processus (création et ordonnancement)

Communication inter-processus (IPC)

Définition

La communication inter-processus (IPC, Inter-Process Communication) regroupe l'ensemble des mécanismes qui permettent à des processus distincts d'échanger des informations.

Pourquoi l'IPC ?

- Chaque processus a son espace mémoire isolé (protection OS).
- Pour échanger des données ou coopérer, ils doivent passer par des mécanismes fournis par l'OS.
- C'est indispensable dans :
 - un système multitâche (serveur web gérant plusieurs clients),
 - une application parallèle (plusieurs processus traitant un même problème),
- l'interaction utilisateur (shell ↔ commandes).

Communication inter-processus (IPC)

Introduction à l'IPC

- Les processus sont isolés par l'OS.
- Pour coopérer, ils utilisent des mécanismes IPC :
 - Pipes
 - Files de messages
 - Mémoire partagée
 - Sockets

Les pipes

- Définition : canal de communication unidirectionnel entre processus.
 - un processus écrit à une extrémité
 - un autre lit à l'autre extrémité.
- Exemple : communication père → fils

Très utilisés dans les commandes shell

```
ls -l | grep ".c"
```

- `ls -l` écrit dans un pipe, `grep` lit depuis ce pipe

Trois grands types

1. Pipe du shell |
2. Pipes anonymes (pipe()) en C)
3. Pipes nommés (FIFOs)

Pipe du shell |

Utilisé dans bash/zsh/sh.

- Syntaxe : `commande1 | commande2`



Redirige la stdout de `commande1` vers la stdin de `commande2`.

Exemple : `ls -l | grep ".c" | wc -l`

- `ls -l` → liste fichiers
- `grep ".c"` → filtre fichiers .c
- `wc -l` → compte le nombre de lignes

Points clés :

- Simple à utiliser.
- Chaînage de plusieurs programmes.
- Communication éphémère (pas de stockage)

Pipes anonymes

Créés avec l'appel système

```
int fd[2];  
  
pipe(fd); // fd[0] = lecture,  
fd[1] = écriture
```

- Utilisés entre processus apparentés (père ↔ fils).
- Communication unidirectionnelle.

```
char buffer[100];  
int fd[2];  
pipe(fd);  
  
if (fork() == 0) {  
    // Fils lit  
    read(fd[0], buffer,  
        sizeof(buffer));  
    printf("Fils a lu : %s\n",  
        buffer);  
} else {  
    // Père écrit  
    write(fd[1], "Bonjour fils",  
        13);  
}
```

Points clés :

- Mécanisme rapide (mémoire du noyau).
- Disparaît quand les processus se terminent.
- Uniquement pour processus liés.

Pipes nommés

Fichiers spéciaux créés avec :mkfifo canal

- Permettent la communication entre **processus indépendants**.
- Visible dans le système de fichiers (type p dans ls -l)

Exemple

Terminal 1

```
echo "Salut" > canal
```

Terminal 2

```
cat canal
```



Le message est transmis via le pipe nommé.

Points clés :

- Persistants (existe comme un fichier spécial).
- Plusieurs processus peuvent y accéder.

Comparaison des types

Type	Contexte d'utilisation	Lien entre processus	Persistance
Pipe du shell		Commandes en ligne de commande	Aucun lien, juste chaînage
Pipe anonyme	Entre processus apparentés (père-fils)	Obligatoire	Temporaire
Pipe nommé (FIFO)	Entre processus indépendants	Aucun lien requis	Persiste dans /dev ou fichier spécial

Files de message

Les files de messages sont un mécanisme de communication entre processus (IPC).

- Fonctionnement comparable à une boîte aux lettres :
 - Les processus envoient des messages.
 - D'autres processus peuvent les lire plus tard.
- Communication asynchrone → l'expéditeur et le récepteur n'ont pas besoin de s'exécuter au même moment.

Caractéristiques

Chaque message a :

- un type (long),
- un contenu (texte, structure).

Permet à plusieurs processus de trier les messages selon le type.

- Messages stockés par le noyau jusqu'à lecture.
- Utilisés dans des systèmes plus anciens mais encore très instructifs.

Exemple

```
#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

struct msg { long type; char text[100]; };

int main() {

    // Création ou récupération d'une file (ID=1234)

    int qid = msgget(1234, IPC_CREAT | 0666);

    // Envoi d'un message de type 1

    struct msg m = {1, "Hello"};

    msgsnd(qid, &m, sizeof(m.text), 0);

    // Réception d'un message de type 1

    msgrcv(qid, &m, sizeof(m.text), 1, 0);

    printf("Reçu: %s\n", m.text);

}
```

Commandes Unix utiles

- Lister les files de messages actives :

`ipcs -q`

- Supprimer une file de messages


`ipcrm -q <ID>`

Mémoire partagée

La mémoire partagée est un mécanisme d'IPC (communication inter-processus).

- Principe : plusieurs processus peuvent accéder directement au même segment mémoire.
- Très efficace → pas de copies entre processus.

Caractéristiques

- Segment mémoire alloué par le noyau.
- Identifié par un ID (shmid).
- Accessible via shmat() (attach) et shmdt() (detach).
- Les processus doivent se mettre d'accord sur l'ID (ou clé IPC).
-  Problème potentiel : concurrence → nécessite synchronisation (sémaphores, mutex).

Exemple

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <stdio.h>

int main() {
    // Création ou récupération d'un segment de 1024 octets
    int shmid = shmget(1234, 1024, IPC_CREAT | 0666);

    // Attacher le segment
    char *data = shmat(shmid, NULL, 0);

    // Écrire dans la mémoire partagée
    strcpy(data, "Bonjour mémoire partagée !");

    // Détacher le segment
    shmdt(data);

    return 0;
}
```

Commandes Unix utiles

Lister les segments mémoire partagée :

```
ipcs -m
```

Supprimer un segment :

```
ipcrm -m <ID>
```

Avantages & inconvénients

✓ Avantages :

- Communication très rapide (pas de copie).
- Idéal pour gros volumes de données.

⚠ Inconvénients :

- Besoin de synchronisation (éviter les accès concurrents).
- Plus complexe à gérer que les pipes ou files de messages.
- Ressources doivent être libérées (shmdt, ipcrm).

Les sockets

- Un socket est un mécanisme de communication entre processus.
- Fonctionne aussi bien :
 - Localement (même machine) → sockets UNIX.
 - À distance (machines différentes) → sockets TCP/IP.
- Base de la communication réseau (tous les serveurs : web, SSH, FTP, etc.).

Caractéristiques

- Plus généraux que pipes ou mémoire partagée.
- Supportent :
 - Mode flux (TCP, fiable, orienté connexion).
 - Mode datagramme (UDP, non fiable, rapide).
- Permettent la communication entre processus non liés.
- Identifiés par une adresse (fichier dans /tmp pour UNIX, IP:port pour TCP/IP).

Exemple : socket UNIX locale

```
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
```

```
int main() {
    int sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock < 0) perror("socket");
    else printf("Socket créée: %d\n", sock);
    return 0;
}
```

- AF_UNIX → communication locale (fichier spécial).
- SOCK_STREAM → communication fiable (comme TCP).
- socket() retourne un descripteur de fichier utilisable avec read/write.

Exemple TCP client

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int sock = socket(AF_INET, SOCK_STREAM, 0); // TCP socket
    struct sockaddr_in server = {0};
    server.sin_family = AF_INET;
    server.sin_port = htons(8080);
    server.sin_addr.s_addr = inet_addr("127.0.0.1");

    connect(sock, (struct sockaddr*)&server, sizeof(server));
    write(sock, "Hello serveur", 13);
    close(sock);
    return 0;
}
```

Exemple d'utilisation

- Serveur crée une socket, l'associe à une adresse (bind), attend un client (listen, accept).
- Client se connecte au serveur (connect).
- Ensuite : communication bidirectionnelle avec send et recv.

Synchronisation des processus

Pourquoi synchroniser ?

- Problèmes possibles :
 - Conditions de course → plusieurs processus accèdent à une ressource en même temps.
 - Incohérences de données.
- Solution : mécanismes de synchronisation.

Sémaphores

- Un sémaphore est un compteur qui contrôle combien de threads (ou processus) peuvent accéder à une ressource partagée en même temps.
 - Si le compteur > 0 → un thread peut entrer dans la section critique (le compteur est décrémenté).
 - Si le compteur $= 0$ → les autres threads doivent attendre.

- Exemple avec POSIX semaphores :

```
#include <semaphore.h>
#include <pthread.h>
```

```
sem_t sem;
void* routine(void* arg) {
    sem_wait(&sem);
    printf("Section critique\n");
    sem_post(&sem);
    return NULL;
}

int main() {
    sem_init(&sem, 0, 1);
    pthread_t t1, t2;
    pthread_create(&t1, NULL, routine, NULL);
    pthread_create(&t2, NULL, routine, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

Mutex

- Définition : verrou binaire, utilisé pour protéger une section critique.
- Exemple avec pthread_mutex

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

pthread_mutex_t lock;

void* task(void* arg) {
    pthread_mutex_lock(&lock);
    printf("Accès protégé par mutex\n");
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_mutex_init(&lock, NULL);
    pthread_t t1, t2;
    pthread_create(&t1, NULL, task, NULL);
    pthread_create(&t2, NULL, task, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

Sémaphores vs Mutex

Sémaphore	Mutex
Compteur entier	Binaire (0 ou 1)
Plusieurs accès autorisés (selon compteur)	Exclusif (1 seul accès)
Utilisé pour gérer files d'attente, producteur/consommateur	Utilisé pour protéger section critique

Problème classique : Producteur/Consommateur

- Un producteur écrit dans un tampon.
- Un consommateur lit dans ce tampon.
- Nécessite synchronisation :
 - Mutex pour accès au tampon
 - Sémaphores pour compter les places libres/occupées

Exemple simplifié Producteur/Consommateur

```
sem_t vide, plein;  
pthread_mutex_t lock;  
int buffer;  
  
void* producteur(void* arg) {  
    sem_wait(&vide);  
    pthread_mutex_lock(&lock);  
    buffer = rand()%100;  
    printf("Produit: %d\n", buffer);  
    pthread_mutex_unlock(&lock);  
    sem_post(&plein);  
}
```


Deadlocks (blocages mutuels)

- Situations où deux processus attendent indéfiniment une ressource détenue par l'autre.
- Ex. :
 - Processus A verrouille R1 et attend R2
 - Processus B verrouille R2 et attend R1
 - Solutions : ordonnancement strict, prévention, détection.

Résumé

- Création → `fork()`, `exec()`, `exit()`
- Communication → pipes, files de messages, mémoire partagée, sockets

Mécanisme	Avantage	Limite
Pipe	Simple, efficace	Unidirectionnel
File de messages	Messages typés, file persistante	Plus lent
Mémoire partagée	Très rapide, pas de copie	Synchronisation complexe
Socket	Local + réseau	Configuration lourde

- Synchronisation → sémaphores, mutex, gestion de concurrence
- Risques → conditions de course, deadlocks

À vous de jouer

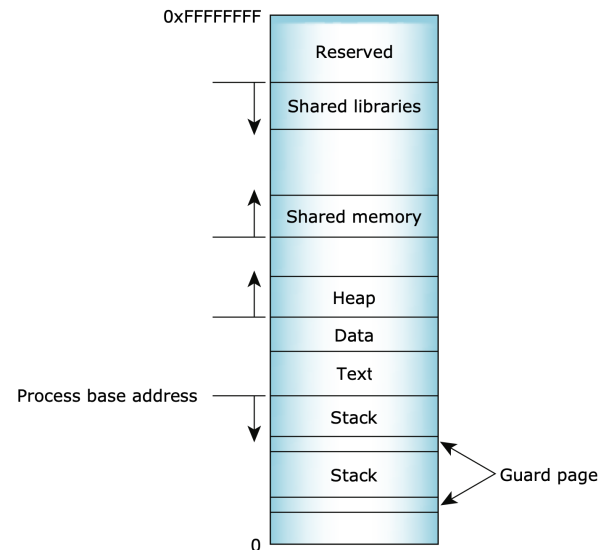
- TP3: IPC
- TP4: synchronisation
- TD 2:IPC

Gestion de la mémoire

Allocation et libération dynamique de mémoire

Définition

- La mémoire d'un processus est divisée en plusieurs zones :
 - **Pile (stack)** : variables locales, appels de fonctions
 - **Tas (heap)** : mémoire dynamique, allouée à l'exécution
 - **Segment données** : variables globales, constantes
 - **Segment code (text)** : instructions du programme



Gestion de la mémoire en programmation

Introduction

- Chaque processus pense qu'il dispose d'un grand espace mémoire contigu (par ex. 4 Go sur une machine 32 bits).
- Il « voit » sa propre organisation (code, données, tas, pile) comme si la mémoire lui appartenait exclusivement
- Ce qui se passe en réalité
 - Le système d'exploitation (OS) et la MMU (Memory Management Unit) du processeur traduisent les adresses virtuelles du processus en adresses physiques (RAM réelle).
 - Plusieurs processus peuvent donc tourner en même temps sans se marcher dessus.
 - Chaque accès mémoire passe par cette traduction → c'est transparent pour le programmeur

Avantages principaux

1. Simplicité pour le programmeur
 - a. Pas besoin de gérer où placer les données en mémoire physique.
 - b. Chaque programme croit être seul en mémoire.
2. Sécurité (isolation)
 - a. Un processus ne peut pas lire/écrire la mémoire d'un autre.
 - b. Si un programme fait un accès illégal → segmentation fault au lieu de corrompre un autre programme.
3. Partage contrôlé
 - a. Possibilité de partager certaines zones mémoire entre processus (par exemple, avec mmap ou mémoire partagée System V).
 - b. Utile pour la communication inter-processus (IPC)

Organisation mémoire d'un processus

- Code (text) : instructions en lecture seule
- Données (data, BSS) : variables globales
- Tas (heap) : mémoire dynamique (malloc/free)
- Pile (stack) : variables locales, appels de fonctions
- Zones mappées : fichiers partagés (mmap)
- Exemple en C avec une variable globale, locale et dynamique.

Example

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

// ===== Données (data/BSS) =====
int global_init = 42;           // .data (global initialisée)
int global_uninit;              // .bss (global non initialisée)

int main(void) {
    // ===== Pile (stack) =====
    int local = 7;              // variable locale → pile

    // ===== Tas (heap) =====
    char *dyn = malloc(20 * sizeof(char)); // allocation dynamique → tas
    if (dyn == NULL) {
        perror("malloc");
        return 1;
    }
    strcpy(dyn, "Bonjour mémoire");

    // ===== Code (text) =====
    // Les instructions de main() sont exécutées depuis le segment texte.

    // Affichage des adresses pour visualiser
    printf("Adresse code (fonction main) : %p\n", (void *) main);
    printf("Adresse données init (global_init) : %p\n", (void *) &global_init);
    printf("Adresse données bss (global_uninit) : %p\n", (void *) &global_uninit);
    printf("Adresse pile (local) : %p\n", (void *) &local);
    printf("Adresse tas (malloc) : %p\n", (void *) dyn);

    free(dyn); // libération du tas
    return 0;
}
```

- Code (text) → les instructions de main() sont dans la section texte.
- Données (.data et .bss) → global_init et global_uninit.
- Tas (heap) → malloc réserve de la mémoire dynamique.
- Pile (stack) → local est stocké dans la pile.
- Zones mappées → pas illustrées ici, mais apparaissent si on utilise mmap ou si des bibliothèques dynamiques (.so) sont chargées.

Allocation dynamique

- **malloc(size)** → réserve
- **calloc(n, size)** → réserve et met à 0
- **realloc(ptr, newsize)** → redimensionne un bloc existant
- **free(ptr)** → libère

Allocation dynamique avec malloc

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

int main() {
    int *ptr = (int*) malloc(5 * sizeof(int));
    if (ptr == NULL) {
        printf("Erreur d'allocation\n");
        return 1;
    }
    for (int i = 0; i < 5; i++) ptr[i] = i * 2;
    free(ptr);
    return 0;
}
```

- malloc réserve un bloc mémoire
- free libère ce bloc

Autres fonctions d'allocation

- Exemple :

```
int *tab = calloc(10, sizeof(int));  
tab = realloc(tab, 20 * sizeof(int));  
free(tab);
```

Problèmes fréquents

- **Fuite mémoire** : oubli de `free()` → bloc inutilisable mais toujours occupé
- **Double libération** : appel `free()` deux fois → comportement indéfini
- **Segmentation fault** : accès à une zone non autorisée

Segmentation

Définition

- La segmentation divise la mémoire en segments logiques :
 - Code
 - Données
 - Pile
- Chaque segment est défini par :
 - Base → adresse de départ en mémoire.
 - Limite → taille du segment (jusqu'où il s'étend).
- Exemple :
 - Segment code : base = 0x400000, limite = 64 Ko.
 - Segment pile : base = 0x7fff0000, limite = 8 Mo

Pagination

- Mémoire divisée en pages fixes (souvent 4 Ko)
- La MMU traduit adresses virtuelles → physiques
- Permet :
 - Allocation flexible
 - Mémoire virtuelle (swap sur disque)
- Commande : `getconf PAGE_SIZE`

Mémoire virtuelle

- Chaque processus croit avoir un espace continu
- En réalité : pages dispersées en RAM + disque
- Avantages :
 - Isolation des processus
 - Permet d'exécuter de grands programmes même avec RAM limitée
- Exemple : `pmap <PID>` ou `/proc/<PID>/maps`

Protection mémoire

- Empêche un processus d'écrire dans la mémoire d'un autre
- Garantit la stabilité et la sécurité du système
- Accès interdit → SIGSEGV (segfault)
- Exemple :

```
int *p = NULL; // pointeur nul (adresse 0, interdite)
*p = 42; // Crash, tentative d'écriture illégale
```
- Autres cas fréquents
 - Accéder à une zone libérée (free(ptr); *ptr = 10;).
 - Lire/écrire au-delà d'un tableau (arr[1000] quand le tableau fait 100 cases).
 - Écrire dans une zone en lecture seule (ex. segment code).

mmap – Mapper un fichier en mémoire

Principe

- mmap() permet de projeter un fichier (ou une zone anonyme) directement dans l'espace mémoire du processus.
- Le contenu du fichier est alors vu comme un tableau en mémoire.
- Pas besoin de faire des boucles read() / write().

Avantages

- Accès direct par pointeur (comme un tableau).
- Gestion efficace des grands fichiers (lazy loading page par page).
- Partage entre processus possible → utile pour IPC.
- Utilisé par : bases de données, bibliothèques dynamiques, mémoire partagée.

Exemple code avec mmap pour lire un fichier

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

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <fichier>\n", argv[0]);
        return 1;
    }

    // Ouvrir le fichier en lecture seule
    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    // Récupérer la taille du fichier
    struct stat sb;
    if (fstat(fd, &sb) == -1) {
        perror("fstat");
        close(fd);
        return 1;
    }

    // Mapper le fichier en mémoire
    char *addr = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (addr == MAP_FAILED) {
        perror("mmap");
        close(fd);
        return 1;
    }

    // Afficher le contenu (comme une chaîne de caractères)
    write(STDOUT_FILENO, addr, sb.st_size);

    // Nettoyer
    munmap(addr, sb.st_size);
    close(fd);

    return 0;
}
```

- open() : ouvre le fichier.
- fstat() : récupère sa taille.
- mmap() : projette le fichier en mémoire → pointeur data.
- write() : envoie directement le contenu sur la sortie standard.
- munmap() et close() : libèrent les ressources.

Outils d'analyse

Valgrind — Détection des erreurs mémoire

Détecte :

- Fuites mémoire (malloc sans free).
- Accès invalides (hors tableau, après free).

```
valgrind --leak-check=full
```

```
./programme
```

GDB — Débogueur interactif

- Permet d'analyser un segfault ou un plantage.
- Usage typique :

```
gdb ./programme
```

```
(gdb) run
```

```
(gdb) backtrace      # montre où ça a  
planté
```

pmap — Carte mémoire d'un processus

- Affiche les zones mémoire utilisées par un processus.

/proc/PID/maps — Vue détaillée des segments mémoire

- Fichier virtuel listant toutes les zones mémoire mappées

À vous de jouer

- TP 5– Gestion de la mémoire
- TD3 – Gestion de la mémoire
- Mini-Projet 1

Programmation multi-thread

Introduction

- Un thread (fil d'exécution) est une unité légère d'exécution à l'intérieur d'un processus.
- Un processus peut contenir plusieurs threads qui s'exécutent en parallèle.
- Tous les threads partagent les ressources du processus parent (mémoire, fichiers).
- Objectif : améliorer la concurrence et exploiter les processeurs multi-cœurs.

Différence Processus vs Thread

Aspect	Processus	Thread
PID	Unique	Partage le PID du processus
Mémoire	Séparée	Partagée entre threads
Création	Lourde (via fork)	Légère (pthread_create)
Communication	IPC (pipes, sockets, etc.)	Directe (variables globales)
Isolation	Forte (sécurité)	Faible (un bug → tout le processus)

Threads POSIX (pthreads)

- Bibliothèque POSIX Threads (pthread).
- Fonctions principales :
 - `pthread_create()` → créer un thread.
 - `pthread_join()` → attendre la fin d'un thread.
 - `pthread_exit()` → terminer un thread.

Exemple

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

void* routine(void* arg) {
    printf("Hello depuis un thread !\n");
    return NULL;
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, routine, NULL);
    pthread_join(t, NULL);
    return 0;
}
```


Partage de mémoire entre threads

- Tous les threads accèdent aux variables globales et au tas (heap).
- Risque : conditions de course si plusieurs threads modifient une même variable sans synchronisation.
- Exemple de problème

```
int compteur = 0;
void* routine(void* arg) {
    for (int i = 0; i < 1000000; i++) {
        compteur++; // non protégé
    }
    return NULL;
}
```

Résultat final incorrect car les threads écrivent en même temps.

Synchronisation

Mutex (verrou)

- Garantit qu'un seul thread accède à une section critique à la fois
- Fonctions :
 - pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_destroy.
- Exemple

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int compteur = 0;

void* routine(void* arg) {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);

        compteur++;

        pthread_mutex_unlock(&lock);
    }

    return NULL;
}
```

Sémaphores

- Compteurs synchronisés pour gérer la disponibilité d'une ressource.
- Fonctions : `sem_init`, `sem_wait`, `sem_post`.
- Utiles pour le problème producteur/consommateur.

Problèmes courants

- Deadlock (interblocage)
- Quand deux (ou plus) threads s'attendent mutuellement et restent bloqués indéfiniment.
- Exemple :
 - Thread A : verrouille mutex1, attend mutex2
 - Thread B : verrouille mutex2, attend mutex1.
 - 🙅 Aucun ne progresse.
- Solutions :
 - Toujours verrouiller les ressources dans le même ordre.
 - Utiliser des timeouts (pthread_mutex_timedlock).
 - Préférer des mécanismes plus sûrs (sémaphores, verrous hiérarchiques).

Starvation (inanition)

Un thread n'arrive jamais à accéder à la ressource car d'autres prennent toujours le verrou avant lui.

👉 Il « meurt de faim » alors que le programme continue.

Solutions :

- Utiliser des mutex équitables (avec politiques FIFO).
- Limiter la durée de possession des verrous.

Condition de course

- Plusieurs threads modifient une donnée sans coordination → résultat imprévisible.
- Exemple classique sans mutex :

```
counter++; // pas atomique
```

Avantages et inconvénients

Avantages :

- Plus léger que les processus.
- Communication directe via mémoire partagée.
- Permet de tirer parti du multi-cœur.

Inconvénients :

- Moins isolés → un bug dans un thread peut planter tout le processus.
- Synchronisation complexe (risques de deadlocks).

Outils d'analyse et debug

- htop → voir threads avec H.
- gdb → debug multi-thread (info threads, thread <id>).
- valgrind -tool=helgrind → détecter conditions de course.

À vous de jouer

- TP6-Multithreading

Organisation des systèmes de fichiers

Arborescence Unix/Linux

- Principe général
 - Un seul arbre qui commence à la racine /.
 - Tout est fichier : programmes, périphériques, sockets, pipes, etc.
 - Les sous-systèmes (disques, partitions, périphériques) viennent s'accrocher (mount) quelque part dans l'arborescence
- Répertoires principaux
 - / : racine de l'arbre.
 - /home : dossiers personnels des utilisateurs.
 - /root : répertoire personnel de l'administrateur (root).
 - /bin : commandes essentielles (exécutables accessibles même en mode secours).
 - /usr/bin : la majorité des programmes installés pour les utilisateurs.
 - /sbin, /usr/sbin : programmes d'administration système.
 - /etc : fichiers de configuration du système et des services.
 - /var : données variables (logs, mails, spool d'impression...).
 - /tmp : fichiers temporaires.
 - /dev : périphériques vus comme des fichiers (/dev/sda, /dev/tty, etc.).
 - /proc et /sys : pseudo-systèmes de fichiers donnant accès aux infos du noyau et des processus.

Gestion des fichiers et systèmes de fichiers

Manipulation des fichiers (ouverture, lecture, écriture, fermeture)

- `open(path, flags, mode)` → ouvrir ou créer un fichier.
- `read(fd, buf, size)` → lire des octets depuis un descripteur de fichier.
- `write(fd, buf, size)` → écrire des octets dans un fichier.
- `close(fd)` → fermer un fichier et libérer la ressource.

Types de fichiers

- Fichiers réguliers : texte, binaire.
- Répertoires : contiennent d'autres fichiers.
- Liens symboliques : raccourcis (ln -s).
- Fichiers spéciaux : périphériques (/dev/null, /dev/sda).
- Sockets : communication entre processus.

Métadonnées (inodes)

Chaque fichier est représenté par un inode contenant :

- Taille,
- Permissions,
- Propriétaire,
- Dates (création, modification),
- Pointeurs vers les blocs de données.
- Commande utile :

```
ls -li fichier.txt    # montre l'inode
```

```
stat fichier.txt      # détails complets
```

Permissions et sécurité des fichiers

Modèle Unix classique

Chaque fichier a :

- Un propriétaire (user)
- Un groupe (group)
- Des permissions (mode) : lecture (r), écriture (w), exécution (x).
- Exemple

```
ls -l exemple.txt
```

```
-rw-r--r-- 1 alice users 20 sep 22 10:00 exemple.txt
```

- rw- → propriétaire (alice) peut lire/écrire.
- r-- → groupe (users) peut lire.
- r-- → autres peuvent lire.

Exemple

```
#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

int main() {

    int fd = open("exemple.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    if (fd < 0) { perror("open"); return 1; }

    const char *msg = "Bonjour, fichier !\n";

    write(fd, msg, strlen(msg));

    close(fd);

    return 0;

}
```

Ce code :

- crée exemple.txt,
- écrit un message dedans,
- ferme le fichier.

Modification des permissions et sécurité

- Modification des permissions
 - `chmod 600 fichier.txt` → seul le propriétaire peut lire/écrire.
 - `chown bob:bob fichier.txt` → change propriétaire et groupe.
- Sécurité
 - Isolation : un utilisateur ne peut pas lire/modifier les fichiers d'un autre sans permission.
 - Processus système (root) peuvent tout voir, d'où la nécessité d'être prudent.

À vous de jouer

- TP7 Système de fichiers

Optimisation et performances en programmation système

Pourquoi optimiser ?

- En programmation système, on manipule directement :
 - CPU (calculs, threads),
 - mémoire (allocation, accès),
 - périphériques (disques, réseau).
- La performance a un impact direct sur :
 - Serveurs (nombre de requêtes/s),
 - Applications temps réel (latence),
 - Bases de données / OS (efficacité des ressources).

Techniques d'optimisation de code

Réduire les appels système

Chaque `read()`, `write()`, `fork()`, etc. implique un changement de contexte vers le noyau → coûteux.

Exemple inefficace :

// Lire 1 caractère à la fois

```
while (read(fd, &c, 1) == 1) {  
    process(c);  
}
```

Version optimisée :

// Lire en bloc

```
char buf[4096];  
  
ssize_t n;  
  
while ((n = read(fd, buf,  
sizeof(buf))) > 0) {  
    process_block(buf, n);  
}
```

Techniques d'optimisation de code

- Éviter les copies inutiles
 - Travailler directement en mémoire si possible (mmap plutôt que read/write en boucle).
 - Utiliser des pointeurs plutôt que recopier des tableaux.
- Utiliser les bons algorithmes
 - Complexité algorithme > optimisations mineures.
 - Exemple : remplacer une recherche linéaire $O(n)$ par une recherche dichotomique $O(\log n)$.

Buffers et caches

Buffers

- Zone mémoire tampon → regroupe les données avant écriture/lecture.
- Exemple : printf utilise un buffer, qui est vidé d'un coup quand il est plein ou quand on écrit `\n`.
- Illustration :
 - Sans buffer : 1000 appels `write()` → 1000 syscalls.
 - Avec buffer : 1 seul `write()` → bien plus rapide.

Caches

- Le CPU et l'OS utilisent des caches pour réduire les temps d'accès :
 - Cache CPU (L1, L2, L3).
 - Cache disque (page cache dans le noyau).
- Le programmeur système doit en tenir compte :
 - Localité spatiale : parcourir un tableau séquentiellement (meilleur usage du cache).
 - Localité temporelle : réutiliser les mêmes données souvent.
- Exemple :
 - Accéder à un tableau séquentiellement est bien plus rapide qu'y accéder aléatoirement.

Introduction au benchmarking

Qu'est-ce que le benchmarking ?

Le benchmarking consiste à évaluer les performances d'un programme en mesurant différents critères objectifs, au lieu de se fier à des intuitions.

➡ Cela permet de comparer différentes versions d'un code ou de vérifier l'impact d'une optimisation

Ce qu'on peut mesurer

1. Temps d'exécution

- a. Durée totale (mur)
- b. Temps CPU (user/system)
- c. Exemple : `time ./prog`

2. Utilisation CPU

- a. Pourcentage d'occupation, nombre de cœurs utilisés
- b. Exemple : `top`, `htop`

3. Consommation mémoire

- a. Pic de mémoire utilisée, fuites
- b. Exemple : `valgrind --tool=massif, /usr/bin/time -v ./prog`

Ce qu'on peut mesurer

- Entrées/Sorties (I/O)
 - Volume lu/écrit sur disque, appels systèmes
 - Exemple : `strace -c ./prog`, `iostat`
- Profiling plus fin
 - Temps passé par fonction, hot spots
 - Exemple : `gprof`, `perf record`

Bonnes pratiques

- Mesurer plusieurs fois : les performances varient (bruit du système, caches).
- Tester sur des données représentatives : éviter les “toy examples” trop petits.
- Comparer équitablement : même machine, mêmes conditions, sans autres charges.
- Documenter : noter les paramètres, l’environnement, la version du compilateur.

À vous de jouer

- TP8-Benchmarking
- MiniProjet-ParallelGrep

Reference

- Juliusz Chroboczek, Programmation système, notes de cours, 4 mars 2024.
- Michael Blondin, Programmation système (IFT209), notes de cours, 22 février 2024.
- Georges-André Silber, Introduction à la programmation système, École des Mines de Paris, notes de cours, septembre 2024.