

Introduction aux systèmes d'exploitation

Ceci est un support non officiel, qui a pour but de regrouper l'ensemble des notions vues en cours d'introduction aux systèmes d'exploitation dispensé en deuxième année de licence informatique à l'UFR sciences et techniques du Madrillet par le professeur ZIADI DJELLOUL Il est non exhaustif et collaboratif. Ce document est à jour pour l'année universitaire 2025-2026. Pour toute suggestion, ouvrez une issue sur le dépôt GitHub. Merci aux contributeurs : <https://github.com/SDC-M/Introduction-aux-syst-mes-d-exploitation.git>

Le support se décompose en deux parties. Premièrement des rappels de cours avec des exemples commentés ainsi que des définitions puis une partie avec des exercices corrigés.

5 Octobre 2025

Table des matières

1. Chapitre 1: Introduction et gestion des processus	4
1.1. Introduction	4
1.1.1. Types de systèmes d'exploitation	4
1.2. Les normes	5
1.2.1. POSIX	5
1.2.2. SUS	5
1.3. Les appels systèmes	6
1.4. Fonctionnement	6
1.4.0.1. Mode Noyau vs Mode Utilisateur	6
1.4.1. Les interruptions	6
1.5. Gestion des erreurs	7
1.5.1. appels systèmes vs bibliothèque standard	7
1.6. Les processus	7
1.6.1. Création	8
1.6.2. Cycle de vie	9
1.6.3. Terminaison	9
1.6.4. Sessions de processus	10
1.6.5. Recouvrement des processus	10
1.7. Ordonnancement	11
1.7.1. Ordonnancement préemptif vs non préemptif	11
1.7.2. Mesures de performance	12
1.7.3. Les différentes politiques	12
1.7.3.1. Premier Arrivé, Premier Servi (FCFS)	12
1.7.3.2. Job le Plus Court en Premier (SJF)	13
1.7.3.3. Tourniquet (Round Robin)	13
1.7.4. Exemple complet	14
1.7.5. Time slicing	17
1.7.6. Sous linux	17
1.8. Les threads POSIX	18
1.8.1. Threads vs Processus	19
1.8.2. Types de threads	19
1.8.3. Création	20
1.8.4. Terminaison	21
1.8.5. Attributs et états	21
1.9. Problèmes de mémoire	23
1.10. Bonnes pratiques	23
1.10.1. Recommandations pour une application C portable	23
2. Chapitre 2: Système de gestion de fichiers (SGF)	25
2.1. Systèmes de gestion de fichiers	25
2.1.1. SGF vs SF	25

2.2.	Structure interne d'un SF Ext	26
2.3.	Structure d'un i-noeud	27
2.3.1.	Opérations sur l'i-noeud	28
2.3.2.	Permissions sur les fichiers	28
2.4.	Opérations sur les fichiers	29
2.5.	Tables système de manipulation de fichiers	30
2.6.	Opération sur les fichiers	30
2.6.1.	L'effet de l'appel système open():	31
2.6.2.	L'effet de l'appel système close():	32
2.6.3.	Duplication de descripteur de fichier dup & dup2	32
2.7.	Opérations sur les répertoires	33
2.8.	Implémentation et manipulation	33
2.9.	Manipulation des répertoires	34
2.10.	Création et suppression	34
2.11.	Récapitulatif des appels système	35
3.	Chapitre 3: Les tubes	36
3.1.	Types de tubes sous Linux	36
3.1.1.	Tubes nommés vs tubes anonymes	37
3.2.	Création et ouverture tube anonyme	37
3.2.1.	Contenu de l'i-noeud virtuel d'un tube	39
3.2.2.	Structure pipe_inode_info	40
3.3.	Création et ouverture tube nommé	40
3.3.1.	Définition des rôles	41
3.4.	Lecture sur un tube	41
3.4.1.	Mode bloquant (Par défaut)	42
3.4.2.	Mode non-bloquant	42
3.5.	Écriture dans un tube	43

1. Chapitre 1: Introduction et gestion des processus

1.1. Introduction

Un système d'exploitation est un **logiciel intermediaire** remplissant deux fonctions principales:

- Assurer la **gestion efficace** des périphériques matériels (le clavier, l'écran, le disque, la mémoire, le processeur, ...)
- Offrir aux programmes une **interface abstraite** et simplifiée pour interagir avec le matériel, sans en connaître les détails techniques.

Le système d'exploitation optimise et sécurise l'utilisation des ressources en répartissant le temps CPU entre les différents processus on parle d'ordonnancement.

Alloue libère la mémoire utilisée pour chaque processus.

Gère et sécurise les lectures / écritures ainsi que l'organisation des fichiers sur le disque (arborescence de fichier).

1.1.1. Types de systèmes d'exploitation

- **Mainframes** : traitement de très gros volumes de données (ex. IBM z/OS).
- **Serveurs** : gestion des services réseaux (ex. Linux, Windows Server).
- **Multiprocesseurs** : exploitation de plusieurs CPU en parallèle.
- **Personnels** : ordinateurs individuels, interface conviviale (ex.Windows, macOS).
- **Temps réel** : respect de délais stricts, applications critiques (ex.VxWorks).

- **Embarqués** : systèmes pour appareils spécifiques (ex. Android, FreeRTOS).
- **Cartes à puce** : ultra-légers et sécurisés (ex. Java Card).

1.2. Les normes

Les normes sont un moyen d'assurer la portabilité ainsi que la durabilité d'un code, pour cela nous disposons de différents niveaux de certifications afin de mesurer la qualité du code produit.

1.2.1. POSIX

Créée en 1983 par Institute of Electrical and Electronics Engineers - IEEE <https://posix.opengroup.org/>.

Standard définissant une **interface commune pour les systèmes d'exploitation** de type UNIX.

Garantit la portabilité des applications entre différents systèmes de type UNIX.

Pour définir le respect des normes POSIX avec GCC il suffit de définir `_POSIX_SOURCE` pour demander le respect général de POSIX ou bien `_POSIX_C_SOURCE` pour activer des fonctionnalités spécifiques selon la version : « Année + Mois + L ».

1.2.2. SUS

La Single UNIX Specification (SUS) est une norme définie et gérée par The Open Group.

Fin des années 1990 : POSIX devient un sous-ensemble de la SUS. Tous les systèmes UNIX certifiés respectent POSIX.

Pour définir le respect de la norme SUS avec GCC il suffit de définir `_XOPEN_SOURCE` pour demander le respect général de SUS. Le chiffre « X » correspondant à l'aversion majeure de SUS souhaitée. Vous retrouverez l'ensemble des informations dans le manuel avec la commande : `man 7 standards`

1.3. Les appels systèmes

1.4. Fonctionnement

Un appel système est une interface entre un programme utilisateur et le noyau du système d'exploitation. Il donne accès à des ressources protégées (fichiers, réseau, mémoire, ...) Il s'exécute en mode noyau pour réaliser des tâches privilégiées. Les appels système provoquent donc un passage du mode utilisateur au mode noyau.

1.4.0.1. Mode Noyau vs Mode Utilisateur

Le mode utilisateur :

- Espace où s'exécutent les applications.
- Accès limité aux ressources matérielles.
- Protection contre les erreurs utilisateur.

Le mode noyau :

- Espace privilégié pour le système d'exploitation.
- Accès direct au matériel.
- Gère les appels système et les interruptions.

1.4.1. Les interruptions

Gestion des erreurs

- En cas d'erreur : la fonction retourne -1.
- Le code d'erreur est stocké dans la variable globale `errno`.
- Utiliser `perror()` ou `strerror(errno)` pour afficher un message compréhensible

1.5. Gestion des erreurs

Fonction	POSIX	Thread-safe	Recommandation
<code>perror()</code>	Oui	Oui	exemples simples
<code>strerror()</code>	Oui	Non	À éviter en multithread
<code>strerror_r()</code>	Oui	Oui	Préférée en multithread

Les recommandations sont donc :

- Programmes simples : `perror()` ou `strerror()`
- Programmes modernes / multithread : `strerror_r()` (POSIX)

1.5.1. appels systèmes vs bibliothèque standard

	Appels Système	Bibliothèque Standart
Niveau	Bas niveau	Haut niveau
Portabilité	Dépend du système	Portable
Complexité	Plus complexe	Plus simple
Exemple	<code>write</code>	<code>printf</code>

1.6. Les processus

L'un des premiers choix de conception que nous devons faire lors de la

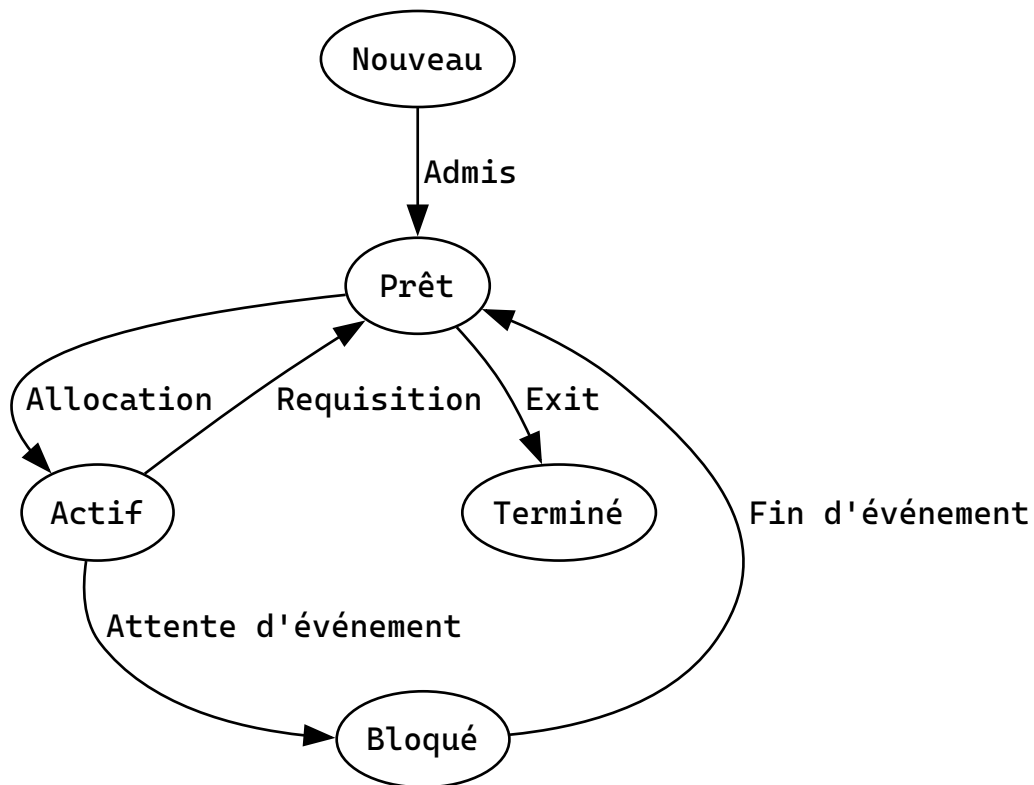
conception d'une application multitâches sera : **processus** ou **threads**. Chacune des approches possède son lot d'avantages et d'inconvénients. Les processus s'exécutent dans des espaces mémoires distincts. Ceci est très important : **chaque processus dispose d'une zone de mémoire totalement indépendante et protégée des autres processus**.

1.6.1. Création

Le seul moyen que nous possédons pour créer des processus est de passer par l'appel système `fork()` qui va dupliquer le processus appelant. Au retour de cet appel système, deux processus identiques continueront d'exécuter le code à la suite de `fork()`. La différence essentielle entre ces deux processus est un numéro d'identification. On distingue le processus père du processus fils par leur **PID**. (processus identifiant). Que l'on pourrait changer au cours du programme si nécessaire avec l'appel système `setuid()`.

De plus, lorsqu'un processus est créé il dispose d'une copie des données de son père, mais également de l'environnement de celui-ci notamment la **table des descripteurs de fichiers**. De plus chaque processus appartient à un ou plusieurs groupes **GID**.

1.6.2. Cycle de vie



Automate représentant le cycle de vie d'un processus

1.6.3. Terminaison

Il existe différents types de terminaisons d'un processus :

- Arrêt normal (volontaire) Le programme s'exécute jusqu'à la fin prévue. Exemple : retour de `main()` avec `return 0;`
- Arrêt suite à une erreur (volontaire) Le programme détecte une erreur et décide de se terminer. Exemple : appel à `exit(EXIT_FAILURE);`
- Arrêt pour erreur fatale (involontaire) Le système arrête le processus à cause d'une faute grave. Exemple : `segmentation fault`.
- Arrêt forcé par un autre processus (involontaire) Un signal externe met fin au processus. Exemple : `kill -9 PID`.

1.6.4. Sessions de processus

Il existe finalement un dernier regroupement de processus, les **sessions**, qui réunissent divers groupes de processus. Ce sont principalement les applications s'exécutant en **arrière plan** qui utilisent les sessions. De manière générale une session est attachée à un terminal de contrôle, celui qui a servi à la connexion de l'utilisateur. Au sein d'une session, un groupe de processus est en avant-plan. Il reçoit directement les données saisies sur le clavier du terminal, et peut afficher ses informations de sortie sur l'écran de celui-ci. Les autres groupes de processus de la session s'exécutent en arrière plan.

La création d'une session s'effectue par l'appel système `setsid()`.

1.6.5. Recouvrement des processus

Le recouvrement d'un processus désigne le remplacement de son image mémoire par un nouveau programme. Il est réalisé via la famille d'appels système **exec**. Le recouvrement implique donc que nous conservons la même entrée dans la table des processus. Les appels système sont:

- **execl**: `int execl(const char *pathname, const char *arg, ... /*, (char *) NULL */);`
- **execle**:
`int execle(const char *pathname, const char *arg, ... /*, (char *) NULL, char *const envp[] */);`
- **execlp**: `int execlp(const char *file, const char *arg, ... /*, (char *) NULL */);`
- **execv**: `int execv(const char *pathname, char *const argv[]);`
- **execvp**: `int execvp(const char *file, char *const argv[]);`
- **execvpe**: `int execvpe(const char *file, char *const argv[], char *const envp[]);`

Ces fonctions sont des variantes de l'appel système `execve`, offrant différentes façons de passer les arguments et l'environnement.

Les noms des fonctions `exec` sont construits avec une combinaison de suffixes, chacun ayant une signification précise:

- **l (List)** : Les arguments de la ligne de commande sont passés sous forme d'une liste de paramètres individuels à la fonction.
- **v (Vector)** : Les arguments sont transmis via un tableau de chaînes de caractères.
- **p (Path)** : La fonction recherche l'exécutable dans les répertoires de la variable d'environnement `PATH`.
- **e (Environment)** : Le dernier paramètre est un tableau de variables d'environnement à transmettre.

1.7. Ordonnancement

L'ordonnancement des processus est un principe fondamental de la matière, en effet, à un instant donnée, plusieurs processus peuvent être en concurrence pour l'utilisation du processeur. Il faut donc choisir quel processus sera exécuté et à quel moment, cette décision est prise par l'ordonnanceur qui lui même suit une **politique d'ordonnancement**.

1.7.1. Ordonnancement préemptif vs non préemptif

Ordonnancement non préemptif:

- Une fois choisi, le processus garde le processeur jusqu'à ce qu'il se termine, ou se bloque (par exemple en attente d'une E/S).
- L'ordonnanceur n'intervient pas avant ce moment.

Ordonnancement préemptif:

- Le processus ne garde le processeur que pour un temps limité (quantum de temps)
- À l'expiration de ce délai, l'ordonnanceur peut choisir un autre processus, même si le premier n'a pas fini.
- Permet une meilleure réactivité et un partage équitable du CPU.

1.7.2. Mesures de performance

Pour mesurer la performance des différentes politiques on utilise différents indicateurs:

- Temps de réponse: Le temps écoulé entre la soumission d'un processus (son arrivée dans la file d'attente) et le moment où il commence à s'exécuter pour la première fois. **Temps de réponse = Temps de première exécution - Temps d'arrivée**
- Temps de rotation: Le temps total écoulé entre la soumission d'un processus et son achèvement complet. **Temps de rotation = Temps de fin - Temps d'arrivée**
- Temps d'attente: La somme de toutes les périodes pendant lesquelles un processus est prêt à s'exécuter et attend dans la file d'attente (Prêt). **Temps d'attente = Temps de rotation - Temps d'exécution CPU total.**

1.7.3. Les différentes politiques

1.7.3.1. Premier Arrivé, Premier Servi (FCFS)

Ordonnancement non préemptif où les processus sont exécutés dans l'ordre d'arrivée.

Mécanisme:

- Une seule file d'attente pour les processus créés.
- Le suivant est exécuté si un processus se bloque.
- Un processus débloqué est remplacé en fin de file.

1.7.3.2. Job le Plus Court en Premier (SJF)

Ordonnancement non préemptif nécessitant les durées d'exécution à l'avance.

Mécanisme:

- Liste des processus triée par durée.
- L'ordonnanceur choisit le processus le plus court.
- Optimal pour le délai de rotation si tous arrivent en même temps.

1.7.3.3. Tourniquet (Round Robin)

Algorithme préemptif avec un quantum de temps par processus.

Mécanisme:

- Chaque processus reçoit un intervalle (quantum) d'exécution.
- Interruption à la fin du quantum, passage au suivant.
- File d'attente pour les processus prêts.

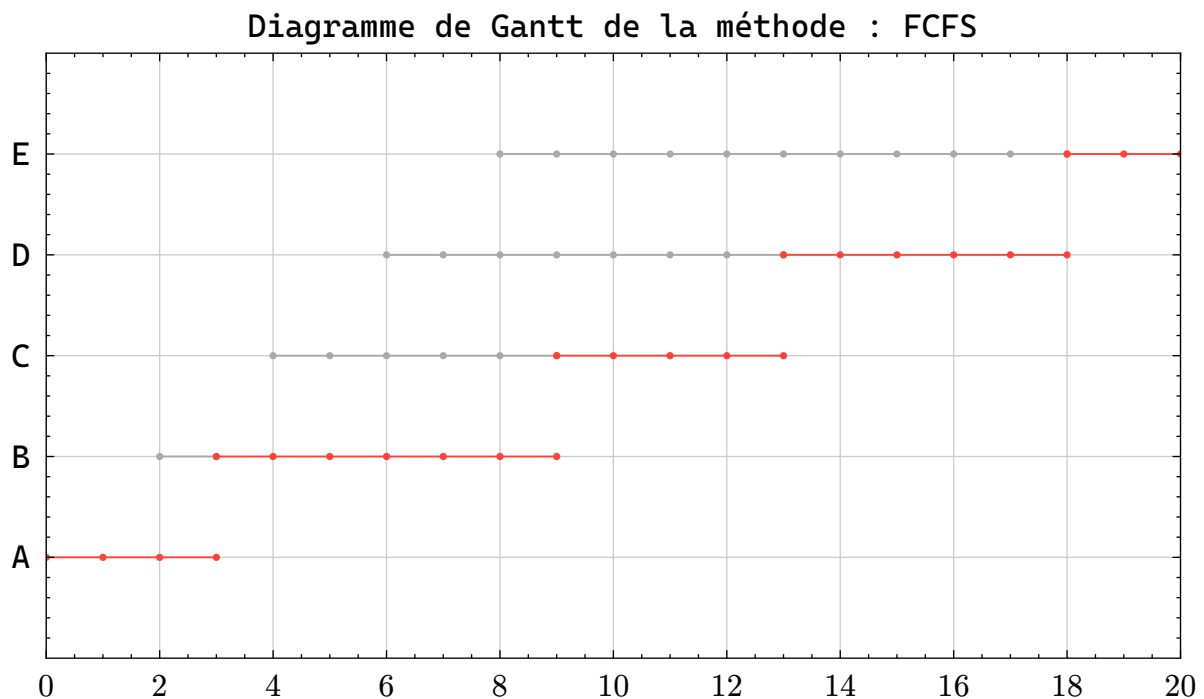
Attention au choix du quantum, s'il est trop court dans ce cas on aura un coût élevé pour le changement de contexte, dans le cas contraire s'il est trop long nous aurons un temps de réponse lent pour les interactions.

1.7.4. Exemple complet

Pour illustrer ces différentes politiques d'ordonnancement nous allons prendre un exemple et appliquer chacune des politiques, nous comparerons alors celles-ci avec les différents indicateurs que nous avons énumérés précédemment.

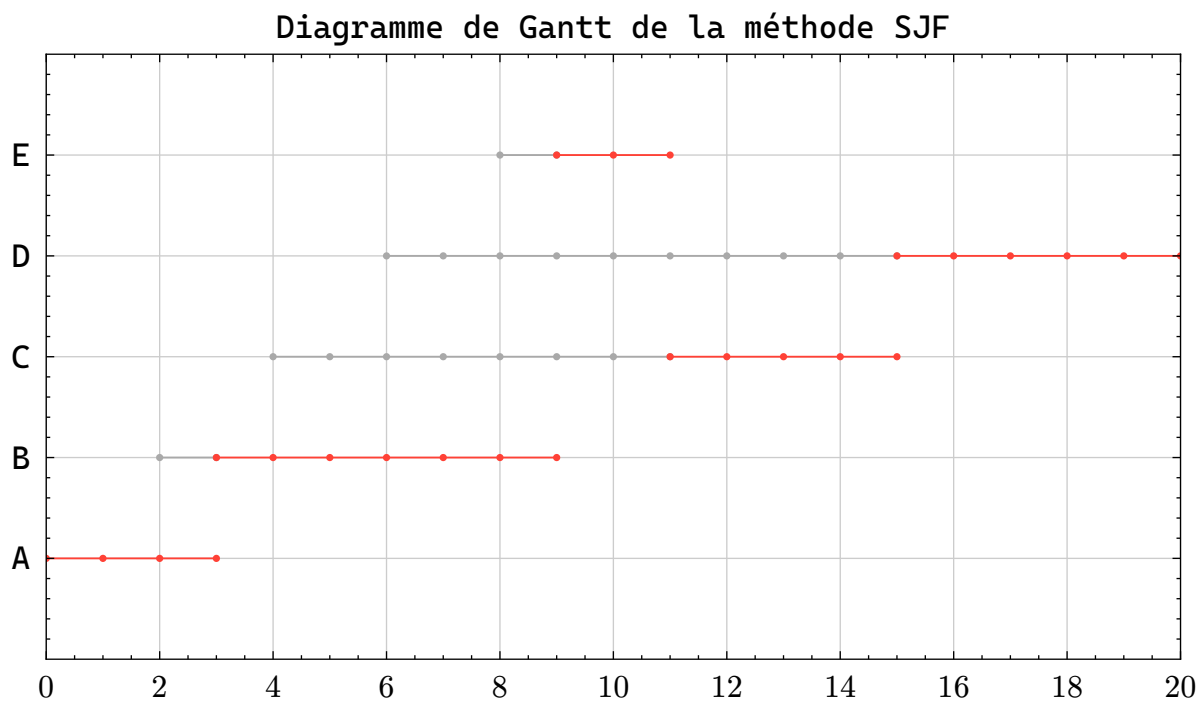
Premier Arrivé, Premier Servi (FCFS):

Processus	arrivé	demandé	réponse	rotation	attente
A	0	3	$3 - 0 = 3$	$3 - 0 = 3$	$3 - 3 = 0$
B	2	6	$9 - 2 = 7$	$9 - 2 = 7$	$7 - 6 = 1$
C	4	4	$13 - 4 = 9$	$13 - 4 = 9$	$9 - 4 = 5$
D	6	5	$18 - 6 = 12$	$18 - 6 = 12$	$12 - 5 = 7$
E	8	2	$20 - 8 = 12$	$20 - 8 = 12$	$12 - 2 = 10$
Moyenne			4,6	8,6	4,6



Job le Plus Court en Premier (SJF):

Processus	arrivé	demandé	réponse	rotation	attente
A	0	3	$0 - 0 = 0$	$3 - 0 = 3$	$3 - 3 = 0$
B	2	6	$3 - 2 = 1$	$9 - 2 = 7$	$7 - 6 = 1$
C	4	4	$11 - 4 = 7$	$15 - 4 = 11$	$15 - 4 = 11$
D	6	5	$15 - 6 = 9$	$20 - 6 = 14$	$14 - 5 = 9$
E	8	2	$11 - 8 = 3$	$11 - 8 = 3$	$3 - 2 = 1$
Moyenne			4	7,6	4,4



Liste du plus court à l'instant $x = ?$:

0 : A

3 : B

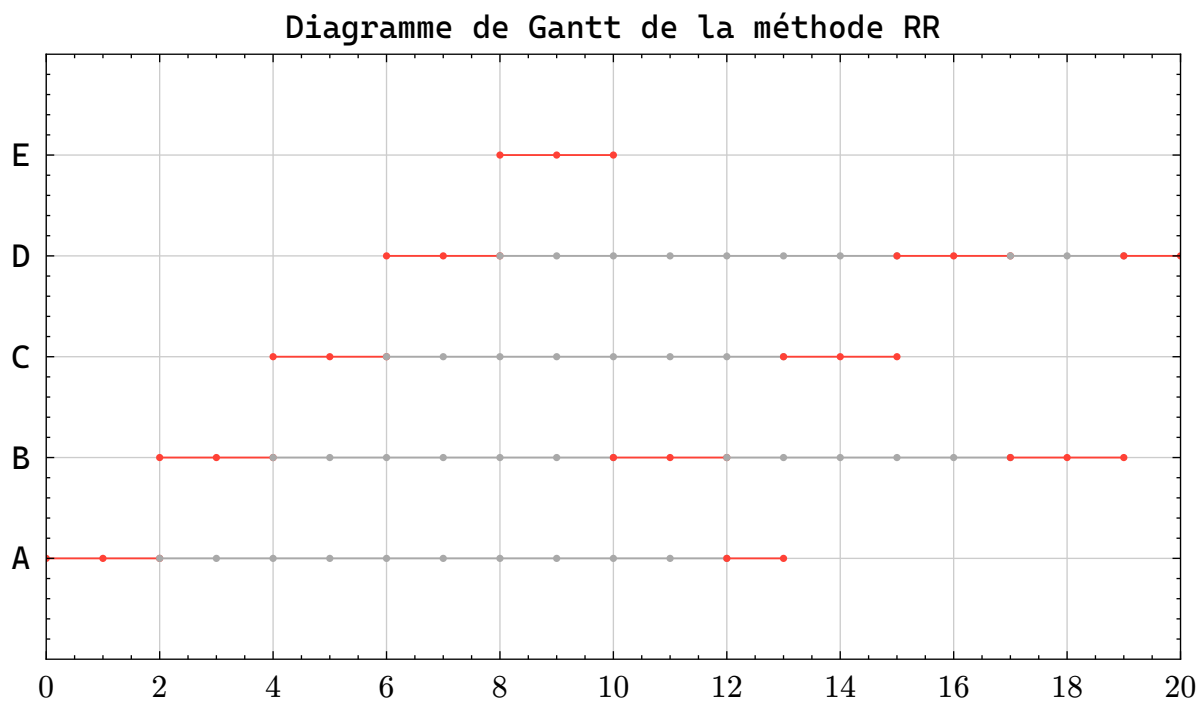
9 : E -> C -> D

11 : C -> D

15 : D

Tourniquet Round Robin (RR) : Choix du quantum = 2

Processus	arrivé	demandé	réponse	rotation	attente
A	0	3	$0 - 0 = 0$	$2 - 0 = 2$	$2 - 2 = 0$
B	2	6	$2 - 2 = 0$	$19 - 2 = 17$	$17 - 6 = 11$
C	4	4	$4 - 4 = 0$	$15 - 4 = 11$	$11 - 4 = 7$
D	6	5	$6 - 6 = 0$	$20 - 6 = 14$	$14 - 5 = 9$
E	8	2	$8 - 8 = 0$	$10 - 8 = 2$	$2 - 2 = 0$
Moyenne			0	9.2	5,4



État de la file d'attente pour les processus prêts:

T	État de la file	T	État de la file
0	A	12	A -> C -> D -> B
2	B -> A	13	C -> D -> B
4	C -> A -> B	15	D -> B
6	D -> B -> A -> C	17	B -> D
8	E -> B -> A -> C -> D	19	D
10	B -> A -> C -> D	19	∅

1.7.5. Time slicing

Le time slicing est une technique d'ordonnancement où le temps CPU est divisé en petits intervalles égaux appelés **quanta** ou **tranches de temps**.


- Chaque processus s'exécute pendant un quantum de temps.
- À la fin du quantum, le processus est préempté.
- Le CPU est attribué au processus suivant dans la file d'attente.
- Crée l'illusion d'un parallélisme sur les systèmes mono-core.

Attention tout de même dans le cas où nous devrions changer de contexte trop souvent cela créerait un **Overhead**. Dans le cas contraire un mauvais temps de réponse est donc une impression de latence.

Dans le cas d'un exemple concret un calcul demandant beaucoup de ressources CPU bloquerait toute interface graphique, alors qu'avec le time slicing nous pourrions continuer à l'utiliser.

Sous linux nous pouvons voir cette valeur avec la commande:


```
1 # en général 100ms
2 sudo cat /proc/sys/kernel/sched_rr_timeslice_ms
```

 Bash

1.7.6. Sous linux

Sous linux il est possible de définir la politique d'ordonnancement grâce à ces différents appels système:

```
1 • sched_setscheduler() - Définir la politique d'ordonnancement
2 • sched_getscheduler() - Obtenir la politique actuelle
3 • sched_setparam() - Définir les paramètres d'ordonnancement
4 • sched_getparam() - Obtenir les paramètres actuels
5 • sched_get_priority_max() - Priorité maximale pour une politique
6 • sched_get_priority_min() - Priorité minimale pour une politique
7 • sched_rr_get_interval() - Obtenir le quantum RR
8 • sched_setaffinity() - Définir l'affinité CPU
9 • sched_getaffinity() - Obtenir l'affinité CPU
```



1.8. Les threads POSIX

Un **thread** (ou « fil d'exécution ») parfois appelés « **processus légers** », est l'unité fondamentale que le système d'exploitation planifie sur un processeur. Il correspond à une séquence d'instructions qui s'exécute de manière indépendante en partageant l'espace d'adressage d'un processus.

Un même processus peut contenir un ou plusieurs threads, concurrents (sur un cœur) ou parallèle (sur plusieurs cœurs).

Chaque thread possède ses composants privés indispensables à son exécution :

- sa **propre pile d'exécution** (pour les variables locales et les appels de fonctions)
- son **contexte d'exécution** (état des registres du processeur et valeur du compteur ordinal)

⚠ **Attention: Tous les threads partagent la même mémoire globale!**

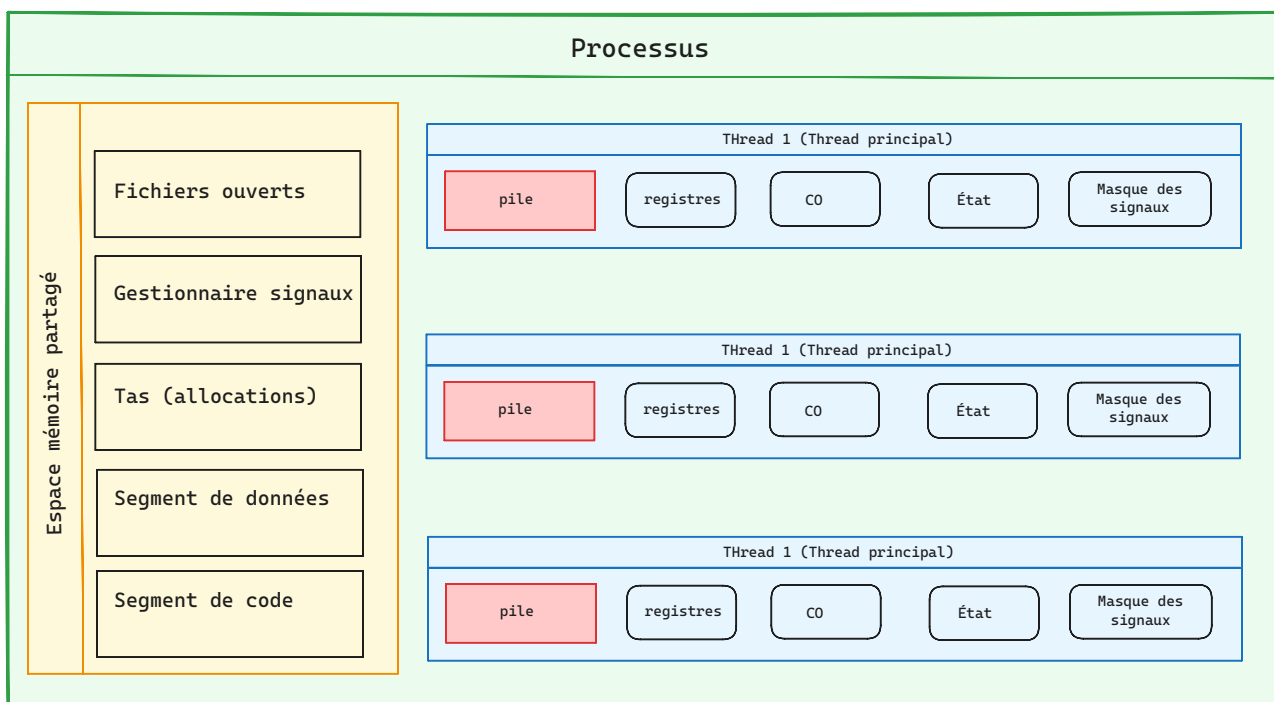


Schéma provenant du Cours de M.DJELLOUL

1.8.1. Threads vs Processus

Processus	Threads
Espace mémoire séparé	Espace mémoire partagé
Fichiers ouverts séparé	Fichiers ouverts partagés
Contexte d'exécution complet	Contexte d'exécution minimal
Communication inter-processus	Communication directe

Avantages Processus	Avantages Threads
Isolation & Stabilité	Réponse améliorée
Gestion fine des ressources	Économie de ressources

1.8.2. Types de threads

Il existe deux modèles principaux de gestion des threads:

1. Threads implantés dans l'espace utilisateur: **User-level Threads (ULTs)**

Threads gérés entièrement par une bibliothèque au niveau de l'application (espace utilisateur). Le noyau OS n'a pas connaissance de leur existence.

Ils ont pour avantage d'être très performant, car pas d'appel système, de plus ils sont portables car implémenter par une bibliothèque donc indépendante du noyau OS.

Cependant ils manquent de parallélisme réel, en effet un ULT bloquant bloque tous les threads du processus, de plus le scheduler OS ne voit qu'un seul processus, ne peut pas répartir les threads sur plusieurs coeurs.

2. Threads implantés dans le noyau **Kernel-level Threads (KLTs)**

Threads gérés directement par le système d'exploitation. Le noyau planifie leur exécution.

Le noyau peut planifier différents threads sur différents cœurs CPU. Un thread bloqué (e.g., sur une E/S) n'affecte pas les autres. Accès direct aux services et ressources du noyau. De plus ils bénéficient des politiques de scheduling du noyau.

Cependant la création, commutation et destructions sont plus lentes (appels système). L'API dépend souvent de l'OS et chaque thread nécessite des structures de données dans le noyau.

1.8.3. Création

```
1 int pthread_create ( pthread_t * thread,  
2     const pthread_attr_t * attr,  
3     void *(* start_routine ) ( void *),  
4     void * arg );
```

- thread : Pointeur pour stocker l'identifiant du nouveau thread
- attr : Attributs du thread (NULL pour défaut)
- start_routine : Fonction exécutée par le thread
- arg : Argument passé à la fonction
- Retourne : 0 en cas de succès, code d'erreur sinon

On peut récupérer son identifiant avec:

```
1 pthread_t pthread_self ( void );
```

Et comparer deux identifiants avec:

```
1 int pthread_equal ( pthread_t thread1 , pthread_t thread2 );
```

⚠ Attention: pthread_t est un type opaque et diffère en fonction des implémentations.

1.8.4. Terminaison

La fonction du thread se termine par un return, cependant pour une terminaison explicite nous devons faire un appel à la procédure:

```
1 // depuis un autre processus
2 void pthread_exit ((void *) retval);
3
4 // attendre les threads secondaire dans le main
5 void pthread_exit (nullptr);
```

Cependant pour attendre explicitement nous pouvons faire appel à:

```
1 pthread_join(th, nullptr);
```

1.8.5. Attributs et états

On caractérise les threads en deux catégories:

- **Joignable (par défaut)** : Doit être joint avec pthread_join:
 - Ressources conservées après terminaison
 - Nécessite un pthread_join() pour libérer les ressources
- **Détaché** : Ressources libérées automatiquement
 - Impossible de joindre le thread
 - Libération automatique à la terminaison

⚠ **L'appel à pthread_join() est bloquant jusqu'à ce que le thread attendu se termine. Le thread appelant est suspendu en attendant la terminaison du thread cible.**

De plus la gestion des attributs nécessite un petit peu de logistique.

Voici un exemple complet qui sera plus démonstratif:

```
1  # include <stdio.h>
2  # include <stdlib.h>
3  # include <pthread.h>
4  # include <unistd.h>
5
6  void *run (void * arg) {
7      printf(" Thread démarré (détachable)\n");
8      sleep(2);
9      printf("Thread terminé\n");
10     return nullptr;
11 }
12
13 int main (){
14     pthread_t thread;
15     pthread_attr_t attr;
16     // Initialisation des attributs
17     if(pthread_attr_init (&attr) != 0) {
18         perror ("pthread_attr_init");
19         exit(EXIT_FAILURE);
20     }
21     // Configuration du thread comme détachable
22     if(pthread_attr_setdetachstate(&attr , PTHREAD_CREATE_DETACHED ) != 0) {
23         perror("pthread_attr_setdetachstate");
24         exit(EXIT_FAILURE);
25     }
26     // Configuration de la taille de la pile de 2 MB
27     size_t stack_size = 2 * 1024 * 1024;
28     if(pthread_attr_setstacksize(&attr, stack_size) != 0) {
29         perror("pthread_attr_setstacksize") ;
30         exit(EXIT_FAILURE) ;
31     }
32     // Création du thread avec attributs personnalisés
33     if (pthread_create (&thread, &attr, run, nullptr) != 0) {
34         perror("pthread_create");
35         exit(EXIT_FAILURE);
36     }
37     // Destruction des attributs
38     if (pthread_attr_destroy (&attr) != 0) {
39         perror ("pthread_attr_destroy");
40         exit(EXIT_FAILURE);
41     }
42     printf ("Thread Principal Terminé\n");
43     pthread_exit(nullptr);
44 }
```

1.9. Problèmes de mémoire

Il faut néanmoins faire attention aux variables automatiques qui seraient initialisées par un thread car ils sont allouées dans la pile du thread en question, il suffit qu'un pointeur soit renvoyé sur cette valeur puisque le thread se termine (libération des ressources de sa stack), nous nous retrouverions avec une référence invalide. Ce qui représente une faille de sécurité majeure mais également un comportement indéfini lors de la lecture de la mémoire à partir de ce pointeur. La bonne solution est donc d'allouer dynamiquement dans le tas avec un malloc, ce qui nous confère une gestion sécurisée de la mémoire pointée par celui-ci.

1.10. Bonnes pratiques

1.10.1. Recommandations pour une application C portable

- **Respecter les Standards :**
 - Utiliser la norme C99 ou C11.
 - Se conformer à l'API POSIX (`#define _POSIX_C_SOURCE 200809L`).
- **Gérer les dépendances :**
 - Privilégier les bibliothèques standards.
 - Utiliser des outils de construction comme GNU Autotools ou CMake (outils standard de l'écosystème open-source).
- **Adhérer à la philosophie UNIX :**
 - Chaque programme fait une seule tâche.
 - Gérer les fichiers et les entrées/sorties via les descripteurs de fichiers.
 - Respecter la hiérarchie du système de fichiers.
- **Gérer les architectures :**

- Utiliser des types de données de taille fixe ().
- Gérer l'ordre des octets (endianness) pour les données binaires (htons() et ntohl()) pour garantir la compatibilité.
- **Utiliser des constantes symboliques**: améliore la portabilité et évite les nombres dits « magique ».

2. Chapitre 2: Système de gestion de fichiers (SGF)

Concept fondamental : Sur Linux tout est fichier !

Un **système de fichiers** est la structure logique qui organise et gère le stockage des données sur un support physique (disque dur, SSD, clé USB). C'est le coeur du système qui gère le stockage, l'accès et la modification des fichiers.

- **Gestion de l'espace disque** : Alloue et libère des blocs de mémoire de manière efficace pour optimiser le stockage.
- **Organisation hiérarchique** : Crée une structure en arborescence avec des dossiers (répertoires) et des fichiers, facilitant la navigation.
- **Gestion des métadonnées** : Conserve les informations cruciales sur chaque fichier (nom, taille, dates de création/modification, etc.).
- **Contrôle d'accès** : Gère les permissions pour déterminer qui peut lire, écrire ou exécuter un fichier.

2.1. Systèmes de gestion de fichiers

2.1.1. SGF vs SF

Système de Fichiers	Système de Gestion de Fichiers
C'est la structure logique	C'est la partie logicielle composante du noyau
Définit comment les fichiers sont organisés (arborescence)	Gère toutes les opérations: créer, lire, supprimer..
Exemple: le format ext4 de votre disque dur	Assure la cohérence des données et les permissions

Le SGF utilise les règles du SF pour manipuler concrètement les données sur le support de stockage. Le SGF est l'implémentation du SF.

2.2. Structure interne d'un SF Ext

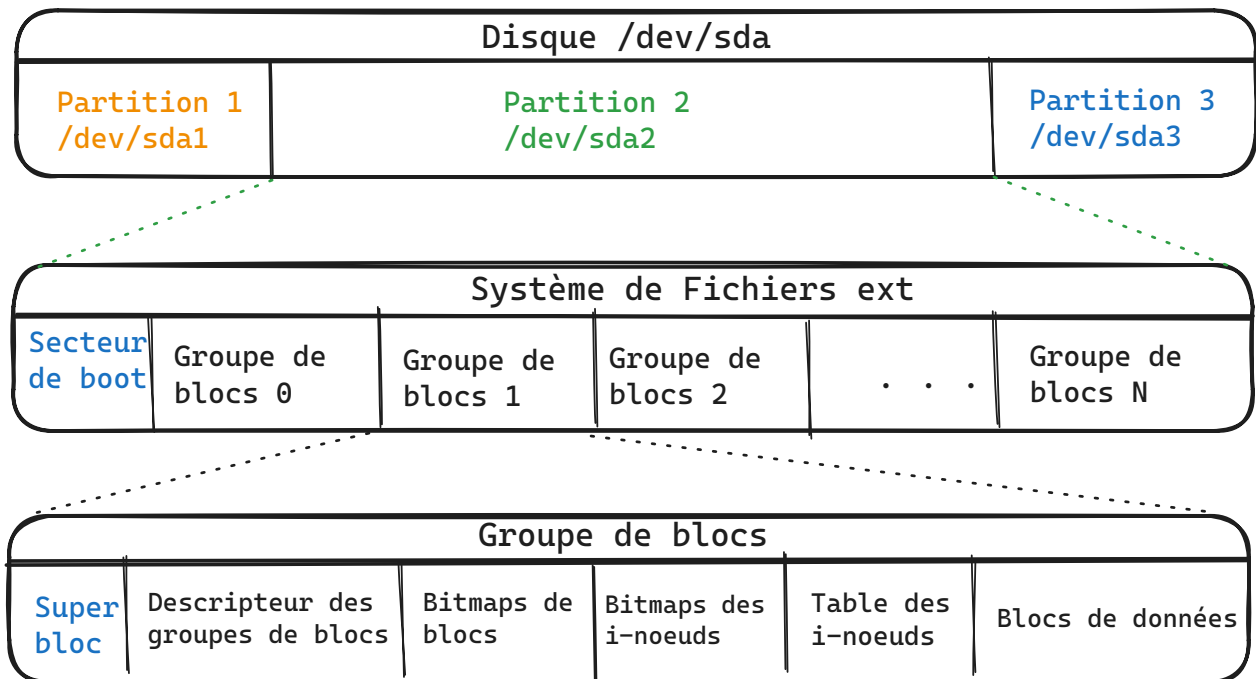


Schéma de la structure interne d'un SF Ext

Le Système de Gestion de Fichiers (SGF) navigue dans cette structure. Pour lire un fichier, il utilise la table des i-noeuds pour trouver les blocs de données correspondants sur le disque.

Voici le détail d'un groupe de blocs:

- **Superbloc** : Informations générales sur l'ensemble du système de fichiers (taille, nombre d'inodes, etc.).
- **Descripteurs de groupes de blocs** : Contiennent des pointeurs vers les structures clés (bitmaps et tables d'inodes) de chaque groupe.
- **Bitmap de blocs** : Une carte des blocs de données, indiquant s'ils sont libres ou occupés.

- **Bitmap d'inodes** : Une carte des i-noeuds, indiquant s'ils sont libres ou occupés.
- **Table des i-noeuds** : Le coeur du système. Chaque entrée (i-node) contient les métadonnées d'un fichier (emplacement, permissions, etc.).
- **Blocs de données** : Contiennent le contenu réel des fichiers.

2.3. Structure d'un i-noeud

Un i-noeud (ou inode « index node ») est une structure de données qui décrit un fichier. C'est l'identité d'un fichier, indépendante de son nom.

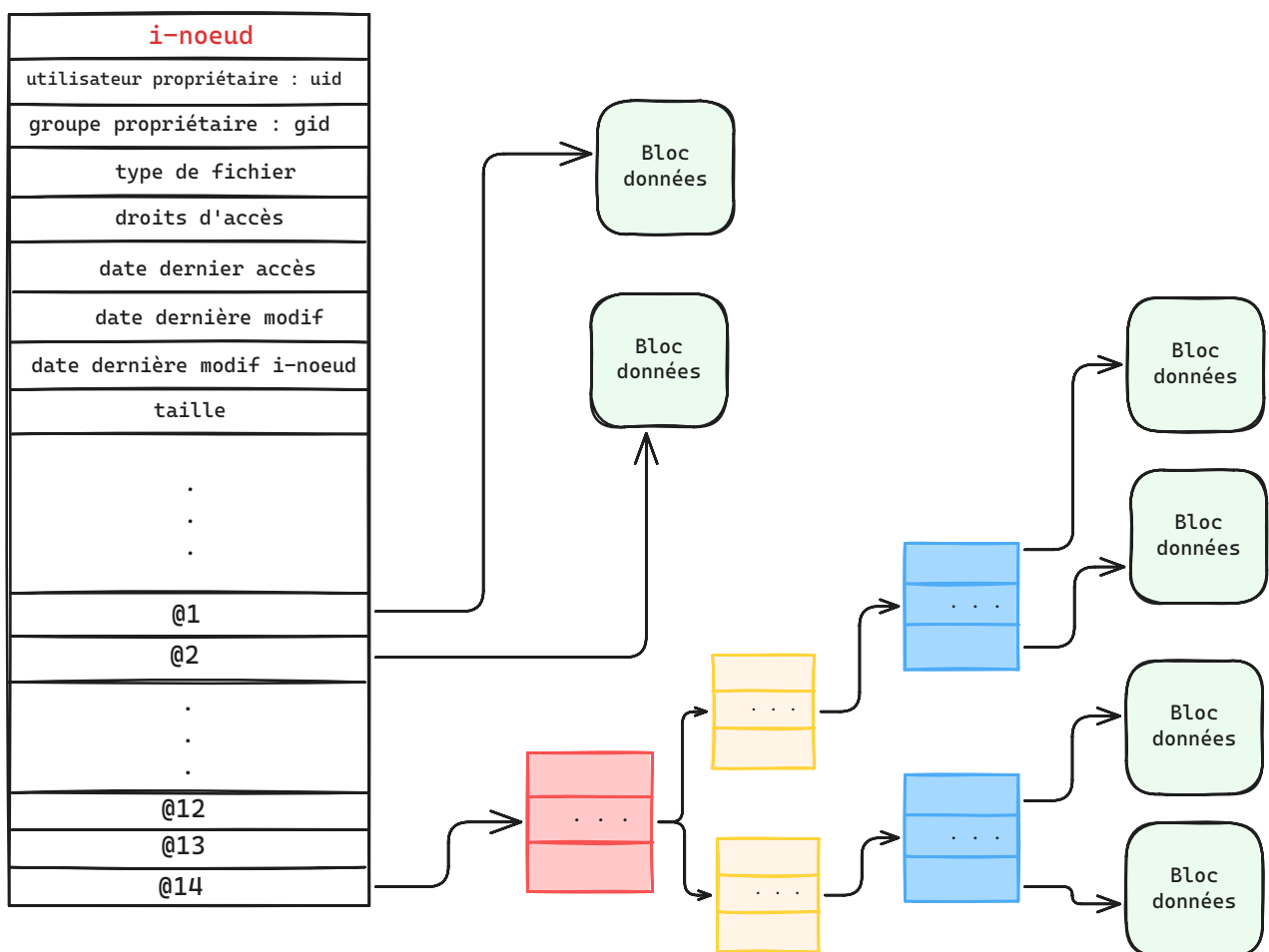


Schéma de la structure interne d'un i-noeud

Nous pouvons nous demander que contient vraiment un répertoire?

- Des noms de fichiers (lisible par l'utilisateur)
- Des numéros d'inode (référence du système)

2.3.1. Opérations sur l'i-noeud

Nous disposons de différents appels systèmes pour manipuler les i-noeuds:

```
1 int stat(const char *path, struct stat *buf);
2 int fstat(int fd, struct stat *buf);
3 int lstat(const char *path, struct stat *buf);
```

Attention tout de même : `lstat()` retourne les informations du lien symbolique lui même, et non pas celles du fichier cible

De plus nous disposons d'une structure de données `struct stat` voici une énumération des métadonnées principales ainsi que leur appellation.

```
1 dev_t st_dev; // Périphérique contenant le fichier
2 ino_t st_ino; // Numéro d'inode
3 mode_t st_mode; // Type et permissions du fichier
4 nlink_t st_nlink; // Nombre de liens physiques
5 uid_t st_uid; // UID du propriétaire
6 gid_t st_gid; // GID du propriétaire
7 off_t st_size; // Taille en octets
8 time_t st_atime; // Date dernier accès
9 time_t st_mtime; // Date dernière modification
10 time_t st_ctime; // Date dernier changement d'état
```

2.3.2. Permissions sur les fichiers

Les permissions de base:

- **Propriétaire** : `S_IRUSR` (r), `S_IWUSR` (w), `S_IXUSR` (x)
- **Groupe** : `S_IRGRP` (r), `S_IWGRP` (w), `S_IXGRP` (x)
- **Autres** : `S_IROTH` (r), `S_IWOTH` (w), `S_IXOTH` (x)

Nous disposons également d'appels systèmes afin de modifier ces permissions :

```
1 int chmod(const char *path, mode_t mode);
2 int fchmod(int fd, mode_t mode);
```

Il est également possible de modifier le propriétaire du fichier grâce aux appels systèmes suivants :

```
1 int chown(const char *path, uid_t owner, gid_t group);
2 int fchown(int fd, uid_t owner, gid_t group);
3 int lchown(const char *path, uid_t owner, gid_t
4 group);
```

Seul root peut modifier le propriétaire et lchown() agit sur le lien symbolique lui-même.

2.4. Opérations sur les fichiers

Nous disposons d'une multitude d'appels systèmes afin d'effectuer des opérations sur les fichiers, en voici une courte présentation, n'hésitez pas compléter avec la documentation officielle.

Pour gérer l'ouverture d'un fichier :

```
1 int open(const char *pathname, int flags, mode_t mode);
2 int creat(const char *pathname, mode_t mode);
```

Pour gérer la lecture / l'écriture dans un fichier :

```
1 ssize_t read(int fd, void *buf, size_t count);
2 ssize_t write(int fd, const void *buf, size_t count);
3 off_t lseek(int fd, off_t offset, int whence);
```

Pour gérer les descripteurs :

```
1 int dup(int oldfd);
2 int dup2(int oldfd, int newfd);
3 int close(int fd);
```

Pour gérer la taille d'un fichier :

```
1 int truncate(const char *path, off_t length);
2 int ftruncate(int fd, off_t length);
```

2.5. Tables système de manipulation de fichiers

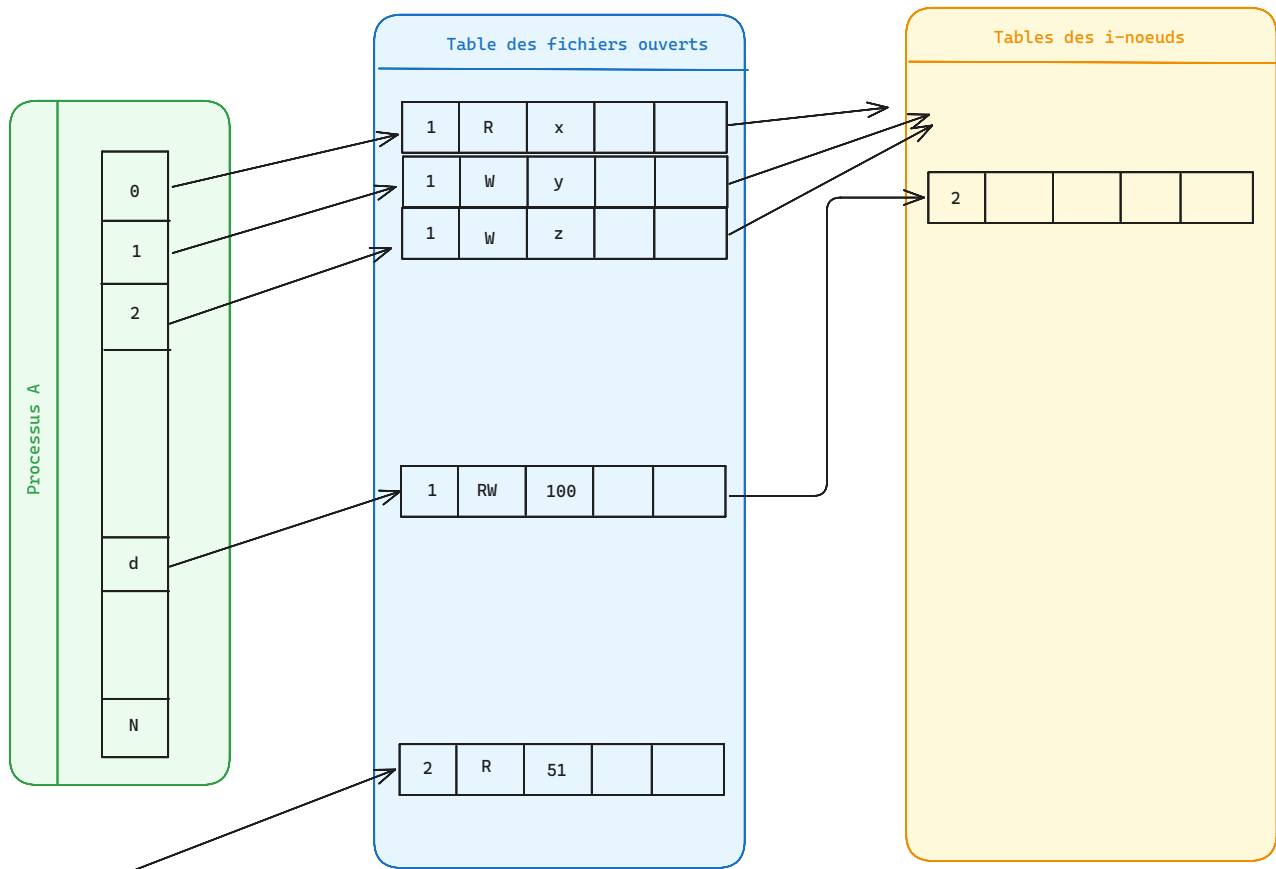


Schéma de la structure d'une table système

Les descripteurs de fichiers sont des entiers identifiant des fichiers / ressources ouvertes, la plage typique sous Linux est de 0 à 1024. Chaque processus possède sa propre table des descripteurs comme montrer ci-dessus.

Remarque: Les descripteurs peuvent également représenter des :

- Tubes (pipes)
- Sockets réseau
- Périphériques

2.6. Opération sur les fichiers

L'ouverture de fichiers:

```

1 # include <fcntl.h>
2 # include <sys/stat.h>
3
4 int open (const char *pathname, int flags);
5 int open (const char *pathname, int flags, mode_t mode);

```

Les modes d'ouverture :

Mode de base	Attributs de création	Attributs d'état
O_RDONLY	O_CREAT	O_APPEND
O_WRONLY	O_EXCL	O_SYNC
O_RDWR	O_TRUNC	

En ce qui concerne le mode d'ouverture il est à noter que nous avons un filtrage par `umask`. L'opération réalisée est donc le suivant:
`Permissions = mode & ~umask.`

2.6.1. L'effet de l'appel système `open()`:

Sur la table des descripteurs:

- Ajoute une ntrée avec le plus petit descripteur disponible
- Pointe vers une entrée de la table des fichiers ouverts

Sur la table des fichiers ouverts:

- Crée toujours une nouvelle entrée même si le fichier est déjà ouvert
- Chaque `open()` crée une entrée indépendante
- Stocke position = 0 ou taille fichier (si O_APPEND), flags, pointeur vers inode

Sur la table des i-noeuds:

- Charge l'inoeud en mémoire s'il n'est pas déjà présent
- Incrémente le compteur de référence) chaque `open()`
- Met à jour `st_atime` sur accès.

Toujours verifier la valeur de retour du `close`.

2.6.2. L'effet de l'appel système `close()`:

Sur la table des descripteurs:

- Libère l'entrée du descripteur de fichier
- Le descripteur devient disponible pour la réutilisation

Sur la table des fichiers ouverts:

- Décrémente le compteur de références.
- Supprime l'entrée si le compteur devient 0.
- Libère les buffers d'E/S associés (cache du système de fichiers, buffer réseau).

Sur la table des i-noeuds:

- Décrémente le compteur de références.
- Finalise l'écriture des données ou métadonnées modifiées sur le disque si nécessaire.
- Peut libérer l'i-noeud en mémoire s'il n'est plus référencé.

Nous disposons également de `read()`, `write()` & `lseek()`. Vous pouvez vous référer au cours de M.Hancart d'Algo1 qui fournit une spécification complète sur le sujet.

Il est à noter que la synchronisation des écritures est très intéressante pour les données critiques:

```
1 #include <unistd.h>
2 void sync ( void ) ; // Synchronise tous les buffers du système
3 int fsync ( int fd ) ; // Synchronise un fichier spécifique
4 int fdatasync ( int fd ) ; // Synchronise les données ( pas métadonnées )
```

2.6.3. Duplication de descripteur de fichier `dup` & `dup2`

```
1 #include <unistd.h>
2 int dup ( int oldfd ) ;
3 int dup2 ( int oldfd , int newfd ) ;
```


Ces appels permettent à deux descripteurs de pointer vers le même fichier ouvert, partageant ainsi la même position courante.

- `dup2` est plus sûr que la combinaison `close()` + `dup()`, notamment dans un environnement multi-thread.
- En effet, entre `close(STDOUT_FILENO)` et `dup()`, un autre thread peut ouvrir un descripteur qui réutilisera `STDOUT`, entraînant un comportement inattendu.
- `dup2` évite ce risque en réalisant l'opération de manière atomique

2.7. Opérations sur les répertoires

- Les fichiers sont organisés en répertoires (dossiers)
- Structure en arborescence avec imbrication
- Le répertoire racine est la base de l'arborescence
- Sous UNIX : symbolisé par `/`

2.8. Implémentation et manipulation

Nous disposons de deux fonctions pour changer de répertoire:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main() {
5      char cwd[1024];
6      getcwd(cwd, sizeof(cwd));
7      printf("Avant: %s\n", cwd);
8      if (chdir("/tmp") == -1) {
9          perror("chdir");
10         return EXIT_FAILURE;
11     }
12     getcwd(cwd, sizeof(cwd));
13     printf("Après: %s\n", cwd);
14     return EXIT_SUCCESS;
15 }
```

et donc `getcwd()` pour récupérer le répertoire courant.

2.9. Manipulation des répertoires

Fonctions principales:

- `Aopendir()` : ouvre un flux répertoire
- `Areaddir()` : lit une entrée
- `Aclosedir()` : ferme le flux

Nous avons également accès à la structure de données `dirent` définit comme suit:

```
1 struct dirent {  
2     ino_t d_ino ; // numéro i-noeud  
3     char d_name []; // nom du fichier  
4 };
```

Fonctions avancées:

- `rewinddir()` : retour au début
- `tellldir` : position courante
- `scandir()` : lecture avec filtrage

2.10. Création et suppression

Pour la création nous passons par la fonction :

```
1 int mkdir(const char *pathname, mode_t mode);
```

Pour renommer / déplacer des fichiers:

```
1 int rename(const char *oldpath, const char *newpath);
```

2.11. Récapitulatif des appels système

Appel	Type	Description	Retour
open	Fichier	Ouvrir/créer un fichier	Descripteur
close	Fichier	Fermer un descripteur	0 / -1
read	Fichier	Lire des données	Octets lus
write	Fichier	Écrire des données	Octets écrits
opendir	Répertoire	Ouvrir un répertoire	DIR*
readdir	Répertoire	Lire une entrée	dirent*
closedir	Répertoire	Fermer le répertoire	0 / -1
mkdir	Répertoire	Créer un répertoire	0 / -1
rmdir	Répertoire	Supprimer répertoire vide	0 / -1
unlink	Les deux	Supprimer un lien	0 / -1
link	Lien	Créer lien physique	0 / -1
symlink	Lien	Créer lien symbolique	0 / -1
readlink	Lien	Lire cible lien symbolique	Octets lus

Points clés à retenir

Fichiers : manipulation du contenu (read/write)

Répertoires : parcours et organisation (opendir/readdir)

Liens : multiples noms pour un fichier (link/symlink)

Gestion : création/suppression (mkdir/rmdir/unlink)

3. Chapitre 3: Les tubes

Un tube est un canal de communication unidirectionnel qui relie un processus écrivain (qui envoie des données) à un processus lecteur (qui les reçoit). Les données circulent selon une logique FIFO (First In, First Out) et sont stockées temporairement dans un tampon mémoire géré par le noyau.

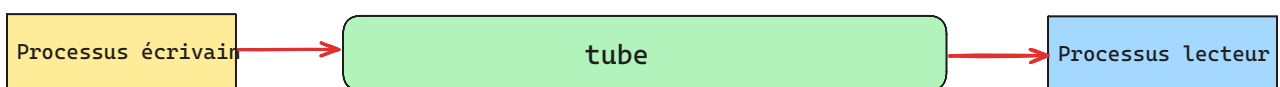


Schéma de la structure d'un tube

3.1. Types de tubes sous Linux

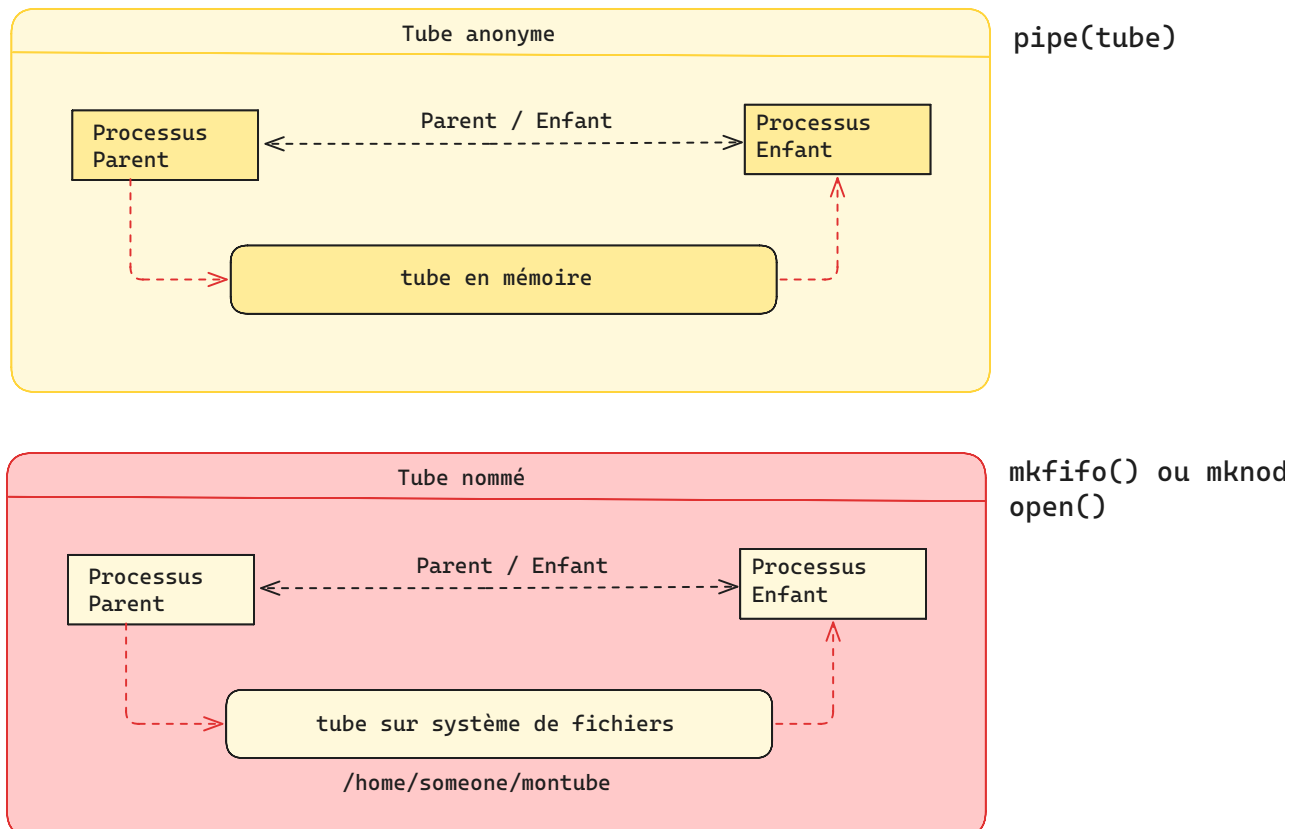


Schéma de la structure interne d'un tube anonyme & nommé

3.1.1. Tubes nommés vs tubes anonymes

Type	Création	Portée	Persistance	Visibilité
Tubes Anonymes	pipe()	Processus parent enfant uniquement	Durée de vie des processus	Invisible dans le système de fichiers
Tubes Nommés (FIFOs)	mkfifo() ou mknod()	Tous les processus du système	Durée du système de fichiers	Fichier spécial (type 'p')

Les données dans le tube sont volatiles et stockées uniquement dans les buers noyau.

Exemple shell:

```
1 Tube anonyme
2 ps aux | wc -l
3 # Tube nommé
4 mkfifo tube
5 echo " test " > tube &
6 cat < tube
```

3.2. Création et ouverture tube anonyme

Nous utilisons l'appel système `pipe()` :

```
1 #include <unistd.h>
2
3 int tube[2]
4 int pipe(int tube[2])
```

Description : Créer un tube et retourne deux descripteurs de fichiers :

^- tube[0] : Extrémité de lecture.

^- tube[1] : Extrémité d'écriture. ^ Retourne 0 en cas de succès, -1 en cas d'erreur.

Effet sur les tables système:

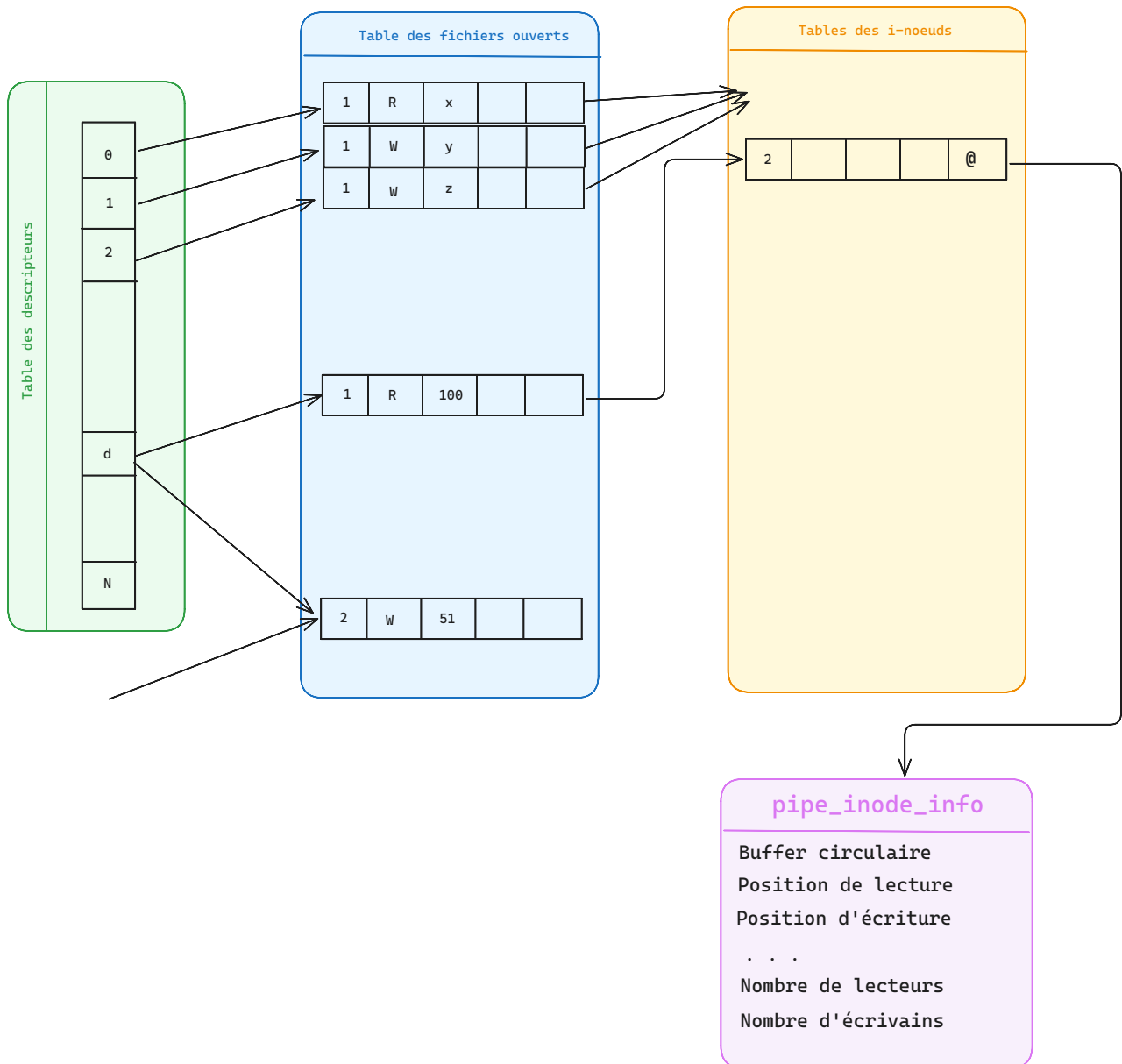


Schéma de l'effet du `pipe()` sur les tables système

Séquence d'exécution de pipe():

1. Allocation inode virtuel dans la table des inodes en mémoire
2. Création structure pipe_inode_info en mémoire
3. Lien inode vers structure pipe
4. Création deux entrées(resp. Read et Write) dans la table des chiers
5. Lien chier vers inode virtuel
6. Allocation de deux descripteurs dans le processus
7. Lien descripteurs vers entrée chiers respectives
8. Initialisation compteurs (readers=1, writers=1)

3.2.1. Contenu de l'i-noeud virtuel d'un tube

Champ	Valeur pour un tube
i_mode	S_IFIFO (fichier FIFO)
i_pipe	Pointeur vers pipe_inode_info
i_size	Taille des données dans le tampon
i_count	Compteur de références
i_op	Opérations pipes (pipe_iops)
i_fop	Opérations fichiers (pipefifo_fops)

3.2.2. Structure `pipe_inode_info`

Champ	Description
<code>bufs</code>	Tableau de buffers circulaires
<code>head</code>	Position de lecture
<code>tail</code>	Position d'écriture
<code>readers</code>	Nombre de lecteurs actifs
<code>writers</code>	Nombre d'écrivains actifs
<code>wait</code>	File d'attente pour blocage
<code>lock</code>	Verrou pour synchronisation
<code>r_counter</code>	Compteur de lectures
<code>w_counter</code>	Compteur d'écritures

3.3. Création et ouverture tube nommé

Nous utilisons les appels système `mkfifo()` & `open()`.

```
1 #include <unistd.h>
2 int mkfifo(const char *pathname, mode_t mode);
3 int open(const char *pathname, int flags);
```

Description : ^

- Crée un tube nommé `pathname` dans le système de fichiers avec les permissions `mode`.
- Retourne 0 en cas de succès, -1 en cas d'erreur.

L'appel `open` pour un tube nommé en mode non-bloquant (`O_NONBLOCK`) diffère selon le mode lecture ou écriture:

^Lecture (`open(« montube », O_RDONLY | O_NONBLOCK)`):

- Ouvre immédiatement, retourne un descripteur même sans écrivain.

- Raison : Un lecteur peut attendre des données ou gérer leur absence (read retourne EAGAIN ou 0 pour EOF). Permet l'asynchronisme.

Écriture (open(« montube », O_WRONLY | O_NONBLOCK)) :

- Ouvre immédiatement, mais échoue avec ENXIO si aucun lecteur.
- Raison : Écrire sans lecteur est inutile (mènerait à EPIPE). ENXIO signale tôt l'absence de communication possible.

3.3.1. Définition des rôles

- Écrivain : Tout processus disposant d'un descripteur ouvert en écriture sur le tube
- Lecteur : Tout processus disposant d'un descripteur ouvert en lecture sur le tube

Conséquences Importantes:

Multiple écrivains possibles sur un même tube

Multiple lecteurs possibles sur un même tube

Un processus peut être lecteur et écrivain

Les rôles sont dénis par les descripteurs ouverts

3.4. Lecture sur un tube

Pour lire nous utilisons l'appel système `read()`

```
1 ssize_t read (int fd, void *buf, size_t count);
```



Description :

^Lit jusqu'à count octets depuis le descripteur fd (tube[0]) dans le buffer buf. Retourne le nombre d'octets lus (peut être < count), 0 pour EOF, ou -1 en cas d'erreur

Attention le comportement diffère selon le mode!

3.4.1. Mode bloquant (Par défaut)

Description :

- Si des données sont disponibles : Lit les données immédiatement.
- Si aucune donnée n'est disponible :
- Bloque jusqu'à ce que des données arrivent (via write()).
- Ou jusqu'à ce que tous les écrivains ferment leurs descripteurs d'écriture (EOF, read retourne 0).
- Utile pour la synchronisation de processus.

3.4.2. Mode non-bloquant

Activation du mode non-bloquant:

```
1 fcntl(tube[0], F_SETFL, O_NONBLOCK); // tube anonyme
2 open("montube", O_RDONLY | O_NONBLOCK); // tube nommé
```

Description :

- Si des données sont disponibles : Lit les données.
- Si aucune donnée : Retourne immédiatement -1 avec errno = EAGAIN ou EWOULDBLOCK.
- Ne bloque pas : Idéal pour des lectures asynchrones.

3.5. Écriture dans un tube

Condition d'écriture atomique :

- Lorsque la taille des données \leq PIPE_BUF (généralement 4096 octets)
- L'écriture est atomique : pas d'entrelacement avec d'autres écrivains

Vérification de PIPE_BUF :

```
1 #include <unistd.h>
2 printf("PIPE_BUF = %ld\n", fpathconf(fd, _PC_PIPE_BUF));
```

Exemple d'écriture atomique :

```
1 char message [256];
2 snprintf(message, sizeof(message), "Mon pid : %d", getpid());
3 // Écriture atomique si sizeof (message) <= PIPE_BUF
4 write(fd, message, strlen (message) + 1) ;
```

Attention : Pour les données > PIPE_BUF, les écritures peuvent être intercalées entre plusieurs processus.