

Programmation Système

INSA CVL

—

Table des matières

1 Plan du Cours

2 Introduction

3 Système d'exploitation

4 Principes des systèmes
d'exploitation

5 Utilisation de la bibliothèque C
standard et de l'API POSIX

6 Les Entrées-Sorties

7 Système de fichiers

8 Processus

9 Bibliographie

Sommaire

1 Plan du Cours

2 Introduction

3 Système d'exploitation

4 Principes des systèmes
d'exploitation

5 Utilisation de la bibliothèque C
standard et de l'API POSIX

6 Les Entrées-Sorties

7 Système de fichiers

8 Processus

9 Bibliographie

Objectif du cours

- Comprendre ce qu'est un Système d'Exploitation
- Comprendre comment fonctionne un Système d'Exploitation
- Etudier les bases de la programmation Système

Plan du Cours

■ Introduction

- Système d'exploitation
- Historique UNIX
- Mode d'exécution d'un processeur
- Norme (Unix- \rightarrow POSIX- \rightarrow SUS)

■ Utilisation de la bibliothèque C standard et de l'API POSIX

- interagir avec le système d'exploitation
- Appel système
- Cycle d'exécution d'un programme
- Accès à l'environnement
- Gestion des erreurs
- utilisation appels système
- implantation de la bibliothèque au dessus des appels système

Plan du Cours

- Les entrées-sorties
 - Généralités
 - Manipulation des i-noeuds
 - Primitives de base
 - Descripteurs
- Système de fichiers
 - Exemple : ext2, FAT32
- Processus
 - création, terminaison, interruptions, ordonnancement
 - Gestion des processus
 - Attributs des processus
 - Vie des processus

■ Qu'est-ce qu'un système d'exploitation ?

Qu'est-ce qu'un système d'exploitation ?

■ Qu'est-ce qu'un système d'exploitation ?

def1 Programme qui agit comme un intermédiaire entre **l'utilisateur** d'un ordinateur et le matériel ; il fournit un environnement dans lequel l'utilisateur peut exécuter des **programmes** de manière *pratique et efficace*.

def2 Le **système d'exploitation**, abrégé SE (en anglais *operating system*, abrégé OS), est l'ensemble de programmes central d'un appareil informatique qui sert d'interface entre le **matériel** et les **logiciels applicatifs**.

Qu'est-ce qu'un système d'exploitation ?

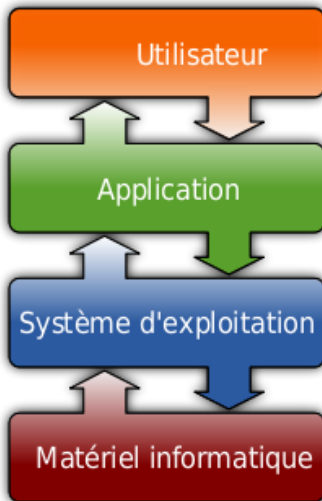
- Ordinateur = ensemble de **ressources physiques**
 - Processeur/Unité Centrale (CPU)
 - Mémoire principale
 - Mémoires secondaires
 - Périphériques d'E/S
 - Périphériques internes (horloge,...)

- Son utilisation génère des **ressources logiques**
 - Processus
 - Fichiers
 - Bibliothèques Â« Système Â» partagées
 - Sessions utilisateurs

Qu'est-ce qu'un système d'exploitation ?

- Un système d'exploitation :
 - **Alloue** les ressources aux utilisateurs
 - **Contrôle** leur bonne utilisation
 - Problèmes : efficacité, fiabilité, sécurité, équité, etc.

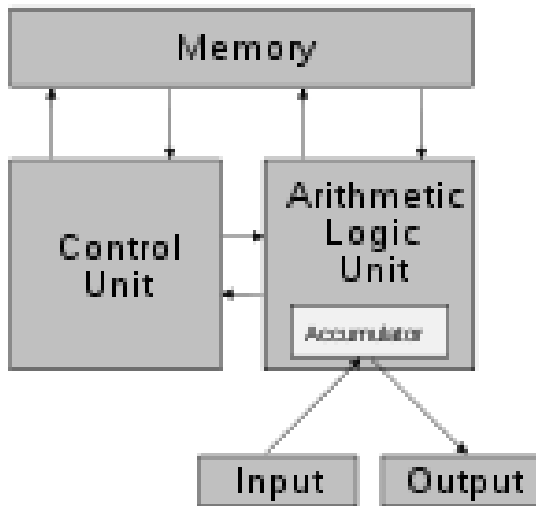
Place du système d'exploitation



Historique des systèmes informatiques

- 1936-55 : Les premiers ordinateurs
- 1955-65 : Traitement par lots
- 1965 : Tamponnement des entrées/sorties
- 1965-70 : Multiprogrammation
- 1970 : Temps partagé

Architecture de Von Neumann



Ordinateur :

- Machine programmable
 - Traite des informations *numériques* ou *discrètes* (\neq *analogiques* ou *continues*)
-
- Apparition des premiers ordinateurs :
 - à relais et à tubes vides
 - programmés par tableaux de connecteurs, puis par cartes perforées (1950)
 - **mono-utilisateur** et **mono-tâches**
 - apparition du terme **BUG**

Première Génération (1936 - 1955)

Ordinateur :

- Machine programmable
- Traite des informations *numériques* ou *discrètes* (\neq *analogiques* ou *continues*)
- Apparition des premiers ordinateurs :
 - à relais et à tubes vides
 - programmés par tableaux de connecteurs, puis par cartes perforées (1950)
 - **mono-utilisateur** et **mono-tâches**
 - apparition du terme **BUG**

Première Génération (1936 - 1955)

- Séance-type de **programmation** :
 - Ecriture sur cartes (programmeur)
 - Chargement des cartes compilateur (opérateur)
 - Chargement des cartes du programme
 - Création du code intermédiaire (assemblage)
 - Chargement des cartes de l'assembleur
 - Création du code en langage machine
 - Exécution du programme

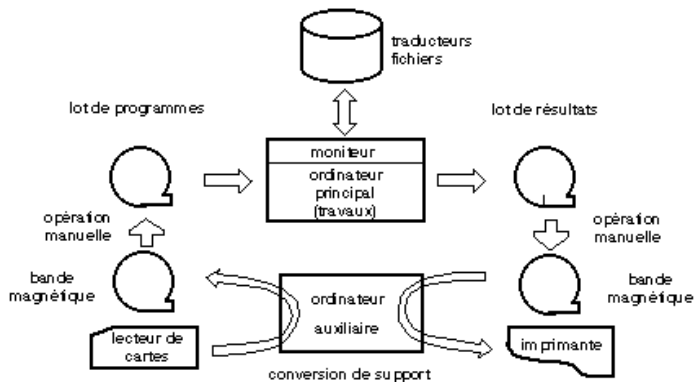
Problèmes :

- Exécution des instructions d'un programme sans intervention extérieure possible
- Temps d'inactivité importants : matériel sous-employé

Deuxième Génération (1955 - 1965)

- Traitement par lots :
 - Emergence des supports magnétiques
 - conservation de programmes binaires importants sur ce support
 - Améliorations :
 - **Regroupement** et **exécution par groupe** des travaux similaires (batch processing)
 - **Moniteur résident** : enchaîne automatiquement les travaux

Deuxième Génération (1955 - 1965)



Troisième Génération (1965 - 1980)

- Apparition des circuits intégrés
- 1965 : Tamponnement des E/S
 - **Idée** : rendre les opérations d'E/S autonomes
 - Utilisation de **tampons** (Â« buffers Â»)
 - Stocker les données lues non encore nécessaires
 - Stocker les requêtes de sorties quand le périphérique n'est pas disponible
 - Introduction du mécanisme d'interruption

Avantages :

- Diminution coût/performance
- Plus d'ordinateurs annexes/manipulation de bandes

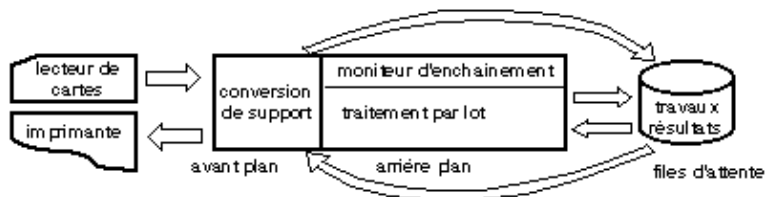
Troisième Génération (1965 - 1980)

- Apparition des circuits intégrés
- 1965 : Tamponnement des E/S
 - **Idée** : rendre les opérations d'E/S autonomes
 - Utilisation de **tampons** (Â« buffers Â»)
 - Stocker les données lues non encore nécessaires
 - Stocker les requêtes de sorties quand le périphérique n'est pas disponible
 - Introduction du mécanisme d'interruption

Avantages :

- Diminution coût/performance
- Plus d'ordinateurs annexes/manipulation de bandes

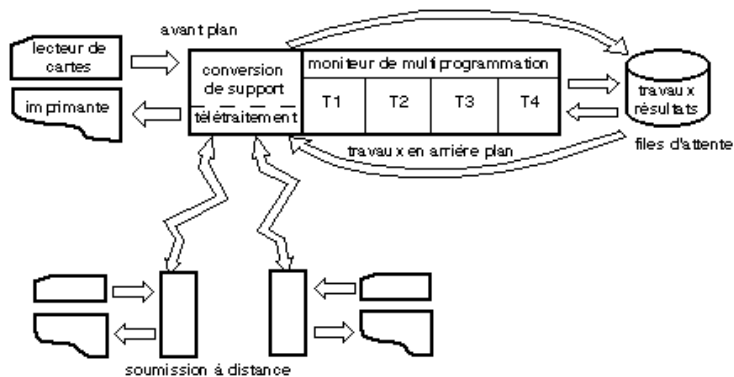
Troisième Génération (1965 - 1980)



Troisième Génération (1965 - 1980)

- 1965-70 : La **multiprogrammation** :
 - Augmentation de la taille de la mémoire principale –> multiprogrammation
 - Charger plusieurs travaux en mémoire simultanément
 - Faire un autre travail au lieu d'attendre
 - Amélioration : exécution parallèle d'un ensemble de programmes
 - Les E/S sont effectuées de façon asynchrone avec des calculs sur l'UC
- *Simultaneous Peripheral Operation On-Line* (**SPOOL**)
 - Disques magnétiques plus rapides
 - Utilisés comme un gros tampon pour plusieurs travaux en même temps

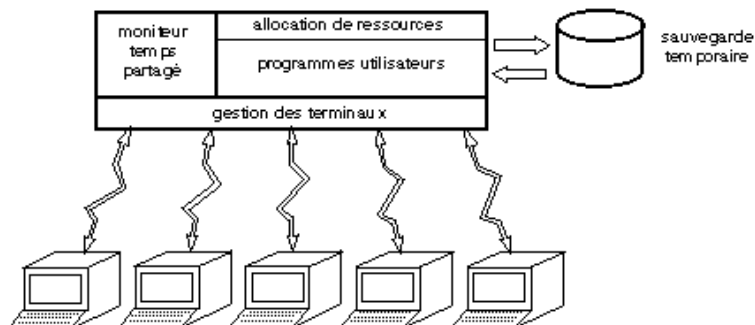
Troisième Génération (1965 - 1980)



Troisième Génération (1965 - 1980)

- 1970 : le temps partagé :
 - Améliorations :
 - Partage du temps en quanta via une horloge temps réel
 - Affectation du processeur à un programme durant un quantum
 - Prise en compte rapide des nouveaux programmes

Troisième Génération (1965 - 1980)



Types de système d'exploitation

- Il n'existe pas de système d'exploitation efficace dans tous les contextes d'utilisation.
- On définit donc différentes catégories/familles de systèmes :
 - Mono-utilisateur
 - Contrôle de processus
 - Serveurs de fichiers
 - Transactionnel
 - Général

Mono-utilisateur

- Accepte un seul utilisateur à un moment donnée
- Construit autour d'une machine virtuelle simple
- Facilite l'utilisation des différents périphériques
- Peut être multi-tâches
- Pas de notion d'usager ou de protection

Controle de processus

- Utilisé principalement dans le milieu industriel
- Controle de machines-outils ou systèmes complexes
- Permet de réagir en un temps garanti à des événements issues de capteurs
- Fortement orienté temps réel
- Doit être fiable et tolérant aux pannes

Serveurs de fichiers

- Contrôle de gros ensembles d'information
- Interrogeable à distance
- Temps de réponse court, mise à jour à la volée
- L'utilisateur ignore la structuration interne du système de fichiers

Transactionnel

- Contrôle de grosse base de données modifiées fréquemment
 - Doit garantir un temps de réponse court
 - Doit garantir la cohérence constante de la base de donnée
 - Doit permettre la résolution des conflits
- => Définition de transactions atomiques et de points de reprise

Général

- Accueille simultanément plusieurs utilisateurs effectuant des tâches différentes
- Multi-Utilisateur
- Multi-Tâches
- Entrées/Sorties variés
- Logiciels de différents types
- Peut disposer de capacités interactives
- Peut disposer de traitement par lots

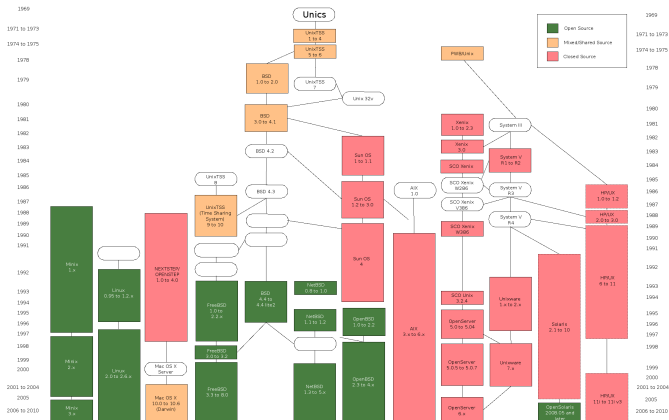
Unix

- Système d'exploitation créé en 1969 sous l'impulsion de Kenneth Thompson et Dennis Ritchie
- Au coeur du développement informatique des trente dernières années.
- Multi-utilisateurs
- Multi-tâches
- Temps partagé

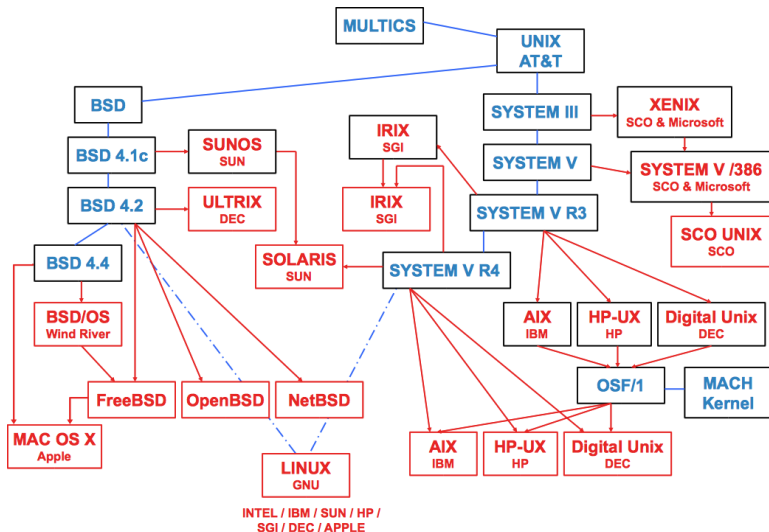
Linux

- Basé sur Unix
- 1984 : lancement du projet GNU par Richard M. Stallman
- 1991 : Linus Torvalds propose la première version d'un noyau baptisé Linux
- 1992 : système d'exploitation GNU/Linux
- Disponible sous la forme de multiples distributions

Evolution d'Unix



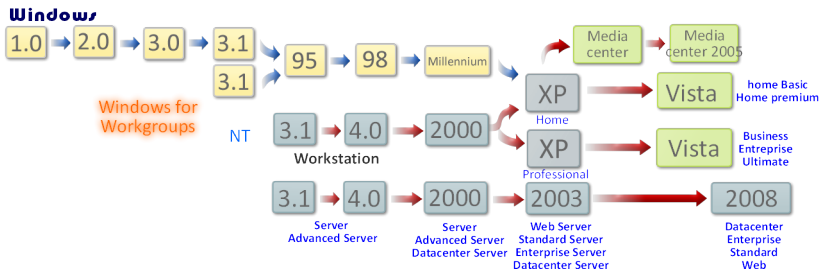
Evolution d'Unix



MS-DOS

- Lancé en 1981, en même temps que le PC d'IBM
- Mono-utilisateur
- Mono-tâche
- Couche graphique : Windows... jusqu'à Windows 2000
- Désormais, l'invite de commande est un émulateur intégré.

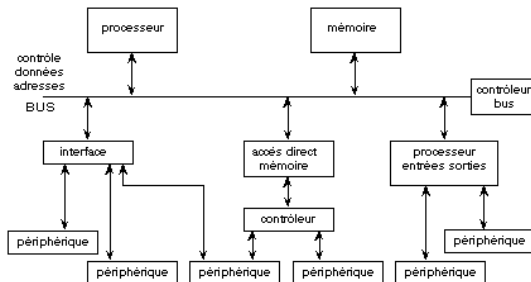
Evolution de Microsoft Windows



Architecture Générale

■ Ordinateur :

- Processeur (traitements)
- Mémoire principale (rangement des données/résultats)
- Périphériques (échange d'informations avec l'extérieur)
- Bus (liaison entre les constituants)



Structures internes des systèmes d'exploitation généraux



■ Types de systèmes :

- Systèmes monolithiques
- Systèmes en couche
- Machines virtuelles
- Systèmes client-serveur

Systèmes monolithiques

- Organisation la plus répandue
- Caractérisé par l'absence de structure interne
- Le système est une collection de procédure, chacune visible de tous les autres, et pouvant appeler toute autre procédure qui lui est utile
- La seule barrière est la protection entre le monde utilisateur et le monde noyau

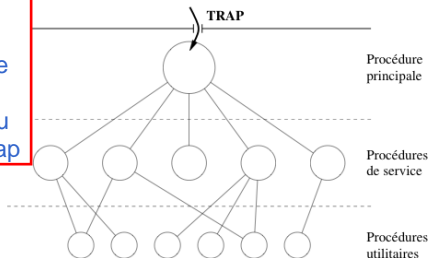
Systèmes monolithiques

■ Structures internes sur 3 niveaux :

- **Procédure principale** exécuté lors de chaque **appel système** (notion de **trap**)
- Les **procédures de service** sont dédiées au traitement de chaque appel système
- Les **procédures utilitaires** assistent les procédures de service

=¿ **Structure d'aucune protection contre les erreurs**

Dans le trap il y a une instruction c'est le processeur qui relève le noyau. par ex: quand vous changer l'heure du systeme ca lance un trap



52/310

52/310

Machines Virtuelles

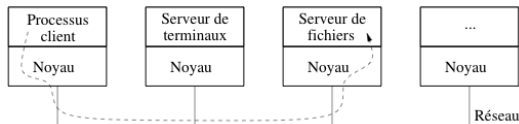
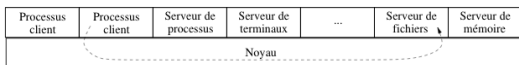
- Un système à temps partagé offre, à la fois,
 - la possibilité de partager le processeur,
 - et une machine virtuelle dotée d'une interface plus pratique que la programmation directe du matériel
- Fonctionnement :
 - Un moniteur de machine virtuelle s'exécute juste au dessus du matériel
 - Chaque machine virtuelle fait tourner un SE

Application	Application	Application
CMS	CMS	CMS
VM/370		
Matériel du 370		

Systèmes client-serveur

- Basé sur une approche horizontale
- SE vu comme un système distribuée
 - Notion de **micro-noyau**
 - Noyau GNU HURD

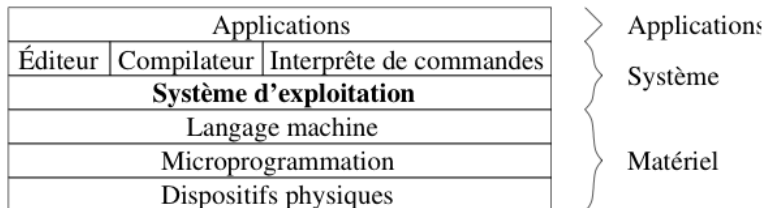
=> Problèmes de latences



Rappel : Modes d'exécution d'un processeur

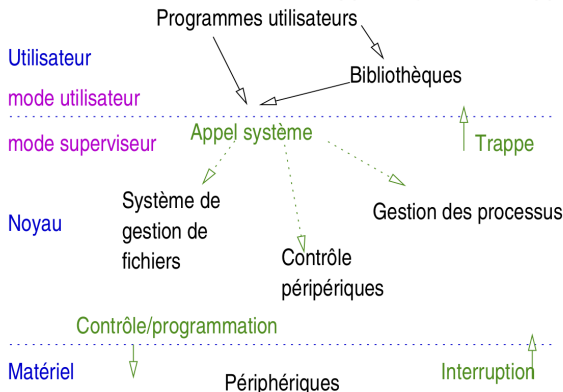
- Différencier les modes d'exécution
 - Mode **Utilisateur**
 - Mode **Superviseur** (ou mode moniteur/ système/ privilégié)
- Fonctionnement :
 - A l'initialisation du système, le matériel démarre en mode superviseur.
 - L'OS est chargé et démarre des processus utilisateur en mode utilisateur.
 - Lors d'un déroutement/interruption, la matériel passe en mode superviseur.
 - Le système revient toujours en mode utilisateur avant de passer le contrôle à un programme utilisateur.

Place du système d'exploitation dans l'ordinateur



- Utilisateur
- Superviseur

- Passage mode superviseur : appels système, trappes, IT



Interruptions

les appels peuvent être lancés à partir d'un programme utilisateur. Il y a un milliard d'appels système actuellement

- Fonctions généralement communes aux mécanismes d'interruption :
 - Appel de la routine de traitement via une table de pointeurs
 - Sauvegarde de l'adresse de l'instruction interrompue
 - Après traitement de l'interruption, chargement de l'adresse de retour dans le compteur ordinal
- Les systèmes d'exploitation modernes sont dirigés par les interruptions.

Structure en couche d'un Unix

- Niveau utilisateur
 - applications utilisateurs
 - logiciel de base
 - bibliothèques système
 - appels de fonctions
- Niveau Noyau
 - gestion des processus
 - système de fichiers
 - gestion de la mémoire
- Niveau matériel
- Système d'exploitation
 - bibliothèques système
 - noyau

- A set of small navigation icons typically found in Beamer presentations, including symbols for back, forward, search, and other slide controls.

Normalisation de l'interface

■ Unix

devient

- système d'exploitation Ken THOMPSON et Dennis RITCHIE, Bell Labs, 1969
- distribution du code source
- multiples versions (branches BSD, System III...)

■ POSIX

ensuite

- Portable Open System Interface
- eXchange Portable Open System Interface X for Unix
- standard IEEE, 1985
- interface standardisée des services fournis par le système

■ Single Unix Specification, SUS

- X/Open reprend les activités de normalisation POSIX, 1999
- The Open Group (ex X/Open) propose Single Unix Specification version 3, 2001
- www.unix.org/version3/

L'objectif d'une norme c'est de permettre a un programme d'operer sur n'importe quel systeme. Ce sont des guidelines pour les programmeurs

Si vous voulez respecter les normes POSIX en gros il faut code en C

Dans certains secteurs on peut vous imposer une norme (l'aviation, le ferroviaire)

Normalisation de l'interface

Norme	Symbole	Valeur
POSIX.1-1988	_POSIX_SOURCE	
POSIX.1-1990 (1003.1)	_POSIX_C_SOURCE	1
POSIX.1b-1993	_POSIX_C_SOURCE	199309L
POSIX.1c-1996	_POSIX_C_SOURCE	199506L
POSIX.1-2001	_POSIX_C_SOURCE	200112L
XPG3	_XOPEN_SOURCE	1
XPG4	_XOPEN_SOURCE	4
XPG4	_XOPEN_VERSION	4
SUS	_XOPEN_SOURCE	1
SUS	_XOPEN_SOURCE_EXTENDED	4
SUSv2	_XOPEN_SOURCE	500

Pour compiler de manière conforme à POSIX.1 :

```
gcc -D_POSIX_C_SOURCE=1 -c source.c
```

Fourniture de l'interface POSIX

- Norme POSIX = interface d'utilisation du système
 - description des fonctions d'appel des services système fournis par le noyau
 - portabilité des applications
 - ne définit pas la construction du système d'exploitation, noyau
- POSIX et l'interface d'un système Unix
 - interface POSIX = l'interface du système !
 - interface native
 - une fonction POSIX = un appel système Unix
- POSIX et l'interface de systèmes propriétaires
 - Windows, VMS, Mach...
 - interface POSIX \neq interface du système
 - bibliothèque niveau utilisateur au dessus des appels système
 - une fonction POSIX = un appel fonction bibliothèque utilisateur = un / multiples appels système

- Interface POS

- Fonctions : appels systèmes

- Basculement du *user mode* au *kernel mode* (*trap*)
- Exécution en mode noyau
- Interruption du processus courant (sauvegarde)

- Exécute des opérations "dangereuses"

- Liste des appels systèmes : `syscall.h`

le noyau regarde
dans le registre
processeur

syscall.h contient l'ensemble des appels systemes

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

```
int syscall (int numero...
```

Interface POSIX II

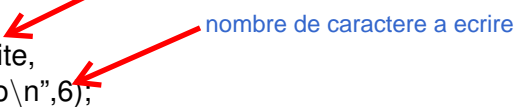
Exemple d'appel système :

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

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

```
int main ()
```

```
{
    (void)syscall(SYS_write,
    STDOUT_FILENO, "hello\n",6);
}
```



numero de l'appel system write

nombre de caractere a ecrire

A chaque fois qu'on fait un appel systeme on passe d'un contexte utilisateur au noyau

Performances : éviter les appels systèmes multiples !

on fera un bench mark en TD pour verifier les appels systemes et temps d'execution. Comme solution pour optimiser le code, des developpeurs utilisent directement du code assembleur pour gagner du temps de calcul

Interface POSIX III

■ sys/types.h : types de base

Type	Description
dev_t	Numéro de périphérique
uid_t	Identifiant de l'utilisateur
gid_t	Identifiant de groupe d'utilisateurs
ino_t	Identifiant de fichier (numéro de série)
mode_t	Droits d'accès et types de fichiers (masque)
nlink_t	Compteur de liens
off_t	Taille de fichier et déplacement
pid_t	Identifiant de processus
fsid_t	Identifiant de système de fichiers
size_t	Taille
ssize_t	Taille signée

definit a la fois un ensembles de fonctions que le systemes doit implementer, ca definit aussi des types generiques (pas facile a utilise), les valeurs de ces types sont completement dependant de l'architecture

Gestion des erreurs

- Code de retour d'erreur : -1
- Utilisation de errno :

```
#include <errno.h> //(sous Linux redirection vers <asm/errno.h>)
```

```
extern int errno;
```

```
#include <errno.h>
```

```
void perror (const char *message); // affiche le message
```

```
char * strerror (int symboleErreur); // retourne le message
```

Erreurs répertoriées par POSIX (extrait) :

Symbole	Signification
EPERM	Opération non autorisée
ENOENT	Fichier ou répertoire inexistant
ESRCH	Processus inexistant
EBADF	Descripteur d'E/S non valide

- Environnement, Terminaison, Commandes systèmes
- Constantes POSIX, Bases de données

Programmation système en C

- Langage C pour programmer le système
 - sémantique claire
 - efficacité
 - accès à toutes les structures de la machine (registres, bits...)
 - allocation mémoire explicite
 - autres approches possibles : langage dédié
- Langage C interface naturelle avec le système
 - bibliothèques écrites en C
 - utilisation de la bibliothèque depuis le C
 - autres approches possibles : Java, OCaml

- normalisation

- man 3 malloc

- non

- man 2 sbrk

- historic

- définitions communes

Bibliothèque et appel système I

- Appel système semblable à un appel de fonction de bibliothèque
 - comme des appels de fonctions C
- Appel système différent d'un appel de fonction de bibliothèque
 - appel système
 - pas d'édition de liens
 - exécution de code système
 - bibliothèque standard
 - abstraction de plus haut niveau
 - édition de liens avec la bibliothèque

Bibliothèque et appel système II

■ Appels système

- Manipulation du système de fichiers et entrées/sorties
- Gestion des processus
 - processus = exécution d'un programme
 - allocation de ressources pour les processus (mémoire...)
 - lancement, arrêt, ordonnancement des processus
- Communications entre processus Informations

Bibliothèque et appel système III

Terminaison d'un appel système

- Sémantique POSIX d'une primitive
 - comportement
 - y compris en cas d'erreur
 - liste des erreurs pouvant être retournées
 - voir le manuel man
- Valeur de retour d'un appel système
 - retourne -1 en cas d'erreur
 - positionne la variable globale errno
 - perror() produit un message décrivant la dernière erreur (appel système ou fonction bibliothèque)
- Exemple typique

Bibliothèque et appel système IV

```
if (creat(pathname, O_RDWR) == -1) { perror("Creation  
de mon fichier");  
}
```

- Test systématique des retours des fonctions (laborieux...)

Bibliothèque et appel système V

Bibliothèques standard

- Nombreuses bibliothèques
 - entrées/sorties formatées & bufferisées
 - localisation (français...) fonctions mathématiques
 - allocation mémoire dynamique
 - etc.
- Abstraction de plus haut niveau
- Performance
- nombre appels système réduits
- exemple : allocation mémoire malloc()/sbrk()

Sommaire

- 1 Plan du Cours
- 2 Introduction
- 3 Système d'exploitation
- 4 Principes des systèmes d'exploitation
- 5 Utilisation de la bibliothèque C standard et de l'API POSIX
- 6 Environnement, Terminaison, Commandes systèmes
- 7 Constantes POSIX, Bases de données
- 8 Les Entrées-Sorties
- 9 Système de fichiers
- 10 Processus
- 11 Bibliographie

Environnement des processus I

Environnement lié à la ligne de commande :

- Construction du binaire : édition des liens avec *.o et crt1.o
- Appel de start crt1.o → `main(int argc, char *argv[]);`

Variables d'environnement :

- L'environnement est accessible via : **extern char **environ;**
- Ou par la fonction POSIX :

#include <stdlib.h>

char *getenv(const char *nomDeVariable);

int putenv(const char *coupleNomValeur);

Environnement des processus II

- Accès aux variables d'environnement (*\$PATH*, *\$USER*...)
 - forme générale de `main()`
 - **int** `main (int argc, char *argv[], char **arge);` tableau, terminé par `NULL`, de chaînes de caractères de la forme `varname=value`
 - variable globale `environ`

Environnement des processus III

```
#include <stdio.h> #include <stdlib.h>
extern char **environ;
int main (int argc, char *argv[])
{
    char **envp = environ;
    while (*envp) printf("%s\n", *envp++);
    exit(EXIT_SUCCESS);
}
```

Environnement des processus IV

■ Fonction POSIX getenv()

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

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

```
#include <assert.h>
```

```
int main (int argc, char *argv[])
```

```
{
```

```
    char *username;
```

```
    username = getenv("USER");
```

```
    assert(username != NULL);
```

```
    printf("Hello %s\n", username);
```

```
    exit(EXIT_SUCCESS);
```

```
}
```

Interface avec le programme appelant

- Programme = nouvelle commande
 - appel depuis une autre commande
 - depuis un shell
 - *mprogram [arguments]...*
- Accès aux arguments de la commande

```
int main (int argc, char *argv[]);
    argc argv[0] argv[1] a argv[argc]
    nombre d'arguments +1 nom de la commande
    arguments
```

Terminaison d'un processus I

- Terminaison d'un processus :
 - de lui-même : `exit` ou `_exit` ou **`return`**
 - d'un signal ***kill*** `-X`
- Processus zombi : info accessible au père

Etapes de terminaison :

- Fermeture de tous les descripteurs ouverts.
- Libération de toutes les ressources allouées. .
- Envoi du signal `SIGHUP` à tous les processus du groupe si le processus est leader.
- Rattachement de tous les fils au processus de pid 1.
- Réveil du père attendant via `wait` ou `waitpid`.

Terminaison d'un processus II

- Un programme termine à la fin de `main()`
- Retourne une valeur à l'environnement
 - succès : `EXIT_SUCCESS`
 - échec : `EXIT_FAILURE`
 - fonction `exit()` de la bibliothèque C
 - qui fait appel à fonction POSIX `_exit()`
- Enregistrement de fonctions de terminaison par `atexit()`

Terminaison d'un processus III

```
void bye(void)
```

```
{printf("A la semaine prochaine!\n");}
```

```
int main (int argc, char *argv[]) {
```

```
    atexit(bye);
```

```
    printf("Hello...\n");
```

```
    exit(EXIT_SUCCESS);
```

```
}
```

Exécution de commandes shell

Commande system :

```
#include <stdlib.h>
int system(const char * commande);
```

Exemple d'appel système :

```
#include <stdlib.h>
int main(int argc, char ** argv) {
    int rep;

    rep = system(argv[1]);
    printf("Valeur renvoyee: %d",
           rep);
}
```


9 Bibliographie

Constantes de configuration POSIX

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

```
long sysconf(int name);
```

```
long pathconf(const char *reference, int symbole);
```

```
long fpathconf(int numeroDescripteur, int symbole);
```

■ **sysconf** : Retrouver la valeur associée au symbole POSIX.

Nom	Symbole	Explication
ARG_MAX	_SC_ARG_MAX	Long. Max des arguments passés à exec
CHILD_MAX	_SC_CHILD_MAX	Nb max de processus par utilisateur
CLK_TCK	_SC_CLK_TCK	Nb de ticks d'horloge par seconde
NGROUP_MAX	_SC_NGROUP_MAX	Nb max de groupes de processus par processus
OPEN_MAX	_SC_OPEN_MAX	Nb max de fichiers ouverts par processus
PASS_MAX	_SC_PASS_MAX	Nb Max de caractères dans un mot de passe
POSIX_VERSION	_SC_POSIX_VERSION	Indique la version POSIX supportée
NAME_MAX	_PC_NAME_MAX	Nb max de caractères dans un nom de fichiers
PATH_MAX	_PC_PATH_MAX	Nb max de caractères dans un nom de chemin relatif
PIPE_BUF	_PC_PIPE_BUF	Nb max de caractères écrits de façon atomique dans un tube

Bases de données systèmes

Accès à la base des comptes :

```
#include <pwd.h>
```

```
struct passwd *getpwuid(uid_t numero);
```

```
struct passwd *getpwnam(const char *nom);
```

```
struct passwd {  
    char *pw_name;  
    char *pw_passwd;  
    uid_t pw_uid;  
    gid_t pw_gid;  
    char pw_gecos;  
    char *pw_dir;  
    char *pw_shell; };
```


6 Les Entrées-Sorties

■ Généralités

- Manipulation des i-noeuds
- Primitives de base : répertoire
- Entrée/Sortie fichier
- Descripteurs
- Verrouillage
- Options d'ouverture
- Projection mémoire
- Interface POSIX : opérations avancées
- Bibliothèque C d'entrées/sorties

8 Prozess

Le système de fichiers

Sous UNIX tout est fichier !!!

Présentation des I/O sous forme de fichiers :

- Fichiers réguliers
- Répertoires
- Tubes (Redirection dans un tube)
- Fichiers spéciaux (*/proc/..*, */dev/null*)
- Bloc (*dd if=/dev/hda*)
- Liens symboliques (pas POSIX.1 :1988 mais POSIX.1 :2001)
- Sockets

Le système de fichiers

Sous UNIX tout est fichier !!!

Présentation des I/O sous forme de fichiers :

- Fichiers réguliers
- Répertoires
- Tubes (Redirection dans un tube)
- Fichiers spéciaux (*/proc/..*, */dev/null*)
- Bloc (*dd if=/dev/hda*)
- Liens symboliques (pas POSIX.1 :1988 mais POSIX.1 :2001)
- Sockets

Fichiers

■ Fichiers

- mémoriser des données
- sur disques (ou autres supports Â« externes Â»)
- de manière persistante
- Fichier = entité gérée par le système d'exploitation
- structuration, nommage, accès, protection, implantation...
- système de fichiers = partie du système d'exploitation
- Fichier = mécanisme d'abstraction
- présentation à l'utilisateur
- implantation des systèmes de fichiers

Répertoires

- Répertoire = fichier particulier
 - mémorise la structure du système de fichiers
 - opérations contrôlées par le système d'exploitation
- Fichier ordinaire
- contient les données $\hat{A} \ll \text{utilisateur} \hat{A} \gg$

Pluralité des systèmes de fichiers

- Différents types de systèmes de fichiers
 - à l'origine fourni par un système d'exploitation
 - exemple : MS-DOS, ufs (Unix), ext2/ext3 (Linux), NTFS (Windows NT), HFS (Mac OS)...
 - fournissent une même abstraction (à première vue...)
- Découplage système d'exploitation / système de fichiers
 - un système Linux peut « monter » un système de fichiers MS-DOS
- Système de fichiers = abstraction d'un disque
 - plusieurs disques
 - plusieurs systèmes de fichiers !
 - éventuellement de types différents !
 - vue unifiée des systèmes de fichiers présents = une unique hiérarchie

Sommaire

1 Plan du Cours

2 Introduction

3 Système d'exploitation

4 Principes des systèmes
d'exploitation

5 Utilisation de la bibliothèque C
standard et de l'API POSIX

6 Les Entrées-Sorties

■ Généralités

fichiers

- Manipulation des i-noeuds
- Primitives de base : répertoire
- Entrée/Sortie fichier
- Descripteurs
- Verrouillage
- Options d'ouverture
- Projection mémoire
- Interface POSIX : opérations avancées
- Bibliothèque C d'entrées/sorties

7 Système de fichiers

8 Processus

Hiérarchie

- Système de fichiers présente une hiérarchie
 - répertoire Â« contient Â» des fichiers
 - racine du système de fichiers
 - position courante dans la hiérarchie
- Système de fichiers n'est pas une hiérarchie
 - implantation sur la machine est un ensemble de nœuds
 - un nœud = un ensemble de blocs de données
 - détails d'implémentation cachés
- Le programmeur doit savoir que le système de fichiers n'est pas une hiérarchie
 - répertoire contient une liste de noms d'entrées
 - manipulation des liens symboliques
 - manipulation des liens physiques

- Le système de fichier est un arbre
 - vue simplificatrice (... sur laquelle on reviendra)
 - arbre = racine + nœuds à un parent unique + arcs
- Racine
 - notée /
 - est son propre parent
- Arcs ou entrées
 - nommés, tous caractères sauf '0' et '/'
 - éviter les espaces, les non imprimables, et non ASCII
- Nœuds non terminaux
 - répertoires toujours deux fils :
 - . et ..
 - . désigne le nœud lui-même, .. désigne son père
- Nœud terminaux
 - fichiers standard
 - contiennent des données

Numérotation des nœuds

- Désignation d'un fichier sur le support matériel
 - numéro de périphérique (device) numéro d'inœud (inode)
- Association d'une numérotation à un nœud
 - lien entre le nommage et le contenu
- Nommages multiples d'un nœud
 - de part les arcs . et ..
 - (entre autres... à suivre)
 - accès au même contenu
 - partage des modifications du contenu

Sommaire

1 Plan du Cours

2 Introduction

3 Système d'exploitation

4 Principes des systèmes
d'exploitation

5 Utilisation de la bibliothèque C
standard et de l'API POSIX

6 Les Entrées-Sorties
■ Généralités

fichiers

- Manipulation des i-noeuds
- Primitives de base : répertoire
- Entrée/Sortie fichier
- Descripteurs
- Verrouillage
- Options d'ouverture
- Projection mémoire
- Interface POSIX : opérations avancées
- Bibliothèque C d'entrées/sorties

7 Système de fichiers

8 Processus

Accès aux fichiers et répertoires

Primitives issues de **#include** <unistd.h>.

- Changement de répertoire : **int** chdir(**const char** *path);
- Consultation du répertoire courant :
char *getcwd(**char** *dirname, size_t size);
- size est la taille du tableau pointé par dirname (et qui a dû être alloué par l'appelant)

Accès aux informations du fichier :

- **int** stat(**const char** *path, **struct** stat *buf);

Manipulation des i-noeuds

Caractéristiques des fichiers :

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

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

```
struct stat {
```

```
    dev_t  st_dev; // id disque logique
```

```
    ino_t  st_ino; // numero du fichier sur ce disque
```

```
    mode_t st_mode; // type fichier et droits
```

```
    nlink_t st_nlink; // nombre de liens physique
```

```
    uid_t  st_uid; // proprietaire
```

```
    gid_t  st_gid; // groupe du proprietaire
```

```
    time_t st_atime; // dernier acces
```

```
    time_t st_mtime; // derniere modification fichier
```

```
    time_t st_ctime; // derniere modification i-noeud
```

```
};
```

Informations d'un fichier

■ Structure **struct** stat

- retournée par

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

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

```
int stat(const char *path, struct stat *sb);
```

```
int lstat(const char *path, struct stat *sb);
```

- stat() : fichier désigné par un lien symbolique
- lstat() : le lien lui-même
- Identification du nœud dans le système
 - partition et inœud

```
struct stat { dev_t st_dev; ino_t st_ino; ...
```

- nombre de liens physiques sur le fichier

```
struct stat {
```

```
...
```

```
nlink_t st_nlink;
```

Type et droits

- propriétaire, groupe propriétaire, mode

```
struct stat {
```

```
...
```

```
mode_t st_mode;
```

```
uid_t st_uid;
```

```
gid_t st_gid;
```

```
...
```

- informations propriétaire

```
#include <pwd.h>
```

```
struct passwd *getpwuid(uid_t uid);
```

```
struct passwd {
```

```
    char *pw_name; /* user name */
```

```
    uid_t pw_uid; /* user uid */
```

```
    gid_t pw_gid; /* user gid */
```

Manipulation des i-noeuds

Macros disponibles :

- `S_ISREG(mode)` vrai si l'i-noeud est celui d'un fichier normal (régulier).
- `S_ISLNK(mode)` vrai si l'i-noeud est celui d'un lien symbolique.
- `S_ISDIR(mode)` vrai si l'i-noeud est celui d'un répertoire.
- `S_ISCHR(mode)` vrai si l'i-noeud est celui d'un fichier spécial à accès par caractère.
- `S_ISBLK(mode)` vrai si l'i-noeud est celui d'un fichier spécial à accès par bloc.
- `S_ISFIFO(mode)` vrai si l'i-noeud est celui d'un tube nommé ou anonyme.

Manipulation des i-noeuds

Macros disponibles :

- Sélection des champs : S_IRWX[UGO] S_IS[UG]ID
- Décodage des droits d'accès : S_I[RWX](USR|GRP|OTHR)

```
struct stat sbuf;
```

```
char *path = "foo/bar";
```

```
if (stat(path, &sbuf) >= 0) {
```

```
    int m = sbuf.st_mode;
```

```
    if (S_ISREG(m)) { /* le fichier est un fichier ordinaire */
```

```
        if (m & (S_IWUSR | S_IWGRP)) {
```

```
            /* fichier lisible par le propriétaire ou son groupe */
```

```
        } } }
```

Manipulation des i-noeuds : exemple I

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

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

```
#include <time.h>
```

```
#include <pwd.h>
```

```
#include <grp.h>
```

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

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

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

```
void print_stat(const char *ref, struct stat *statut) {
```

```
    struct passwd *pw;
```

```
    struct group *gr;
```

```
    char type;
```

Manipulation des i-noeuds : exemple II

```
#define LTEMPPS 32
char pws[9], grs[9], temps[LTEMPPS];

type = '?';
if (S_ISREG(statut->st_mode)) type = '-';
else if (S_ISDIR(statut->st_mode)) type = 'd';
else if (S_ISCHR(statut->st_mode)) type = 'c';
else if (S_ISBLK(statut->st_mode)) type = 'b';
else if (S_ISFIFO(statut->st_mode)) type = 'p';
strftime(temps,LTEMPPS,"%a %e %h %Y %H:%M:%S",
        localtime(&(statut->st_mtime)));

pw = getpwuid(statut->st_uid);
if (pw!=NULL) strcpy(pws,pw->pw_name);
```

Manipulation des i-noeuds : exemple III

```
else sprintf(pws,"%8d",(int)statut->st_uid);
```

```
gr = getgrgid(statut->st_gid);
```

```
if (gr!=NULL) strcpy(grs,gr->gr_name);
```

```
else sprintf(grs,"%8d",(int)statut->st_gid);
```

```
printf("%c%c%c%c%c%c%c%c%c%c%c %2d %8s %8s %9d %  
s
```

```
    %s\n", type,  
    statut->st_mode&S_IRUSR?'r':'-',  
    statut->st_mode&S_IWUSR?'w':'-',  
    statut->st_mode&S_IXUSR?'x':'-',  
    statut->st_mode&S_IRGRP?'r':'-',  
    statut->st_mode&S_IWGRP?'w':'-',  
    statut->st_mode&S_IXGRP?'x':'-',
```


Manipulation des i-noeuds : exemple IV

```
    statut->st_mode&S_IROTH?'r':'-',  
    statut->st_mode&S_IWOTH?'w':'-',  
    statut->st_mode&S_IXOTH?'x':'-',  
    (int)statut->st_nlink, pws,grs, (int)statut->st_size,  
    temps,ref );  
}
```

```
int main(int argc,const char *argv[]) {  
    struct stat statut;  
    int i;  
    if (argc<2) {  
        if ( fstat(STDIN_FILENO,&statut) == -1 ) {  
            fprintf(stderr,"%s: impossible d'obtenir le statut de  
                %s\n",argv[0], "<STDIN>");
```

Manipulation des i-noeuds : exemple V

```
        exit(EXIT_FAILURE);
    }
    print_stat("<STDIN>", &statut);
}
else {
    for (i=1; i<argc; i++) {
        if ( stat(argv[i], &statut) == -1 ) {
            fprintf(stderr, "%s: impossible d'obtenir le statut de %s\n",
                    argv[0],
                    argv[i]);
            continue; }
        print_stat(argv[i], &statut); }
    }
    exit(EXIT_SUCCESS);
```

Manipulation des i-noeuds : exemple VI

}

Manipulation des i-noeuds

- Droits d'accès suffisants :
int access(**const char** *reference, **int** mode_acces);
- Modification des droits d'accès :
int chmod(**const char** *reference, mode_t mode);,
int fchmod(**int** descripteur, mode_t mode);
- Changement de propriétaire :
int chown(**const char** *reference, uid_t uid, gid_t gid);,
int fchown(**int** descripteur, uid_t uid, gid_t gid);
- **int** utime(**const char** *ref, **const struct** utimbuf *buf);
- **int** truncate(**const char** *reference, off_t longueur);,
int ftruncate (**int** descripteur, off_t longueur);

Création des i-noeuds

Création des fichiers “normaux” :

- fichier régulier : create ou open avec O_CREATE
- répertoire : mkdir
- tube : pipe ou mkfifo
- lien symbolique : symlink

Fichiers “spéciaux” :

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

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

```
int mknod(const char *reference, mode_t droits, dev_t  
    ressource);
```

Sommaire

1 Plan du Cours

2 Introduction

3 Système d'exploitation

4 Principes des systèmes
d'exploitation

5 Utilisation de la bibliothèque C
standard et de l'API POSIX

6 Les Entrées-Sorties
■ Généralités

fichiers

■ Manipulation des i-noeuds

■ **Primitives de base : répertoire**

■ Entrée/Sortie fichier

■ Descripteurs

■ Verrouillage

■ Options d'ouverture

■ Projection mémoire

■ Interface POSIX : opérations
avancées

■ Bibliothèque C
d'entrées/sorties

7 Système de fichiers

8 Processus

Droits d'accès I

■ Vérification du droit d'accès à un fichier

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

```
int access(const char *path, int amode);
```

- droit défini par une combinaison $\hat{\ll}$ ou $\hat{\gg}$ des macros R_OK, W_OK, X_OK, et F_OK
- exemple :

```
if (access(pathname, X_OK))  
    printf("Fichier executable\n");
```

Droits d'accès II

■ Positionner les droits d'un fichier

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

```
int chmod(const char *path, mode_t mode);
```

- spécification de mode sous forme d'un masque binaire (S_IRUSR, S_IWUSR, S_IXUSR, etc.)

Parcours des répertoires I

- Contenu d'un répertoire = liste de liens
 - itération de la liste
- Descripteur de répertoire
 - ouverture et fermeture

```
#include <dirent.h>
```

```
DIR *opendir(const char *dirname);
```

```
int closedir(DIR *dirp);
```

Parcours des répertoires II

- Itération sur les entrées du répertoire
 - obtenir les informations d'une entrée :

```
#include <dirent.h>
```

```
struct dirent *readdir(DIR *dirp);
```

- numéro d'inœud et nom :

```
struct dirent {  
    ino_t d_ino;  
    char d_name[];  
    ...  
};
```

Parcours des répertoires III

Listing 1 – Exemple : recherche dans le repertoire courant

```
static int lookup(const char *name) {  
    DIR *dirp; struct dirent *dp;  
    if ((dirp = opendir(".")) == NULL) {  
        perror("couldn't open '.'); return 0;  
    }  
    while ((dp = readdir(dirp))) {  
        if (! strcmp(dp->d_name, name)) {  
            printf("found %s\n", name); closedir(dirp); return 1;  
        }  
    }  
    if (errno != 0) perror("error reading directory");  
    else printf("failed to find %s\n", name); closedir(dirp);
```

Parcours des répertoires IV

```
    return 0;  
}
```

Création d'un répertoire I

- Créer une entrée dans un répertoire
 - répertoire existant
 - sinon le créer préalablement
 - itération nécessite le droit d'écriture sur le répertoire
- Masque des droits
- variable système umask du processus modifiée par

#include <sys/stat.h>

mode_t umask(mode_t cmask);

- qui retourne l'ancien masque
- paramètre mode d'une primitive de création
- droits de l'entrée créée = mode & ~umask

Création d'un répertoire II

■ Créer un répertoire

■ primitive

```
#include <sys/stat.h>  
int mkdir(const char *path, mode_t mode);
```

■ exemple

```
int status;  
status = mkdir("/tmp/dir", S_IRWXU | S_IRWXG | S_IROTH |  
                S_IXOTH);
```

■ résultat

Création d'un répertoire III

```
% mkdir  
% ls -ld /tmp/dir drwxr-xr-x 1 phm phm 4 29 Dec 00:32 /tmp/dir  
% umask  
022  
% umask 002  
% rmdir /tmp/dir  
% mkdir  
% ls -ld /tmp/dir drwxrwxr-x 1 phm phm 4 29 Dec 00:34 /tmp/dir
```

Destruction

- Destruction d'un nœud
 - supprimer une entrée dans la hiérarchie
 - supprimer l'inœud si dernière entrée
 - implémentation : compteur de références sur un inœud
- Détruire un répertoire

```
#include <unistd.h>  
int rmdir(const char *path);
```

- doit être vide
- Détruire un fichier

```
#include <unistd.h>  
int unlink(const char *path);
```

- fichier ordinaire, ou lien symbolique...

127/310

Ouverture d'un fichier

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

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

```
#include <fcntl.h>
```

```
int open(const char *reference, int mode, ... /* mode_t droits */  
);
```

Fonctionnement :

- Retour : numéro du descripteur alloué.
- Mode : construit par disjonction (—) O_RDONLY, O_WRONLY ou O_RDWR
- En cas de succès :
 - la position courante est à 0
 - le compteur d'ouvertures de l'i-node incrémenté
 - le compteur de la table des fichiers ouverts initialisé à un
 - un descripteur reste ouvert

Ouverture d'un fichier : exemple

```
int d1, d2, d3;
```

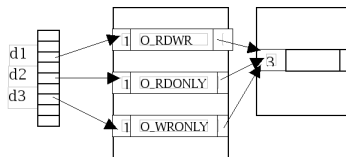
```
char * ref;
```

```
...
```

```
d1=open(ref, O_RDWR);
```

```
d2=open(ref, O_RDONLY);
```

```
d3=open(ref, O_WRONLY);
```



Ouverture d'un fichier : autres options

■ Options d'accès

- aucun, un ou plusieurs parmi les suivants
- `O_APPEND` : mode ajout
- `O_CREAT` création du fichier s'il n'existe pas
 - 3e paramètre de type `mode_t`
- `O_EXCL` et `O_CREAT` : échec si le fichier existe déjà
- `O_TRUNC` (et `O_WRONLY` ou `O_RDWR`) si le fichier existe, le tronquer à zéro
- `O_NONBLOCK`, `O_DSYNC`, `O_RSYNC`, `O_SYNC` à suivre...

Ouvertures typiques I

- lecture d'un fichier (que l'on veut) existant

```
int fd = open(path, O_RDONLY);  
if (fd == -1) {  
    perror("ouverture du fichier");  
}
```

- création d'un fichier (et troncature si existant)

```
open(path, O_WRONLY | O_CREAT | O_TRUNC,  
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH |  
S_IWOTH);  
  
ou  
open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
```

Ouvertures typiques II

- pour un fichier exécutable

```
open(path, O_WRONLY | O_CREAT | O_TRUNC,  
      S_IRWXU | S_IRWXG | S_IRWXO);
```

ou

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, 0777);
```

- ajout à un fichier existant

```
open(path, O_WRONLY | O_APPEND | O_CREAT,  
      S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |  
      S_IROTH | S_IWOTH);
```

Ouverture à la création du processus

- Ouverture à la création du processus
 - réalisée automatiquement par le système
 - entrées/sorties standard
 - descripteur STDIN_FILENO (0) : entrée standard
 - descripteur STDOUT_FILENO (1) : sortie standard
 - descripteur STDERR_FILENO (2) : sortie d'erreur

Ouverture d'un fichier : propriétés

Contrôle de protection effectué par le noyau :

- Création, suppression, modification
- `st_uid` et `st_gid` du propriétaire
- identité du processus

Propriété des nouveaux fichiers ou répertoires :

- propriétaire du fichier : propriétaire du processus
- groupe propriétaire du fichier (implémentation ?) :
 - le groupe effectif du processus
 - le groupe du répertoire conteneur

Fermeture d'un fichier

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

```
int close(int numeroDescripteur);
```

Ferme les descripteurs

- lève les verrous
- décrémente les compteurs de descripteurs

Lecture dans un fichier

Lire tailleBuff via un descripteur :

```
ssize_t read(int numeroDes, void *buffer, size_t tailleBuff);
```

Si aucun verrou :

- si offset j taille fichier : lecture position courante
- sinon, renvoie 0 (et remplit le buffer)

En cas de verrou, le processus est bloqué.

Ecriture dans un fichier

Ecriture de tailleBuff caractères via un descripteur :

```
ssize_t write(int numeroDes, void *buffer, size_t tailleBuff);
```

- Position courante mise à jour (éventuellement fin de fichier)
- Si verrou, attente
- -1 pour un erreur (descripteur inexistant, descripteur non ouvert en écriture)

Exemple : copie de fichier I

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

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

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

```
#include <fcntl.h>
```

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

```
#define BUFSIZE 4096
```

```
static void copy_file(const char *src, const char *dst)  
{
```

```
    int fdsrc, fddst;
```

```
    char buffer[BUFSIZE];
```

```
    int nchar;
```

Exemple : copie de fichier II

```
fdsrc = open(src, O_RDONLY);
fddst = open(dst, O_WRONLY | O_CREAT | O_TRUNC,
             S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH |
             S_IWOTH);
while ((nchar = read(fdsrc, buffer, BUFSIZE))) {
    write(fddst, buffer, nchar);
}
close(fdsrc); close(fddst);
}
```

Modification de l'offset

Modification de la position courante :

`off_t lseek(int numeroDes, off_t déplacement, int origine);`

- `SEEK_SEEK` : début de fichier
- `SEEK_CUR` : courant
- `SEEK_END` : fin de fichier

Exemple d'utilisation de l'offset l

```
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
typedef struct {
    int integer;
    char string[24];
} RECORD;
#define NRECORDS (100)
int main()
{
    RECORD record, *mapped;
    int i, f;
```

Exemple d'utilisation de l'offset II

```
FILE *fp;
fp = fopen("records.dat","w+");
for(i=0; i<NRECORDS; i++) {
    record.integer = i;
    sprintf(record.string,"RECORD-%d",i);
    fwrite(&record,sizeof(record),1,fp);
}
fclose(fp);
```

// ouverture du flux vers le fichier "records.dat"

```
fp = fopen("records.dat","r+");
```

// déplacement de l'offset au 43eme enregistrement

```
fseek(fp,43*sizeof(record),SEEK_SET);
```

// lecture du 43eme enregistrement

Exemple d'utilisation de l'offset III

```
fread(&record, sizeof(record), 1, fp);  
/* mise a jour des donnees de cet enregistrement dans la  
   structure "record" */  
record.integer = 143; sprintf (record.string, "RECORD-%d",  
    record.integer);  
/* ecriture de cette structure "record" maj dans le fichier */  
fseek(fp, 43*sizeof(record), SEEK_SET);  
fwrite(&record, sizeof(record), 1, fp);  
fclose(fp);  
}
```

144/310

Duplication d'un descripteur

Duplique avec un descripteur synonyme :

```
int dup(int numeroDes);
```

Force un descripteur à être synonyme :

```
int dup2(int premier, int second);
```

Exemple :

```
close(STDIN_FILENO); dup(entree); close (entree);  
close(STDOUT_FILENO); dup(sortie); close (sortie);
```

Controle d'un descripteur

int fcntl(**int** numeroDes, **int** commande, ...);

Lire/Ecrire des attributs d'un descripteur :

- F_GETFD : récupérer les attributs dans l'entier retourné
- F_SETFD : modifier les attributs avec le troisième paramètre
- F_GETFL : lire le mode d'ouverture dans la table des fichiers ouverts
- F_SETFL : écrire le mode d'ouverture dans la table des fichiers ouverts
- F_DUPFD : obtenir un numéro synonyme supérieur ou égal au troisième paramètre

2 Introduction

4 Principes des systèmes d'exploitation

6 Les Entrées-Sorties

■ Généralités

- Manipulation des i-noeuds
- Primitives de base : répertoire
- Entrée/Sortie fichier
- Descripteurs
- **Verrouillage**
- Options d'ouverture
- Projection mémoire
- Interface POSIX : opérations avancées
- Bibliothèque C d'entrées/sorties

8 Prozess

Verrouillage

- Nécessité de contrôler les accès concurrents aux données
 - cohérence des données
 - écritures multiples en fin de fichier : cohérence assurée par le système si O_APPEND
 - écritures multiples quelconques ?
 - écriture et lecture simultanées ?
 - lectures multiples simultanées possibles
- Verrou (lock)
 - mécanisme général de contrôle d'accès
 - cas particulier des accès aux données d'un fichier
 - notion de propriétaire d'un verrou
 - opérations autorisées au seul propriétaire du verrou
 - prise et relâche d'un verrou
 - devient propriétaire
 - portée d'un verrou
 - ensemble des positions d'un fichier contrôlées par un verrou

Verrouillage

- Nécessité de contrôler les accès concurrents aux données
 - cohérence des données
 - écritures multiples en fin de fichier : cohérence assurée par le système si O_APPEND
 - écritures multiples quelconques ?
 - écriture et lecture simultanées ?
 - lectures multiples simultanées possibles
- Verrou (lock)
 - mécanisme général de contrôle d'accès
 - cas particulier des accès aux données d'un fichier
 - notion de propriétaire d'un verrou
 - opérations autorisées au seul propriétaire du verrou
 - prise et relâche d'un verrou
 - devient propriétaire
 - portée d'un verrou
 - ensemble des positions d'un fichier contrôlées par un verrou

Verrouillage des fichiers réguliers

- Contrôle de la concurrence d'accès aux fichiers
- Verrou attaché à l'i-node
- Verrous :
 - partagés (shared lock)
 - exclusif (exclusive lock)

Verrouillage

- Accès exclusif en écriture lors de la création d'un fichier
 - options `O_CREATE` | `O_EXCL` de `open()`
 - restrictif : création exclusive et non accès exclusif
 - granularité fixe : le fichier
- Portées des verrous POSIX
 - ensemble de positions d'un fichier
 - intervalle : $[offset_i..offset_f]$
 - intervalle infini : $[offset_i..\infty]$
- Deux types de verrous POSIX
 - verrous partagés ou verrous de lecture
 - verrous exclusifs ou verrous d'écriture

- Deux comportements possibles vis-à-vis des verrous
 - mode consultatif (ou coopératif)
 - les opérations read()/write() sont toujours possibles
 - la pose d'un verrou empêche la pose de verrous incompatibles
 - le programme doit, de lui-même, consulter les verrous
 - mode impératif
 - le système contrôle les accès via read()/write() en fonction des verrous posés
 - verrou partagé : interdit les accès en écriture
 - verrou exclusif : interdit tout accès
- pas de spécification POSIX du choix du mode (!)
 - habituellement mode coopératif

Verrouillage des fichiers réguliers

- Type **struct** flock de description d'un verrou

```
int fcntl(int nDes, int cmd, /* struct flock *flockPtr */);
```

```
struct flock {
    short l_type; /* F_RDLCK, F_WRLCK or F_UNLCK */
    short whence; /* SEEK_SET, SEEK_CUR or
                  SEEK_END */
    off_t l_start; /* offset in bytes relative to l_whence */
    off_t l_len; /* length in bytes; 0 means lock to EOF */
    pid_t l_pid; /* process owner returned with F_GETLK */
}
```

- verrou partagé (F_RDLCK), verrou exclusif (F_WRLCK), ou déverrouillage (F_UNLCK)
- whence parmi SEEK_CUR, SEEK_SET, et SEEK_END

Verrouillage

■ Opérations de verrouillage

■ appel système

#include <fcntl.h>

int fcntl(**int** fildes, **int** cmd, **struct** flock *plock);

- cmd parmi F_SETLK, F_SETLKW, F_GETLK
- F_SETLK : demande non bloquante de prise du verrou
 - erreur -1 en cas de verrou incompatible : EACCESS ou EAGAIN
 - erreur -1 en cas d'interblocage (deadlock) : EDEADLK
- F_SETLKW : demande bloquante de prise de verrou
 - attente que le verrou puisse être posé
 - erreur -1 en cas d'interblocage (deadlock) : EDEADLK
- F_GETLK : teste d'existence d'un verrou incompatible avec le verrou donné
 - verrou incompatible retourné dans plock

Verrouillage des fichiers réguliers

- Pour obtenir un verrou en lecture, le descripteur doit être ouvert en lecture.
- Pour un verrou en écriture, il doit l'être en écriture.

	Read lock	Write lock
Aucun verrou sur la région	OK	OK
Un ou +ieurs verrous en lecture sur région	OK	refus
Un verrou en écriture sur région	refus	refus

1 Plan du Cours

2 Introduction

3 Système d'exploitation

4 Principes des systèmes d'exploitation

5 Utilisation de la bibliothèque C standard et de l'API POSIX

6 Les Entrées-Sorties

■ Généralités

- Manipulation des i-noeuds
- Primitives de base : répertoire
- Entrée/Sortie fichier
- Descripteurs
- Verrouillage
- **Options d'ouverture**
- Projection mémoire
- Interface POSIX : opérations avancées
- Bibliothèque C d'entrées/sorties

7 Système de fichiers

8 Prozess

Options d'ouverture I

- Différents modes de lecture/écriture
 - mode bloquant / non bloquant ; bloquant par défaut
 - mode synchronisé / non synchronisé ; non synchronisé par défaut
- Spécification du mode à l'ouverture du fichier (open())
 - paramètre oflag : mode d'accès + options
- Spécification du mode postérieur à l'ouverture
 - fonction fcntl(), commandes F_GETFL/F_SETFL (get/set flag)
 - exemple : ajout d'un mode

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_FLAG);
```

Options d'ouverture II

■ Mode bloquant / non bloquant

- par défaut un appel `read()` est bloquant
 - lecture depuis un terminal, depuis un tube...
- un appel `write()` peut être bloquant
 - à cause d'un verrou impératif...
- option `O_NONBLOCK` du mode d'ouverture : mode non bloquant

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);
```

- les appels à `read()/write()` ne sont jamais bloquants
- situation de blocage : `read()/write()` retourne une erreur (-1) ; positionne `errno` (EAGAIN)

Options d'ouverture III

■ Mode synchronisé / non synchronisé

- par défaut les écritures se font en mode non synchronisé
- les données à écrire sont mémorisées dans des caches du système d'exploitation
- de manière asynchrone, le système réalise les écritures des caches sur les disques
- évite aux processus d'attendre les écritures disques
- pas de garantie que l'écriture disque ait été réalisée...
- option O_SYNC du mode d'ouverture : écritures en mode synchronisé
- appel à write retourne quand les données sont écrites sur le disque

1 Plan du Cours

2 Introduction

3 Système d'exploitation

4 Principes des systèmes d'exploitation

5 Utilisation de la bibliothèque C standard et de l'API POSIX

6 Les Entrées-Sorties

■ Généralités

- Manipulation des i-noeuds
- Primitives de base : répertoire
- Entrée/Sortie fichier
- Descripteurs
- Verrouillage
- Options d'ouverture
- **Projection mémoire**
- Interface POSIX : opérations avancées
- Bibliothèque C d'entrées/sorties

7 Système de fichiers

8 Prozess

Projection mémoire I

- Charger un fichier dans l'espace d'adressage du processus
 - ensemble ou partie du fichier manipulation du fichier via l'adressage mémoire
- Evite de multiples copies vers/depuis les caches systèmes
 - lors de multiples écritures/lectures successives de mêmes positions
- Implantation
 - le fichier n'est pas lu dans son intégralité lors de la projection !
- Appel système `mmap()`

Projection mémoire II

■ Appel système mmap()

#include <sys/mman.h>

void *mmap(**void** *addr, size_t len, **int** prot, **int** flags, **int** fildes, off_t off);

- considère les données [off..off+len[du fichier fildes
- addr est un adresse de projection choisie ; la positionner à NULL
- prot définit les accès possibles à la zone mémoire
 - PROT_NONE, aucun accès, ou
 - combinaison binaire de PROT_READ, PROT_WRITE, PROT_EXEC
- flags précise des options
 - répercutions des modifications sur toutes les projections du fichier ou non

Projection mémoire III

- retourne l'adresse de la projection

- Appel

```
int munmap(void *addr, size_t len);
```

- libère la zone mémoire de projection

Projection mémoire IV

```
static void copy_file(const char *src, const char *dst) {  
    struct stat stsrc, stdst;  
    int fdsrc, fddst;  
    char *psrc; int size;  
  
    lstat(src, &stsrc);  
    lstat(dst, &stdst);  
  
    if (stsrc.st_ino == stdst.st_ino && stsrc.st_dev == stdst.st_dev  
        ) {  
        fprintf(stderr, "%s et %s sont le meme fichier\n", src, dst  
            );  
        return;  
    }  
}
```

Projection mémoire V

```
fdsrc = open(src, O_RDONLY);  
fddst = open(dst, O_WRONLY | O_CREAT | O_TRUNC,  
             S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |  
             S_IROTH | S_IWOTH);  
size = stsrc.st_size;  
psrc = mmap(NULL, size, PROT_READ, MAP_PRIVATE,  
             fdsrc, 0);  
write(fddst, psrc, size);  
munmap(psrc, size);  
close(fdsrc);  
close(fddst);  
}
```

Sommaire

1 Plan du Cours

2 Introduction

3 Système d'exploitation

4 Principes des systèmes d'exploitation

5 Utilisation de la bibliothèque C standard et de l'API POSIX

6 Les Entrées-Sorties

■ Généralités

fichiers

- Manipulation des i-noeuds
- Primitives de base : répertoire
- Entrée/Sortie fichier
- Descripteurs
- Verrouillage
- Options d'ouverture
- Projection mémoire
- Interface POSIX : opérations avancées
- Bibliothèque C d'entrées/sorties

7 Système de fichiers

8 Processus

Lecture dans un fichier : POSIX.1

Distribution segmentée dans un vecteur :

```
ssize_t readv(int num, const struct iovec *vecteur, int n);
```

Structure iovec :

```
struct iovec {
    char *iov_base; /* adresse du buffer */
    size_t iov_len; /* taille du buffer = nb de caracteres a lire
                      */ }
```

Lecture à un offset donné :

```
ssize_t pread(int numDesc, void *buff, size_t sizeBuff, off_t
    position);
```

Positionnement I

■ Déplacer la position courante

- écritures/lectures se font à la position courante
- écritures/lectures modifient la position courante
- modification “explicite” de cette position courante

#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);

- déplacement de offset octets
- offset peut être négatif
- suivant whence, à partir de
 - la position courante SEEK_CUR
 - le début de fichier (a/c 0) SEEK_SET
 - la fin de fichier SEEK_END
- retourne la nouvelle position absolue

■ Positionnement parfois impossible / interdit

Positionnement II

- fichiers/périphériques sans capacité de positionnement
 - tubes, terminaux...
- retourne erreur (-1) ; positionne errno (ESPIPE)
- Positionnement possible au delà de la fin de fichier
 - ne change pas la taille
 - écriture à cette position changera la taille
 - trou laissé entre l'ancienne fin de fichier et cette position : plein de zéro fichier "creux"

Positionnement : Exemple

```
int main (int argc, char *argv[]) {  
    int status;  
    status = lseek(STDIN_FILENO, 0, SEEK_SET);  
    if (status == -1) {  
        fprintf(stderr, "Non seekable file\n");  
        exit(EXIT_FAILURE);  
    }  
    fprintf(stderr, "Seekable file \n");  
    exit(EXIT_SUCCESS);  
}
```

% ./skable Seekable file

% ./skable < /etc/passwd Seekable file

% cat /etc/passwd | ./skable Non seekable file

Écriture indexée

```
ssize_t pwrite(int fildes, const void *buf, size_t nbyte, off_t  
              offset);
```

- écriture à une position donnée
- ne modifie pas la position courante
- paramètre offset supplémentaire : position absolue / début du fichier

écriture indexée

```
#define SIZEOF_PNGINT 4
```

```
static int png_write_sizes(int fd, png_int x, png_int y)
{
    ssize_t lx, ly;
    lx = pwrite(fd, &x, SIZEOF_PNGINT, 16);
    ly = pwrite(fd, &y, SIZEOF_PNGINT, 20);
    return (lx == SIZEOF_PNGINT) && (ly ==
        SIZEOF_PNGINT);
}
```

```
% cat /usr/share/file/magic
```

```
[...]
```

```
# PNG [Portable Network Graphics, or "PNG's Not GIF"] images
```

```
# 137 P N G \r \n ^Z \n [4-byte length] H E A D [HEAD data] [HEAD crc] ...
```

```
0 string \x89PNG PNG image data,
```

```
>4 belong !0x0d0a1a0a CORRUPTED.
```


Pourquoi une bibliothèque

■ Abstraction de plus haut niveau

- entrées/sorties formatées
- `fprintf()`, `fscanf()`, etc.
- *man 3*

■ Performances

- entrées/sorties temporisées
- réduire le nombre des coûteux appels système
- coût d'un appel système ≈ 1000 instructions écriture (/lecture) dans un tampon utilisateur
- quand le tampon est plein (/vide), appel système `write()(/read())`

Exemple : copie de fichier I

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

```
static void copy_file(const char *src, const char *dst)  
{  
    struct stat stsrc, stdst;  
    FILE *fsrc, *fdst;  
    int c;  
  
    lstat(src, &stsrc);  
    lstat(dst, &stdst);
```

Exemple : copie de fichier II

```
if (stsrc.st_ino == stdst.st_ino && stsrc.st_dev == stdst.st_dev) {  
    fprintf(stderr, "%s et %s sont le meme fichier\n", src, dst)  
    ;  
    return;  
}  
fsrc = fopen(src, "r");  
fdst = fopen(dst, "w");  
while ((c = fgetc(fsrc)) != EOF)  
    fputc(c, fdst);  
fclose(fsrc);  
fclose(fdst);  
}
```

Attention une bibliothèque

- Ne pas mixer les fonctions bibliothèques et appels système
 - la bibliothèque utilise des appels système !
- Bibliothèque temporisée
 - attente que le tampon soit plein (/vide)
- Possible surcoût ?
 - copie supplémentaire
 - regagné par la factorisation des appels systèmes

- ## 6 Les Entrées-Sorties

- 1 Plan du Cours
- 2 Introduction
- 3 Système d'exploitation
- 4 Principes des systèmes d'exploitation
- 5 Utilisation de la bibliothèque C standard et de l'API POSIX

7 Système de fichiers

- Périphériques
- Virtualisation
- Elements d'implantation
- Méthodes d'Allocation

9 Bibliographie

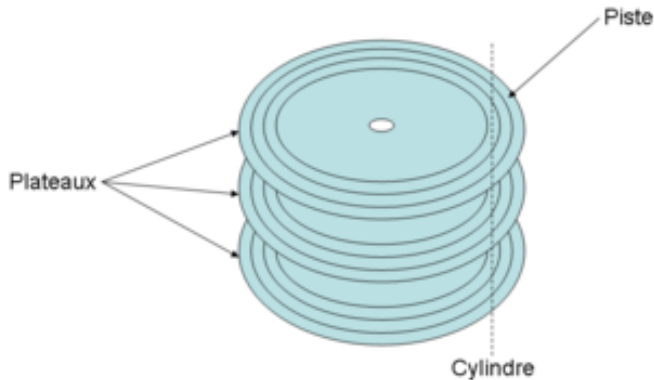
Périphériques

Mémoire persistante

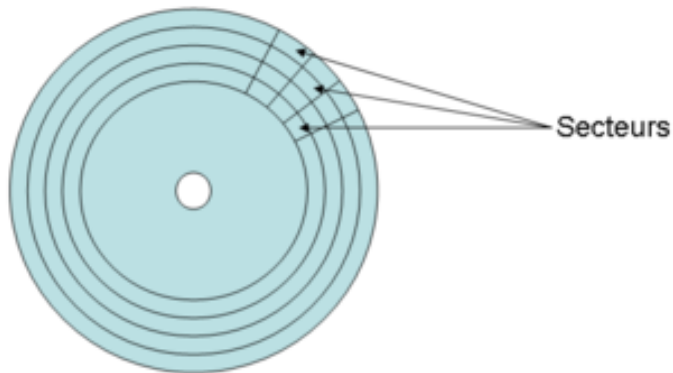
Mémoire dont le contenu n'est pas perdu après un arrêt du système informatique.

- Objectifs premiers :
 - Stocker les informations utiles au démarrage d'un système
 - Stocker des données Â« long terme Â»
 - Sauvegarder des données
- Type de supports physiques :
 - Disque dur, disques amovibles, ...
 - SSD : solid-state drive
 - Disquettes, Disques opto-magnétique, ...
 - bandes magnétiques, ...
 - EEPROM, FlashRam, Fe-RAM, M-RAM, .

Les disques magnétiques (Disques dur) I



Les disques magnétiques (Disques dur) II



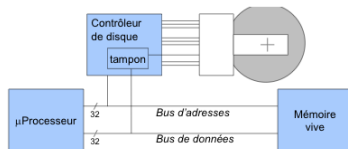
Les disques magnétiques (Disques dur) III

- plateaux, cylindres, secteurs
- 4000 t/mn à 10000 t/mn
- 512 à 16384 octets stocker sur chaque secteur
- 20 à 200 secteurs par cylindre
- 800 à 4000 cylindres (jusqu'à 12 ms de déplacement)
- 1 à 16 plateaux
- Adressage par interface $\{p,c,s\}$
- Capacité standard en 2002 :
 - 40Go,
 - 7200 tours/minute,
 - Déplacement 7 ms,
 - Ultra DMA 100 Mhz \Rightarrow de 20Mo/s à 40Mo/s

Accès aux disques

■ Interface d'accès aux disques :

- Contrôleur de disque IDE : interface normées ATA, ATA-2, ...
- Contrôleur de disque SCSI / SCSI-2 ;



■ Accès au données persistantes :

- Accès direct au travers d'un tampon de lecture/écriture
- DMA (U-DMA) pour copie des données dans/depus la mémoire centrale

Exemple interface logicielles d'accès au disque I

```
void readSectors(  
    int device, // num disques sur le controleur  
    int head, // num du plateau  
    int cylender, // num du cylindre de depart  
    int sector, // num du secteur de depart  
    int nsectors, // nb de secteurs a lire  
    unsigned char *buffer); // adresse destination de lecture  
void writeSectors(  
    int device, // plusieurs disques sur un meme controleur  
    int head, // num du plateau  
    int cylender, // num du cylindre de depart  
    int sector, // num du secteur de depart  
    int nsectors, // nombre de secteurs a lire
```

Exemple interface logicielles d'accès au disque II

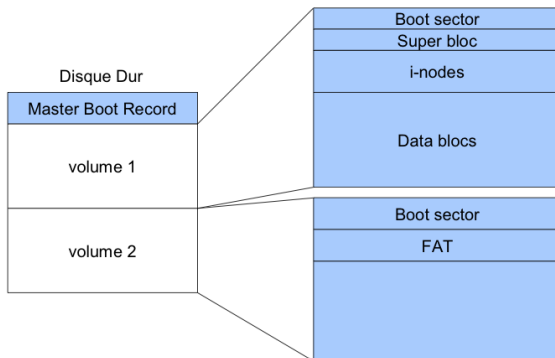
```
unsigned char *buffer); // adresse des donnees a ecrire  
void frSectors( int device, int p, int c, int s, int nsectors);
```

Exemple interface logicielles d'accès au disque III

Dans la pratique, l'interface d'accès au matériel ne saurait se résumer à un appel de fonction. On programme un contrôleur de disque via des registres matériels.

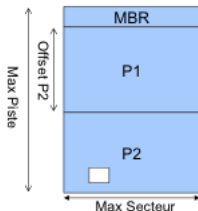
- En réalité : Virtualisation des accès
 - Simplifier/homogénéiser les systèmes d'adressage des disques :
 - $\langle \text{numplateau}, \text{numcylindre}, \text{numsecteur} \rangle$ devient \Rightarrow
 $\langle \text{identifiantdevolume}, \text{numbloc} \rangle$
 - Multiplexer l'usage d'un disque unique en construisant plusieurs parties dédiées à des usages distincts (partitions ou volumes)
 - Gérer les systèmes d'amorce et les systèmes d'amorce multiples

Définir un découpage



Conversion

A un bloc correspond un secteur sur une piste du disque. Aussi les opérations de lecture/écriture sont équivalents. Cependant, le système doit convertir un numéro de bloc en num de secteur/num de piste.



$$\blacksquare \text{ nBloc} = \text{Offset P2} + \text{Secteur} + \text{Piste} \times \text{Max Secteur}$$

Sommaire

1 Plan du Cours

2 Introduction

3 Système d'exploitation

4 Principes des systèmes
d'exploitation

5 Utilisation de la bibliothèque C
standard et de l'API POSIX

6 Les Entrées-Sorties

7 Système de fichiers

■ Périphériques

■ Virtualisation

■ Elements d'implantation

■ Méthodes d'Allocation

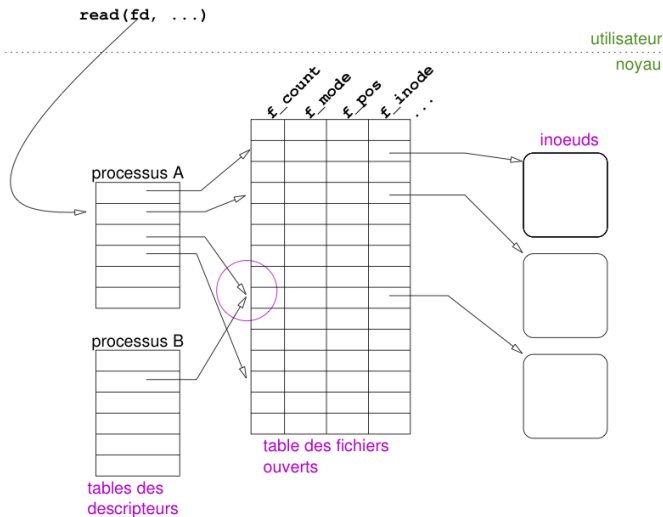
8 Processus

9 Bibliographie

Tables des descripteurs / table des fichiers ouverts I

- Descripteur de fichier
- index dans la table des descripteurs du processus
- une table des descripteurs par processus
- Table des fichiers ouverts
 - une table unique pour tous les processus
 - entrées partagées par tous les processus
 - f_count nombre de références
 - f_mode mode ouverture (lecture/écriture...)
 - f_pos position courante
 - f_inode inœud

Tables des descripteurs / table des fichiers ouverts II



Structure de données inœud

- Inœud = données associées à un fichier
 - propriétaire, droits, date, taille fichier...
 - identification des blocs disques
- Inœud = informations pérennes
 - les inœuds sont stockés sur les disques !
 - copies en mémoire (cache)
- Organisation des données sur le disque
 - données regroupées en blocs
 - bloc \approx secteur disque

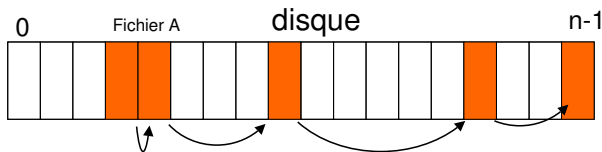
Classique triple indirection I

- Identification des blocs de données d'un fichier
 - inœud contient une liste de numéros de blocs
 - fichiers de taille variable
- Numéros de blocs
 - numéros des 10 premiers blocs de données
 - numéro d'un bloc contenant les numéros des blocs suivants...
- Tout est bloc !
 - blocs de données
 - blocs d'indirection, de double/triple indirection...
 - blocs contenant les inœuds etc.

The diagram illustrates a hierarchical tree structure. At the top left is a vertical stack of 15 horizontal bars, labeled 'inoeud' at the top. To its right is a vertical stack of three boxes labeled '1', '2', and '3', collectively labeled 'blocs' at the top. Three green arrows point from the top three bars of the 'inoeud' stack to the '1', '2', and '3' boxes respectively. Below the 'inoeud' stack, a magenta arrow points from the 7th bar to a magenta-colored stack of 5 horizontal bars. An orange arrow points from the 10th bar of the 'inoeud' stack to an orange-colored stack of 5 horizontal bars. From the magenta stack, two arrows point to a box labeled '11' and another box. From the orange stack, two arrows point to two boxes. The tree continues to branch out with more stacks and boxes, representing a complex hierarchical structure.

Allocation chaînée I

- Un fichier occupe une liste chaînée de blocs sur le disque
- Chaque bloc contient une partie des données et un pointeur sur le bloc suivant



Allocation chaînée II

■ Avantages

- Possibilité d'étendre un fichier
- Allocation par bloc individuel : Tout bloc libre peut être utilisé pour satisfaire une requête d'allocations

■ Inconvénients ? Solution non adaptée à l'accès direct

- L'accès à un bloc quelconque nécessite l'accès à tous les blocs qui le précèdent
- Les pointeurs sont stockés sur disque

- Idée : Séparer les pointeurs et les données Technique
 - Utilisation d'une table d'allocation de fichier
 - (FAT : File Allocation Table)
 - A chaque bloc est associée une entrée dans la FAT qui contient le n° du bloc suivant
 - Méthode Utilisée dans MS-DOS et OS/2

0	3
1	Fin de fichier
2	4
3	6
4	1
5	
6	Fin de fichier
	■
	—

Fichiers	blocs occupés
A	0 3 6
B	2 4 1

Allocation chaînée et indexée II

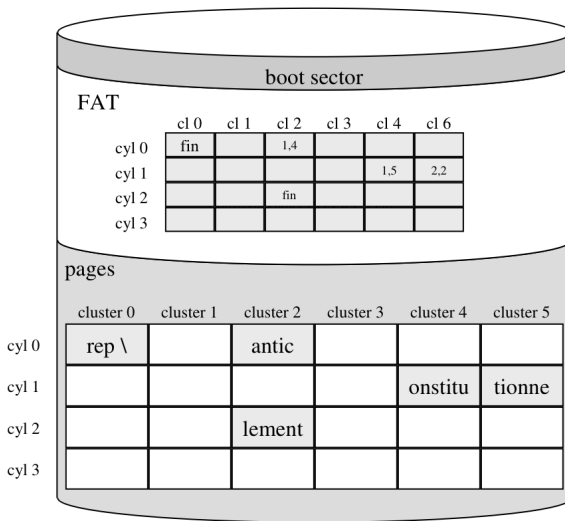
■ Avantages

- Extension des fichiers
- les blocs de données ne contiennent pas les pointeurs
- accès direct facile
 - utilise un mécanisme d'ombre pour protéger la FAT

■ Inconvénients

- Occupation de la mémoire centrale par la FAT
- Problème des disques de grande capacité
- Une table pour un disque de 1Go en blocs de 1Ko occuperait 4 Mo (+ 4Mo pour la FAT « ombre »)

SGF FAT (MSDOS & Windows)



Allocation par nœud d'information

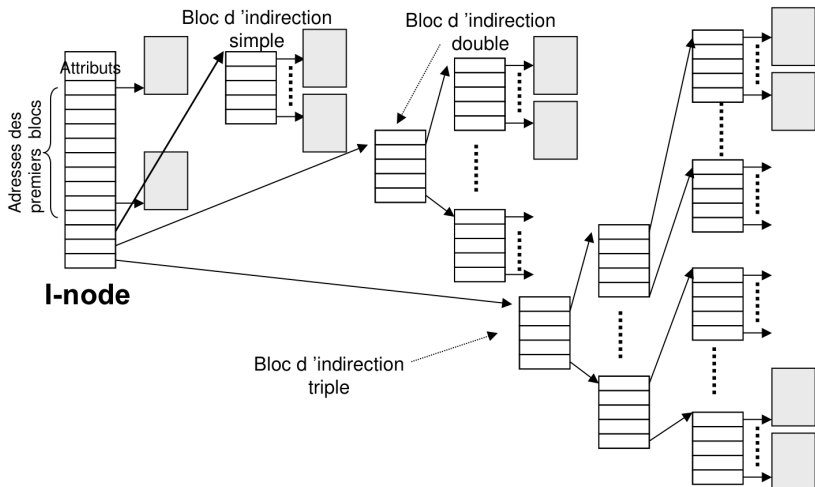
■ Idée

- Eclater la FAT en plusieurs petites tables appelées nœuds d'informations (i-node)
- A chaque fichier est associé un nœud d' information
- Chaque table contient les attributs et les adresses sur le disque des blocs du fichier

■ Unix

- La table est hiérarchisé sur Unix
- FS System V
 - 10 direct, 1 simple indirection, 1 double indirection, 1 triple indirection
- BSD Fast FS / UFS
 - 12 direct, 1 simple indirection, 2 double indirection

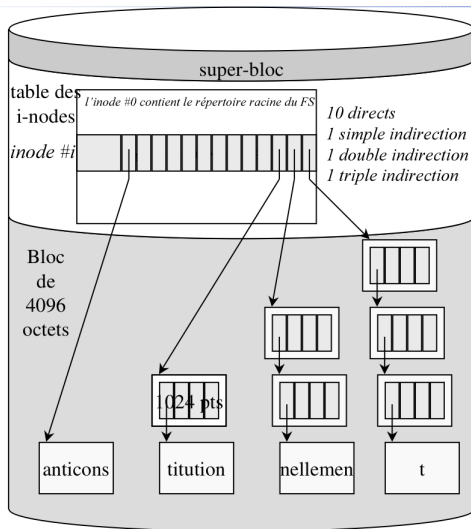
Structure d'un nœud d'information (FS SYSV)



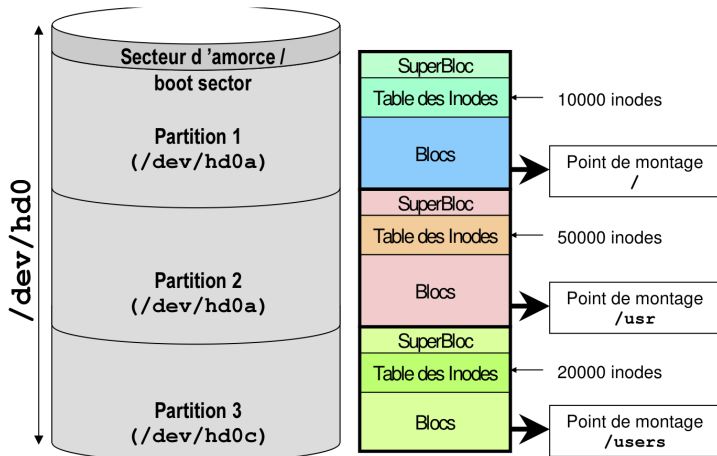
Allocation par nœud d'information

- Seuls les nœuds d'information des fichiers ouverts sont chargés en mémoire centrale
- Allocation par bloc individuel
- Accès direct facile
 - nécessite au maximum 4 accès disque
 - Adaptée aux disques de très grande capacité

le Système de Fichier d'Unix System V



Partitionnement système



Processeur/Processus

- Programme, processus... processeur
 - entité matérielle
 - désigne l'utilisation du processeur
 - Affectation du processeur à un processus
 - pour un temps donné
 - permet de faire progresser le processus
 - Choix de cette affectation = ordonnancement
 - système multiprocessus
 - choix à la charge du système d'exploitation (...à suivre)
 - P..., p..., p..., parallélisme, pseudo-parallélisme
 - plusieurs processus, un processeur
 - entrelacement des processus
- ⇒ **ordonnancement**

Processus = abstraction !

- Processus = exécution abstraite d'un programme
 - indépendante de l'avancement réel de l'exécution
- Exécution d'un programme = réunion des instants d'exécution réelle du programme
 - dépend de la disponibilité du processeur
- Processus = abstraction
 - désigne une entité identifiable
 - par exemple : priorité d'un processus
 - parallélisme, simultanéité, interaction... de deux processus
- Compétition (race condition)
 - résultats de deux processus dépend de cet entrelacement
 - par exemple à cause d'accès partagés à un fichier...
 - à éviter

Processus & ressources

- Processus = exécution d'un programme
 - requiert des ressources
- Ressource
 - entité nécessaire à l'exécution d'un processus
 - ressources matérielles : processeur, périphérique...
 - ressources logicielles : variable...
- Caractéristiques d'une ressource
 - état : libre, occupée
 - nombre de possibles utilisations concurrentes
 - (ressource à accès multiples)
- Ressources indispensables à un processus
 - mémoire propre (mémoire virtuelle)
 - contexte d'exécution (état instantané du processus)
 - pile (en mémoire)
 - registres du processeur

Gérer des processus en POSIX

Gestion POSIX des processus :

- Création, terminaison, chargement, recouvrement
- Processus :
 - espace d'adressage (modifiable au cours de l'exécution)
 - activité
 - état système (signaux, entrées/sorties, état, ...)

Deux modes :

- utilisateur (user mode)
- système (kernel mode)

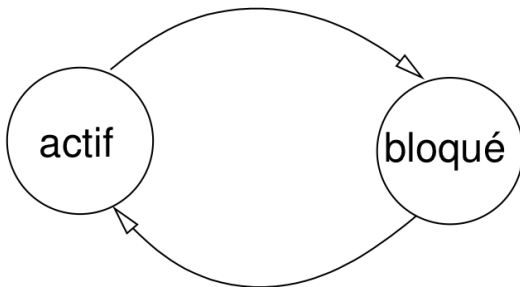
- 1 Plan du Cours
- 2 Introduction
- 3 Système d'exploitation
- 4 Principes des systèmes d'exploitation
- 5 Utilisation de la bibliothèque C standard et de l'API POSIX
- 6 Les Entrées-Sorties

9 Bibliographie

Etats logiques

■ 2 états possibles :

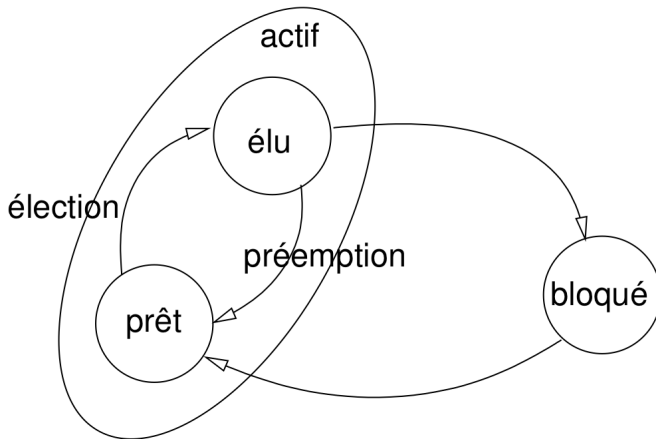
- **Actif** : le processus s'exécute (ou est prêt à s'exécuter)
- **Bloqué** : le processus ne peut pas continuer à s'exécuter
 - Indisponibilité d'une ressource, Verrou, Sémaphore, ...



Etats effectifs I

- Un processeur est une ressource
 - un seul processus peut s'exécuter, à la fois, sur un processeur (core)
 - les autres processus sont en attente
- Lorsque le processus est actif, il peut être dans 2 états :
 - **Elu/actif** :
 - le processus s'exécute sur le processeur
 - il s'agit du processus élu par le processeur pour être exécuté
 - **Prêt** :
 - le processus est prêt à être exécuté
 - pas de blocage au niveau des ressources
 - mais il n'a pas été élu par l'ordonnanceur

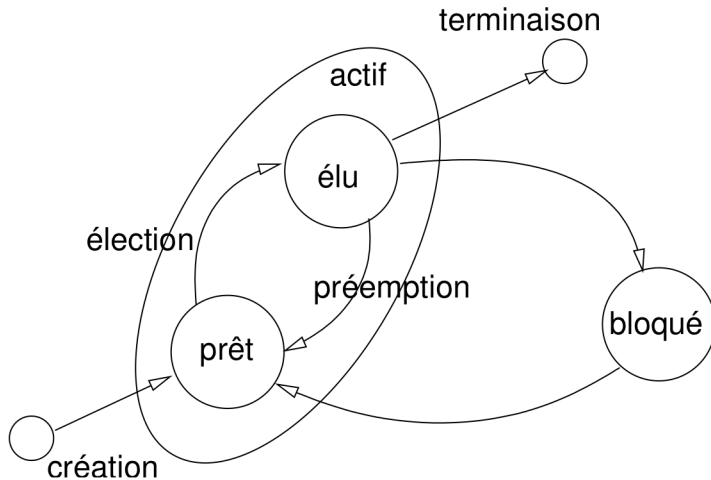
Etats effectifs II



Vie d'un processus I

- Création d'un processus :
 - Un processus est toujours créé par un processus père
 - il passe alors dans l'état prêt
 - Mais il peut être élu immédiatement par l'ordonnanceur et passer dans l'état actif
- Terminaison d'un processus
 - Un processus élu peut se terminer

Vie d'un processus II



Activité d'un processus I

Au niveau processeur

- Une activité possède un contexte d'exécution = valeurs d'un certain nombre de registres.
- Lorsqu'une activité est élue, le système sauvegarde le contexte de l'activité qui perd le processeur et restaure celui de l'activité élue.

L'activité d'un processus présente différents états :

- **prêt** : le processus attend que l'ordonnanceur lui attribue le processeur.
 - prêt → actif : le processus est élu par l'ordonnanceur
- **élu/actif** (user ou kernel) : le processus s'exécute en mode user ou kernel.

Activité d'un processus II

- actif kernel → actif utilisateur : le processus revient d'un appel système
- actif user → actif kernel : le processus réalise un appel système ou une interruption est survenue
- actif kernel → endormi : attente d'un événement interne au système (libération de ressource ou terminaison d'un processus par exemple).
- actif kernel → zombi : le processus se termine

Activité d'un processus III

- **endormi** : le processus attend un événement pour passer dans l'état prêt.
 - endormi → prêt : l'événement attendu s'est produit
 - endormi → suspendu : signal (par exemple SIGSTOP ou SIGTSTP)
 - endormi → prêt : signal (par exemple SIGCONT)
- **suspendu** : le processus a reçu un signal d'arrêt (SIGSTOP) et attend un signal (SIGCONT) pour passer dans l'état prêt.
- **zombi** : le processus est terminé mais son père n'a pas pris connaissance de sa terminaison.

Hiérarchie des processus

- Création d'un processus par un processus père
 - hiérarchie processus ancêtre init

Attributs d'un processus

- Identification univoque
 - process ID
 - numéro entier pid_t
 - numéro du processus père
- Propriétaire
 - propriétaire réel
 - utilisateur qui a lancé le processus, son groupe
- Propriétaire effectif, et son groupe
 - détermine les droits du processus
 - peut être modifié / propriétaire réel

Attributs d'un processus

#include <unistd.h>

Connaitre son PID et celui du père :

```
pid_t getpid(void);
```

```
pid_t getppid(void);
```

Connaitre le propriétaire réel/effectif :

```
uid_t getuid(void);
```

```
uid_t geteuid(void);
```

Modifier son propriétaire :

```
int setuid(uid_t uid);
```

Attributs d'un processus

Connaître le groupe réel :

```
gid_t getgid(void);
```

```
gid_t getegid(void);
```

Modifier son groupe effectif :

```
int setgid(gid_t gid);
```

Attributs d'un processus (autres) I

■ Répertoire de travail

- origine de l'interprétation des chemins relatifs

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

```
char *getcwd(char *buffer, size_t bufsz);
```

- retourne NULL en cas d'échec peut être changé

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

```
int chdir(const char path);
```

■ Date de création du processus

- en secondes depuis l'époque, 1er janvier 1970

■ Temps CPU consommés

- par le processus / par ses processus fils terminés
- en mode utilisateur / en mode noyau
- structure structtms

Attributs d'un processus (autres) II

■ Temps CPU consommés

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

```
clock_t times(struct tms *buffer);
```

- retourne le temps en tics horloge depuis un temps fixe dans le passé (démarrage du système par exemple)
- retourne 4 champs dans la structure structtms

```
struct tms {
    clock_t tms_utime; /* User CPU time. */
    clock_t tms_stime; /* System CPU time. */
    clock_t tms_cutime; /* User CPU time of terminated
                          child processes. */
    clock_t tms_cstime; /* System CPU time of terminated
                          child processes. */
}
```


Attributs d'un processus (autres) III

- tics horloge survenus alors que le processus était actif
- conversion en secondes par division par la constante de configuration `_SC_CLK_TCK`

Attributs d'un processus (autres) IV

Changer le répertoire de travail d'un processus :

```
int chdir(const char *reference);
```

```
char *getcwd(char *referenceAbsolue, size_t taille);
```

Temps CPU :

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

```
clock_t times(struct tms *tempsCPU);
```

Umask de création des fichiers :

```
mode_t umask(mode_t masque);
```

Table des descripteurs : créée/copiée du père.

Temps CPU

```
static struct tms st_cpu;  
static struct tms en_cpu;  
static clock_t st_time;  
static clock_t en_time;
```

```
static float tics_to_seconds(clock_t tics) {  
    return tics/((float)sysconf(_SC_CLK_TCK));  
}
```

```
void start_clock() {  
    st_time = times(&st_cpu);  
}
```

```
void end_clock() {  
    en_time = times(&en_cpu);  
    printf("Real Time: % 2f User Time % 2f System Time % 2f\n",  
           (float)en_time - (float)st_time, (float)en_time, (float)en_time - (float)st_time);  
}
```

Ressources système

Changer la priorité :

```
int nice(int incr);
```

Valeurs limites d'utilisation du CPU :

```
int getrlimit(int ressource, struct rlimit *rlim);
```

```
int setrlimit(int ressource, const struct rlimit *rlim);
```

Permet de modifier : RLIMIT_CPU, RLIMIT_DATA, ...

procfs dans Linux

Répertoire /proc/XXX où¹ XXX=numéro du processus

Informations :

- cmdline : commande exécutée
- cpu : temps cpu consommé
- cwd : répertoire de travail
- environ : environnement du processus
- exe : binaire exécuté
- fd : liste des descripteurs
- maps : régions mémoire du processus
- mem : manipulation de la mémoire
- stat, status : statut du processus

Lancement d'un programme

- Lancement d'un programme réalisé en deux étapes
 - création d'un nouveau processus par clonage de son père
 - copie du processus père
 - sauf l'identité, pid
 - mutation pour exécuter un nouveau programme
- Seul mécanisme possible
- Clonage
 - appel système `fork()`
- Mutation
 - appel système `execve()`
 - famille de fonctions de bibliothèque `exec*()`

Création de processus I

Création d'un processus :

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

```
pid_t fork(void);
```

- duplique le processus courant
- Copie exacte (espace d'adressage, table des descripteurs, ...)
- Non copié : pid, temps cpu, signaux, priorité, verrous
- retourne :
 - retourne le pid du processus fils créé dans le processus père
 - retourne 0 dans le processus fils
 - retourne -1 en cas d'erreur

Création de processus II

Attention : la table des fichiers ouverts n'est pas "dupliquée" !

- Position courante "commune"
- Solution : réouvrir le fichier ou verrous

Clonage de processus I

Clonage de processus II

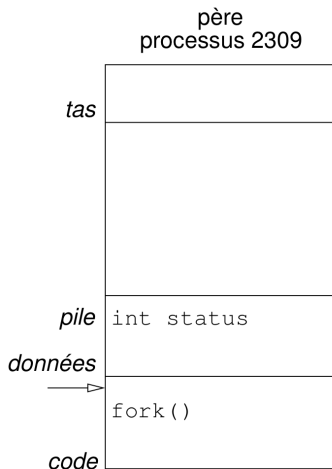
```

int main() {
    pid_t status;
    printf("[%d] Je vais engendrer\n", getpid());
    status = fork();
    switch (status) {
        case -1 :
            perror("Creation processus");
            exit(EXIT_FAILURE);
        case 0 :
            printf("[%d] Je viens de naitre\n", getpid());
            printf("[%d] Mon pere est %d\n", getpid(), getppid());
            break;
        default:
            printf("[%d] J'ai engendre\n", getpid());
    }
}
    
```

Clonage de processus III

```
        printf("[%d] Mon fils est %d\n", getpid(), status);  
    }  
    printf("[%d] Je termine\n", getpid());  
    exit(EXIT_SUCCESS);  
}
```

Clonage de processus IV



```
% ./fork
```

```
[2309] Je vais engendrer
```

```
[2310] Je viens de naitre
```

```
[2310] Mon pere est 2309
```

```
[2310] Je termine
```

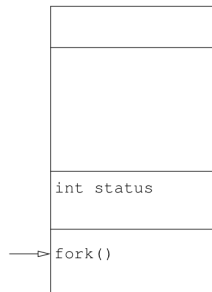
```
[2309] J'ai engendre
```

```
[2309] Mon fils est 2310
```

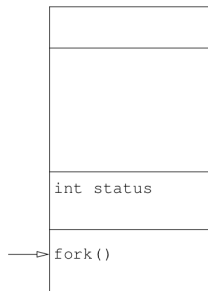
```
[2309] Je termine
```

Clonage de processus V

père
processus 2309



fils
processus 2310



```
% ./fork
```

```
[2309] Je vais engendrer
```

```
[2310] Je viens de naître
```

```
[2310] Mon pere est 2309
```

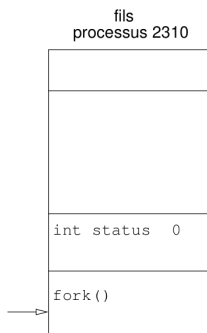
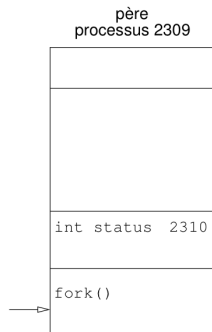
```
[2310] Je termine
```

```
[2309] J'ai engendré
```

```
[2309] Mon fils est 2310
```

```
[2309] Je termine
```

Clonage de processus VI



```
% ./fork
```

```
[2309] Je vais engendrer
```

```
[2310] Je viens de naître
```

```
[2310] Mon pere est 2309
```

```
[2310] Je termine
```

```
[2309] J'ai engendre
```

```
[2309] Mon fils est 2310
```

```
[2309] Je termine
```

Héritage père/fils I

- Mémoire du processus fils = copie de la mémoire du processus père
- Copie de références = partage
 - la mémoire du père est copiée pour créer
 - la mémoire du fils la mémoire du processus référence des structures systèmes
 - des structures systèmes restent partagées entre père et fils
- Conséquences de la copie de la mémoire
 - tampons d'écriture de la bibliothèque standard d'entrées/sorties dupliqués
 - vider ce tampon avant fork() (par un appel à fflush())
 - problème des tampons de lecture
- Conséquences du partage de structures systèmes
 - descripteurs de fichiers dupliqués

Héritage père/fils II

- mais entrées dans la table des fichiers ouverts partagées (f_pos...)
- Attributs non copiés
 - explicitement gérés par le système
 - numéro de processus !
 - numéro de processus du père !!
 - temps d'exécution remis à zéro
 - verrous sur les fichiers détenus par le père
 - signaux (...à suivre)

Héritage père/fils III

```
int main(int argc, char *argv[])  
{  
    int fd;  
    char buf[10];  
    fd = open(argv[1], O_RDWR);  
    assert(fd != -1);  
    read(fd, buf, 2);  
    switch (fork()) {  
        case -1 :  
            perror("Creation processus");  
            exit(EXIT_FAILURE);  
        case 0 :  
            write(fd, "foo", 3);  
            sleep(2);
```

Héritage père/fils IV

```
    read(fd, buf, 3);  
    buf[3]='\0';  
    printf("[%d] fils a lu '%s'\n", getpid(), buf); break;
```

default:

```
    write(fd, "bar", 3);  
    sleep(1);  
    read(fd, buf, 3); buf[3]='\0';  
    printf("[%d] pere a lu '%s'\n", getpid(), buf);
```

```
}
```

```
close(fd);
```

```
exit(EXIT_SUCCESS);
```

```
}
```

Héritage père/fils V

```
% echo _123456789_12345_ > xampl
% ./fdshare xampl
[3101] pere a lu ' 89_'
[3102] fils a lu ' 123'
% cat xampl _1
barfoo89_12345_
```

- 1 Plan du Cours
 - 2 Introduction
 - 3 Système d'exploitation
 - 4 Principes des systèmes d'exploitation
 - 5 Utilisation de la bibliothèque C standard et de l'API POSIX
 - 6 Les Entrées-Sorties
 - 8 Processus
 - Introduction
 - Etats d'un processus
 - Hiérarchie des processus
 - Attributs d'un processus
 - Création de processus
 - **Terminaison de processus**
 - Processus orphelins & zombies
 - Mutation de processus
 - Redirections des entrées/sorties
 - Ordonnancement
 - 9 Bibliographie

Terminaison des fils

- Processus termine
 - appel `_exit()`
 - appel `exit()` : ferme les fichiers ouverts
 - positionne une valeur de retour
 - succès \Rightarrow `exit(0)`
- Processus père peut consulter cette valeur de retour
 - indication de la terminaison du fils : succès ou échec...
- Attente de la terminaison d'un fils

#include <sys/wait.h>

`pid_t wait(int *pstatus);`

- retourne le PID du fils
- -1 en cas d'erreur (n'a pas de fils...)
- bloquant si aucun fils n'a terminé *pstatus renseigne sur la terminaison du fils

Terminaison des fils

- Renseignements concernant la terminaison d'un fils
 - rangées dans l'entier status pointé par pstatus
 - raison de la terminaison

macro	signification
WIFEXITED(status)	le processus a fait un exit()
WIFSIGNALED(status)	le processus a reçu un signal (...à suivre)
WIFSTOPPED(status)	le processus a été stoppé (...à ? suivre)

- valeur de retour
 - si WIFEXITED(status) !
 - 8 bits de poids faible seulement
 - accessible par la macro WEXITSTATUS(status)
- numéro de signal ayant provoqué la terminaison / l'arrêt
 - si WIFSIGNALED(status)/ WIFSTOPPED(status) !
 - accessible par la macro

WTERMSIG(status)/ WSTOPSIG(status)

Termination des fils I

```
int main(int argc, char *argv[])  
{  
    int status;  
    pid_t pid, pidz;  
    switch (pid = fork()) {  
        case -1 :  
            perror("Creation processus");  
            exit(EXIT_FAILURE);  
        case 0 :  
            printf("[%d] fils eclair\n", getpid());  
            exit(2);  
            break;  
        default:
```


Termination des fils II

```

printf("[%d] pere a cree %d\n", getpid(), pid);
pidz = wait(&status);
if (WIFEXITED(status))
    printf("[%d] mon fils %d a termine normlement,\n"
           "[%d] code de retour: %d\n",
           getpid(), pidz,
           getpid(), WEXITSTATUS(status));
else
    printf("[%d] mon fils a termine anormlement\n",
           getpid());
}
exit(EXIT_SUCCESS);
}
    
```

Termination des fils III

```
% ./waitexit  
[3774] pere a cree 3775  
[3775] fils eclair  
[3774] mon fils 3775 a termine normlement,  
[3774] code de retour: 2
```

Termination des fils

- Attente d'un fils désigné
 - primitive

#include <sys/wait.h>

pid_t waitpid(pid_t pid, **int** *pstatus, **int** options);

- pid désigne le fils à attendre
- pid à 0 = fils quelconque
- options combinaison binaire
 - WNOHANG appel non bloquant
 - WUNTRACED processus stoppe

Processus orphelins I

- Terminaison d'un processus parent ne termine pas ses processus fils
 - les processus fils sont **orphelins**
- Processus initial init (PID 1) récupère les processus orphelins

Processus orphelins II

```
int main() {  
    pid_t status;  
    printf("[%d] Je vais engendrer\n", getpid());  
    status = fork();  
    switch (status) {  
        case -1 :  
            perror("Creation processus");  
            exit(EXIT_FAILURE);  
        case 0 :  
            printf("[%d] Je viens de naitre\n", getpid());  
            printf("[%d] Mon pere est %d\n", getpid(), getppid());  
            break;  
        default:  
            printf("[%d] J'ai engendre\n", getpid());
```

Processus orphelins III

```
        printf("[%d] Mon fils est %d\n", getpid(), status);  
    }  
    printf("[%d] Je termine\n", getpid());  
    exit(EXIT_SUCCESS);  
}
```

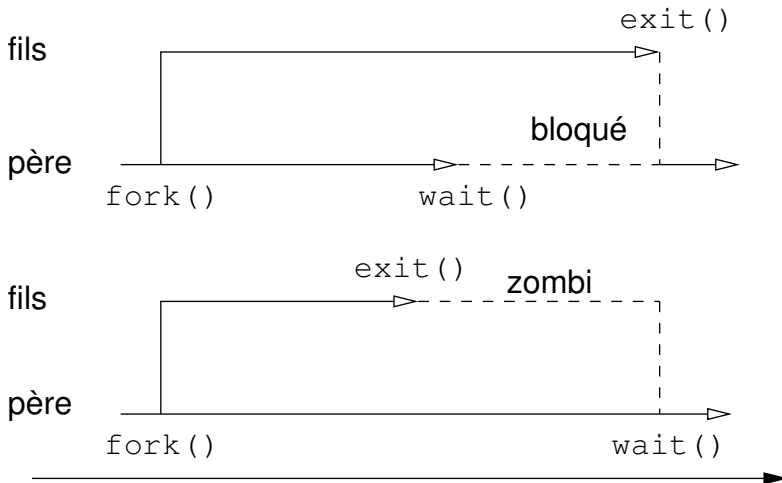
Processus orphelins IV

```
% ./fork
[10825] Je vais engendrer
[10825] J'ai engendre
[10825] Mon fils est 10826
[10825] Je termine
[10826] Je viens de naitre
[10826] Mon pere est 1
[10826] Je termine
```

Processus zombis I

- **Zombi** = état d'un processus
 - ayant terminé
 - non encore réclamé par son père
- Eviter les processus zombis
 - système garde des informations relatives au processus pour les retourner à son père
 - encombre la mémoire
- Technique du double fork()

Processus zombis II



Mutation de processus I

- Rappel : lancement d'un programme
 - 1- clonage
 - 2- **mutation**
- Mutation = remplacement
 - remplacement du code à exécuter
 - c'est le même processus
 - on parle de recouvrement du processus

Mutation de processus II

- Nouveau programme hérite de l'environnement système
 - même numéro de processus PID, PPID
 - héritage des descripteurs de fichiers ouverts
 - pas de remise à zéro des temps d'exécution
 - héritage du masque des signaux, etc.
- Famille de primitives
 - appel système `execve()`
 - fonctions de bibliothèque `exec*()`

Mutation de processus III

■ Appel système

```
#include <unistd.h>  
int execve(const char *filename,  
            char *const argv[],  
            char *const envp[]);
```

- va exécuter le programme filename
- ne retourne pas, sauf erreur de recouvrement

Appel système execve()

■ Appel système

```
int execve(const char *filename,
           char *const argv[],
           char *const envp[]);
```

- va exécuter le programme filename
- qui, s'il correspond à un programme C, invoquera

```
int main(int argc,
         char *const argv[],
         char *const envp[]);
```

■ De manière générale

- le fichier désigné par filename est exécuté
- avec les arguments argv[]
- et les variables d'environnement envp

Appel système execve()

int

main(int argc, char *argv[], char **arge)

{

 execve(argv[1], argv+1, arge);

 perror("recouvrement");

 exit(EXIT_SUCCESS);

}

\$ ls *

execve execve.c fork fork.c

\$./execve /bin/ls *

execve execve.c fork fork.c

\$./execve ls *

recouvrement: No such file or directory

La famille exec*() I

- Plusieurs fonctions de bibliothèque
 - écrites au dessus de execve()
 - différents moyens de passer les paramètres : tableau, liste
 - spécification ou non des variables d'environnement
- Deux spécifications de la commande à exécuter
 - chemin complet, filename
 - absolu ou relatif
 - nom de commande, command
 - recherchée dans les chemins de recherche, \$PATH

La famille exec*() II

#include <unistd.h>

int execl(**const char** *filename, **const char** *arg0, ... /*, (char *)
0 */

int execv(**const char** *filename, **char** ***const** argv[]);

int execl(**const char** *filename, **const char** *arg0, .../*, (char
*)0, char ***const** envp[] */);

int execlp(**const char** *command, **const char** *arg0, ... /*, (
char *)0 */

int execvp(**const char** *command, **char** ***const** argv[]);

La famille exec*() III

■ Illustration

■ exécution du programme ls

```
% ls
execve.c fork.c
% which ls
/bin/ls
```

■ par différents appels d'exec*() :

```
char *argv[]={ "ls", "execve.c", "fork.c", (char *)0 };
execv("/bin/ls", argv);
execvp("ls", argv);
execl("/bin/ls", "ls", "execve.c", "fork.c", (char *)0);
execlp("ls", "ls", "execve.c", "fork.c", (char *)0);
```

Recouvrement I

- Ecrasement de l'espace d'adressage du processus
- Nouveau programme chargé et s'exécute sur de nouvelles données

Le paramètre `arg0` sert de premier argument à la fonction `main` :

```
int execl(const char *nomFichier, const char * arg0, ..., NULL
);
execl("/bin/l", "ls", "-l", "/", NULL);
```

Idem + la recherche du fichier dans `$PATH`.

```
int execlp(const char *nomFichier, const char * arg0, ...,
NULL);
```

Recouvrement II

Idem execl + un nouvel environnement :

```
int execl(const char *nomFichier, const char * arg0, ...,  
        NULL, const char **env );
```

Identique à execl :

```
int execv(const char *nomFichier, const char * argv[]);
```

Recouvrement III

Identique à `execlp` :

```
int execvp(const char *nomFichier, const char * argv[]);
```

Identique à `execle` :

```
int execve(const char *nomFichier, const char * argv[], const  
           char **env);
```

Complémentarité fork() et exec*() I

- Faire exécuter un programme par un nouveau processus
 - nouveau processus pour chaque commande exécutée
 - ce que fait le shell

```
int main()
```

```
{
    printf("Je suis %5d de
           pere %5d\n",
           getpid(), getppid());
    exit(EXIT_SUCCESS);
}
```

```
% ./pid
Je suis 7851 de pere 717
% ./pid
Je suis 7852 de pere 717
% echo $$
717
```

- erreur dans le programme n'affecte pas le shell
- changement d'attributs dans le programme n'affectent pas le shell
- (sauf commandes internes : par exemple cd...)

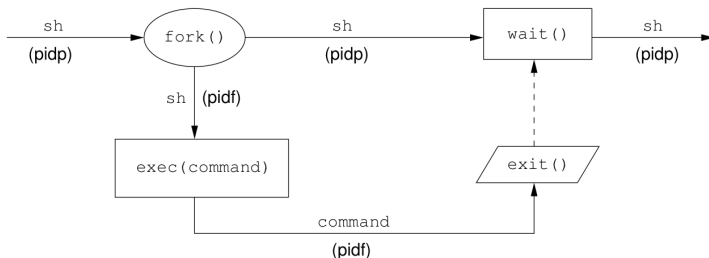
Complémentarité fork() et exec*() II

- Réalisation en deux étapes
 - 1- clonage fork()
 - 2- mutation exec*()
- Souplesse de par la combinaison deux primitives
 - réalisation d'opérations entre le fork() et l'exec*()
 - exemple : redirection des entrées/sorties standard

L'exemple du shell I

■ Exécution d'une commande command par le shell sh

% **command**

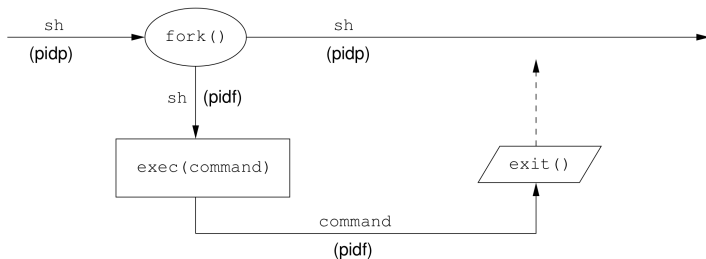


■ 4 primitives fork(), exec(), exit(), et wait()

L'exemple du shell II

- Exécution en arrière plan d'une commande command par le shell sh

% **command** &



- Le shell n'attend plus la terminaison de son fils

Héritage des descripteurs lors de mutation I

- Descripteurs de fichiers sont hérités par le nouveau programme
- largement utilisé par le shell pour assurer les redirections
- exemple :

```
% cmd > out
```

- le shell associe le descripteur 1, STDOUT_FILENO à un fichier out (voir dup())...à suivre)

 exec("cmd", ...)
- le programme cmd écrit sur STDOUT_FILENO
- ... qui référence le fichier out

Héritage des descripteurs lors de mutation II

- Sauf si positionnement de l'option *close on exec* sur le descripteur
 - positionnement de FD_CLOEXEC par un appel à fcntl()

```
int main(int argc, char *argv[])
{
    fcntl(STDOUT_FILENO, F_SETFD,
          fcntl(STDOUT_FILENO, F_GETFD, 0) |
          FD_CLOEXEC);
    execvp(argv[1], argv+1);
    perror ("execvp");
    exit(EXIT_SUCCESS);
}
```

Héritage des descripteurs lors de mutation III

```
% ./cloexec true
```

```
% ./cloexec ls
```

```
ls: write error: Bad file descripton
```

Redirections des entrées/sorties I

- Associer un descripteur de fichier donné à un fichier
 - exemple associer le descripteur de fichier donné 1, `STDOUT_FILENO` au fichier `out`
 - les écritures avec le descripteur `STDOUT_FILENO` se feront dans le fichier `out`
- Primitive POSIX

int dup(**int** fildes);

- recherche le plus petit descripteur disponible
- en fait un synonyme du descripteur `fildes`
- les deux descripteurs partagent la même entrée dans la table des fichiers ouverts du système
- écrire (`/lire`) dans ce plus petit descripteur revient à écrire (`/lire`) dans le fichier référencé par `fildes`

Redirections des entrées/sorties II

- Ce plus petit descripteur correspond à un fichier qui vient d'être fermé...

Redirections des entrées/sorties III

■ Primitive POSIX dup2()

- explicitation du descripteur synonyme créé !

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

```
int dup2(int fildes, int fildes2);
```

- force fildes2 à devenir un synonyme de fildes
- ferme le descripteur fildes2 préalablement si nécessaire

■ Utilisations ultérieures de fildes2

- référencent le fichier identifié par fildes

Exemple 1 I

```
static void print_inode(const char *str, int fd)
{
    struct stat st;
    fstat(fd, &st);
    fprintf(stderr, "\t%s : inode %d\n", str, st.st_ino);
}

int main(int argc, char *argv[])
{
    int fdin, fdout;

    fdin = open(argv[1], O_RDONLY);
```

Exemple 1 II

```
fdout = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC  
    , 666);
```

```
fprintf(stderr, "Avant dup2() :\n");  
print_inode("fdin", fdin);  
print_inode("fdout", fdout);  
print_inode("stdin", STDIN_FILENO);  
print_inode("stdout", STDOUT_FILENO);
```

```
dup2(fdin, STDIN_FILENO);  
dup2(fdout, STDOUT_FILENO);
```

```
fprintf(stderr, "Après dup2() :\n");  
print_inode("fdin", fdin);
```

Exemple 1 III

```
print_inode("fdout", fdout);  
print_inode("stdin", STDIN_FILENO);  
print_inode("stdout", STDOUT_FILENO);  
  
exit(EXIT_SUCCESS);  
  
}
```

%ls -i foo bar

ls: foo: No such file or directory

26553484 bar

%./dup2pr foo bar

Avant dup2() :

fdin : inode 15

fdout : inode 26553484

stdin : inode 1051

stdout : inode 1051

Après dup2() :

Exemple 2 I

```
#define BSIZE 512
```

```
static void cat()
```

```
{
```

```
    unsigned char buffer[BSIZE];
```

```
    int nread;
```

```
    while((nread=read(STDIN_FILENO, buffer, BSIZE)))
```

```
        write(STDOUT_FILENO, buffer, nread);
```

```
}
```

```
int main(int argc, char *argv[])
```

```
{
```

Exemple 2 II

```
int fdin, fdout;
```

```
fdin = open(argv[1], O_RDONLY);
```

```
fdout = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC  
    , 666);
```

```
dup2(fdin, STDIN_FILENO);
```

```
dup2(fdout, STDOUT_FILENO);
```

```
close(fdin);
```

```
close(fdout);
```

```
cat();
```

Exemple 2 III

```
close(STDIN_FILENO);
close(STDOUT_FILENO);

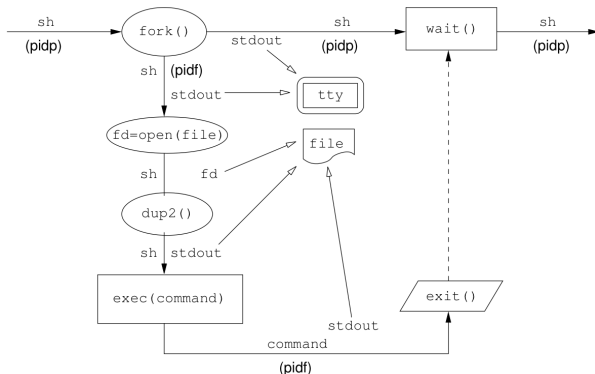
exit(EXIT_SUCCESS);
}
```

```
% cat bar
% cat foo
azerty
qwerty
dvorak
% ./dup2cat foo bar
% cat bar
azerty
qwerty
dvorak
```

Redirection : l'exemple du shell

- Exécution d'une commande command redirigée sur un fichier file

% **command** > file



Ordonnancement = virtualisation

- Multiprogrammation
 - plusieurs processus prêts
 - un seul processeur
- Virtualisation du processeur
 - chaque processus détient le processeur tour à tour
- Ordonnancement
 - définition et mise en œuvre de ce « tour à tour »
 - choisir le processus élu
- Cœur du système d'exploitation
 - bien que tout algorithme d'ordonnancement puisse convenir...
- Ce que nous allons voir ici
 - très grandes lignes de quelques algorithmes d'ordonnancement
 - norme POSIX, système Unix
 - paramétrage par l'utilisateur

Objectif d'un algorithme d'ordonnancement (1/2)

- Divers et multiples !
 - selon les systèmes d'exploitation
 - systèmes d'exploitation spécialisés
 - temps-réel, multimédia, traitement par lots, interactifs...
 - systèmes d'exploitation généralistes
- Équité
 - attribuer à chaque processus un temps processeur équitable
- Réactif
 - répondre rapidement aux requêtes
- Débit
 - nombre de traitements par unité de temps

Objectif d'un algorithme d'ordonnancement (2/2)

■ Performance

- faire en sorte que le processeur soit occupé en permanence
- faire en sorte que toutes les parties du système soient occupées

■ Garanties

- respecter les échéances

Préemptibilité / quantum

■ Préemptibilité des processus

■ ordonnancement préemptif

- peut interrompre l'exécution d'un processus en cours
- possibilité de réquisition du processeur

■ ordonnancement non préemptif

- changement de contexte à la fin du processus, ou
- changement de contexte volontaire du processus en cours (yield())

■ Quantum

- unité de temps processeur attribué au processus élu
- réquisition du processeur à la fin du quantum
- influence de la durée du quantum
 - petit : surcoût des changements de contexte
 - grand : mauvaise réactivité

Priorité des processus

- Propriété d'un processus
 - niveau de priorité
 - dénote l'importance relative des processus
- Numérotation des priorités
 - attention, souvent les processus les plus prioritaires ont un numéro de priorité plus faible !
- Priorité ajustable

#include <unistd.h>

int nice(**int** incr);

- augmente le valeur de priorité du processus
 - donc diminue sa priorité !
 - courtoisie envers les autres utilisateurs
 - incr négatif autorisé pour le superutilisateur
- Commande nice
 - nice [-n increment] command [argument...]

Politiques d'ordonnancement I

- Multitudes de politiques possibles
- Trois politiques définies par POSIX
 - premier arrivé, premier servi : FIFO, first-in, first-out
 - tourniquet : RR, round robin
 - autre : en général l'ordonnancement Unix traditionnel
- FIFO
 - non préemptif
 - changement de contexte quand le processus rend la main ou termine
 - POSIX : un nombre fini de niveaux de priorité FIFO
 - processus préempté par un processus plus prioritaire
- Tourniquet
 - processeur alloué successivement à chacun des processus
 - expiration du quantum : réquisition du processeur

Politiques d'ordonnancement II

- processus placé en queue de la file d'attente
- POSIX : un nombre fini de niveaux de priorité RR
- élection des seuls processus de plus forte priorité

Politiques d'ordonnancement III

■ Politique Unix classique

- quantum, à priorité dynamique
- priorité de base (dite statique) + priorité dynamique
- principe d'extinction des priorités : évite les famine
- priorité dynamique diminue au fur et à mesure que le processus consomme du temps processeur

■ Ordonnancement POSIX

- des processus gérés par les trois politiques
- si un processus associé à FIFO
- sinon si un processus associé à tourniquet
- sinon les processus associés à l'autre politique

Primitives d'ordonnancement I

■ Récupérer les paramètres d'ordonnancement

- politique d'ordonnancement, une valeur parmi SCHED_FIFO, SCHED_RR, et SCHED_OTHER

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

```
int sched_getscheduler(pid_t pid);
```

- structure structsched_param

```
struct sched_param {  
    int sched_priority;  
    ...
```

- paramètre de l'ordonnancement

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

```
int sched_getparam(pid_t pid, struct sched_param *param);
```

Primitives d'ordonnancement II

■ Positionner les paramètres

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

```
int sched_setscheduler(pid_t pid, int policy,  
                        const struct sched_param *param)
```

Primitives d'ordonnancement III

■ Intervalles de priorité

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

```
int sched_get_priority_max(int policy);
```

```
int sched_get_priority_min(int policy);
```

■ Quantum

- dans le seul cas de la politique tourniquet

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

```
int sched_rr_get_interval(pid_t pid, struct timespec *interval)  
    ;
```

- structure structtimespec

Primitives d'ordonnancement IV

```
struct timespec {
    time_t tv_sec; /* Seconds. */
    long int tv_nsec; /* Nanoseconds. */
}
```

■ Utilisation effective

- faible...
- sauf FIFO / tourniquet : urgence de processus temps-réel
 Â« mou Â»
- multimédia sur une système d'exploitation généraliste

Sommaire

1 Plan du Cours

2 Introduction

3 Système d'exploitation

4 Principes des systèmes
d'exploitation

5 Utilisation de la bibliothèque C
standard et de l'API POSIX

6 Les Entrées-Sorties

7 Système de fichiers

8 Processus

9 Bibliographie

Bibliographie I



J.-F. Lalande and C. Toinard.

Cours.



Philippe Marquet.

Cours, <http://www.lifl.fr/~marquet/>.



Jean-Marie Rifflet and Jean-Baptiste Yunès.

UNIX : Programmation et Communication.

Dunod, 2003.



Jean-Paul Rigault.

Programmation-système.

Ecole Polytech'Nice, 2005.

Bibliographie II



Andrew S. Tanenbaum.

Modern Operating Systems.

Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd
edition, 2007.