Programmation système

Pierre David
pda@unistra.fr

Université de Strasbourg

2019 – 2020

Plan

Licence d'utilisation

Plan

Plan

Organisation de l'UE

Thème central de l'UE : comment s'utilise un système d'exploitation ?

- au plus bas niveau :
  - utilisation des primitives systèmes avec le langage C
    ⇒ bon niveau en C
  - primitives POSIX
    ⇒ norme internationale
- comprendre les concepts manipulés par le système
  - idée : comprendre le périmètre du système d'exploitation et les concepts manipulés en les utilisant
  - processus, utilisateurs, fichiers, etc.

Organisation de l'UE

Cours complété par des travaux pratiques

- utilisation des primitives systèmes
- pratiquer, pratiquer, pratiquer...
- se familiariser avec les concepts

Travail demandé

- un QCM au début de chaque cours
  - 4 questions, note $\in$ [0,4]
  - moyenne des $n-1$ meilleures notes (sur 20)
  - coefficient 1
- 4 TP à rendre (individuels)
  - note $\in$ [0,5] (dont 1 point pour la forme du code)
  - moyenne des 3 meilleures notes (sur 20)
  - épreuve rendue, coefficient 3
- un TP noté (individuel)
  - note $\in$ [0,20]
  - épreuve écrite, coefficient 3
- un contrôle final sur table
  - note $\in$ [0,20]
  - épreuve convoquée, coefficient 3

Bibliographie

- M. Rochkind, « Unix, programmation avancée », Dunod (1991)
- W.R. Stevens, S.A. Rago, « Advanced Programming in the UNIX Environment » 3rd Ed, Addison-Wesley (2013)
- IEEE Computer Society, The Open Group « Standard for Information Technology - Portable Operating System Interface (POSIX®) - Base Specifications », IEEE Std 1003.1 (2013)

Plan

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

Comment définir ce qu'est un système d'exploitation (SE) ?

- Facile, m'sieur : Windows, Linux, FreeBSD, etc. !

Est-ce aussi simple ?

- si Linux est un SE, que sont Debian, Ubuntu, etc. ?
- et Android, iOS ?
- et QNX, RTLinux, VxWorks ?
- et Contiki, TinyOS ?

Et, au fait...

- est-ce qu'une box d'accès à l'Internet a un SE ?
- est-ce qu'un terminal X a un SE ?
- est-ce qu'une montre connectée a un SE ?
- est-ce qu'un thermostat de radiateur a un SE ?
- est-ce qu'une voiture a un SE ?

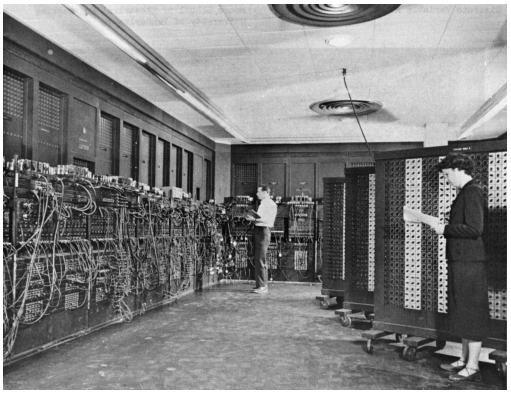
Qu'est-ce qu'un système d'exploitation ?

Pour définir ce qu'est un SE, il faut examiner l'histoire

$\Rightarrow$ quels besoins / problèmes doit résoudre un SE ?

Historique – Les premiers ordinateurs

Exemple : ENIAC (1946)



Crédit photo : U.S. Army – domaine public

Historique – Les premiers ordinateurs

Sur ces ordinateurs :

- programmer : câbler des connexions entre les unités
- résultats : à consulter sur des indicateurs lumineux

L'ordinateur :

- est très onéreux (généralement un seul exemplaire)
- est très difficile à programmer (6 programmeuses à l'origine sur l'ENIAC, la programmation dure plusieurs semaines)
- se programme par câblage physique
- n'exécute qu'un seul programme à la fois
- n'a pas ou peu de périphériques

⇒ pas de système d'exploitation

Historique – Démarrage aux clefs

Étape suivante :

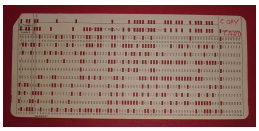
Crédit photo : "PDP-1 control board" by fjarlq / Matt - http://www.flickr.com/photos/fjarlq/147938903/ – ⓒⓘⓢ

- panneau de commande de l'ordinateur
- saisie du programme en mémoire avec des interrupteurs
- lancement du programme avec un interrupteur
- lecture du résultat avec les indicateurs lumineux

⇒ pas de système d'exploitation

Historique – Carte perforée

La carte perforée (début des années 1950)



Crédit photo : Arnold Reinhold – ⓒⓘⓢ

Crédit photo : NASA (IBM 704) – domaine public

- périphérique d'entrée des programmes (et des données)
- programme en mémoire (morte) pour démarrer le lecteur, stocker le programme en mémoire, et lancer son exécution

Historique – Moniteur

Carte perforée ⇒ entrée automatisée des programmes

Cependant :

- intervention d'un opérateur humain pour :
  - placer le bac de cartes dans le lecteur
  - lancer la lecture
  - attendre la fin du programme
  - récupérer les résultats (sur l'imprimante)
- l'ordinateur est très onéreux : toute minute de calcul perdue coûte cher !

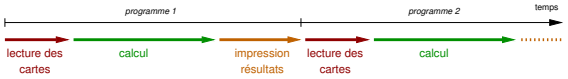
D'où : programmes « **moniteurs** » résidant en mémoire

- toujours un seul programme à la fois
- automatisation du passage des différents programmes
- traitement par « lot » (de cartes) ⇒ **batch processing**
- accès facilité aux périphériques
- premiers embryons de systèmes d'exploitation

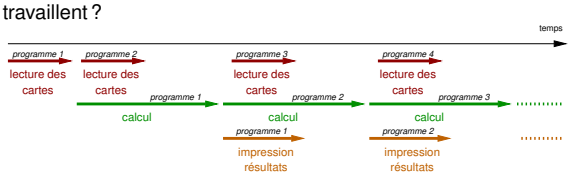
Historique – Spooling

Ordinateur onéreux ⇒ mieux le rentabiliser ?

- certains périphériques (lecteur de cartes perforées, imprimante) sont lents

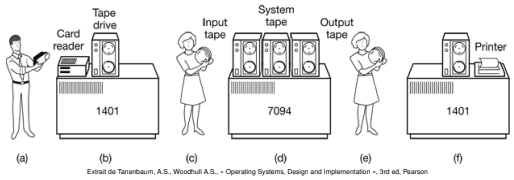


- peut-on lancer un calcul pendant que les périphériques lents travaillent ?



Historique – Spooling

Exemple : IBM 7094 (1963)



Extrait de Tanenbaum, A.S., Woodhull A.S., « Operating Systems, Design and Implementation », 3rd ed, Pearson

- IBM 7094 : très onéreux
- IBM 1401 : peu coûteux (pour l'époque)
  ⇒ dédié aux transferts entre périphériques lents et bandes magnétiques (rapides)
- transfert manuel des bandes
  ⇒ évolution vers une connexion directe
  ⇒ évolution vers des disques durs

Historique – Spooling

En résumé :

- évolution vers des « périphériques » plus « intelligents »
- fonctionnent en parallèle avec le processeur central

⇒ il faut maintenant tirer parti de ce parallélisme

- le moniteur doit gérer les ressources matérielles lentes pour en exploiter le parallélisme

Historique – Multi-programmation

Et lorsque le programme attend le résultat d'une entrée/sortie ?

- « démocratisation » des périphériques
- ex: stockage ou récupération d'une donnée temporaire sur un périphérique (bande magnétique, disque dur, etc.)
- temps mort pour le processeur
  ⇒ utiliser ce temps mort pour un autre programme

Introduction de la multi-programmation ⇒ **superviseur**

Historique – Multi-programmation

Le superviseur charge plusieurs programmes en mémoire :

- lorsque le programme en cours demande une E/S, le superviseur démarre le programme suivant
- lorsque le contrôleur d'E/S signale la fin de l'E/S, le premier programme reprend son exécution

Exemple : Atlas Supervisor de l'U. de Manchester (1962)

Historique – Multi-programmation

Questions posées par la multiprogrammation :

- comment protéger les programmes les uns des autres ?
  ⇒ accès interdit aux variables d'un autre programme
  ⇒ **unité de gestion mémoire** (composant matériel)
- comment le superviseur reprend le contrôle lorsque le contrôleur d'E/S a terminé ?
  ⇒ **mécanisme d'interruptions**

Historique – Télétype

Crédit photo : AlisonW – CC BY SA

Ceci est une révolution !

- changement fondamental dans l'interaction avec l'ordinateur
- plus besoin d'attendre le passage du bac de cartes perforées sur l'ordinateur
- commandes tapées au clavier
  ⇒ langage de commandes
- résultat directement imprimé

Historique – Télétype

- c'est un périphérique comme un autre...
- ... mais l'interaction modifie le besoin
- le superviseur lance des actions suite à des commandes tapées à la console
- usage « interactif » $\neq$ usage « batch »

Télétype en action :
`https://www.youtube.com/watch?v=X9ctLFYSDfQ`

Historique – Temps partagé

Connecter plusieurs terminaux à un même ordinateur :

- accueillir plusieurs utilisateurs
- mieux rentabiliser le coût d'un ordinateur
- connexion via une liaison série directe, ou via un modem
- donner à chacun l'impression d'avoir « son » ordinateur
- systèmes à temps partagé

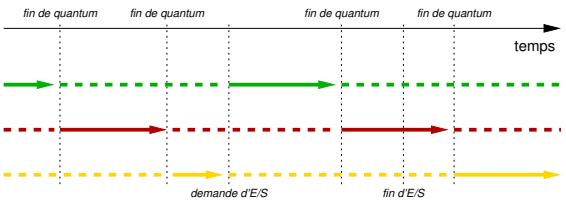
Exemples :

- CTSS (Compatible Time-Sharing System) : MIT (1961)
  IBM 7094 modifié par IBM, 32 utilisateurs maximum
- STSS (Stanford Time-Sharing System) : Stanford (1963)
  DEC PDP-1, 12 terminaux

Historique – Temps partagé

Allouer le processeur pendant des petites portions de temps à chaque utilisateur :

- quantum : durée fixée par le système (ex: 20 ms)
- pour un être humain, les programmes semblent tourner en parallèle



Historique – Temps partagé

Questions posées par le temps partagé :

- comment reprendre le contrôle à la fin de quantum ?
  $\Rightarrow$ mécanisme d'**horloge** avec interruption
- comment gérer plusieurs utilisateurs ?
  $\Rightarrow$ identité : validation et autorisations
  $\Rightarrow$ mécanismes logiciels
- comment éviter qu'un utilisateur outrepasse ses droits ?
  $\Rightarrow$ **modes d'exécution** privilégié et non privilégié
  $\Rightarrow$ quelques instructions interdites en mode non privilégié

  Exemple :
  - accès au disque : réservé au mode privilégié
  - programmes utilisateurs : exécutés en mode non privilégié
    $\Rightarrow$ passage par le système pour accéder aux fichiers

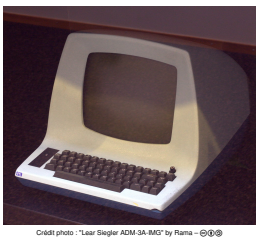
Historique – Appel système

Lorsqu'un programme souhaite effectuer (par exemple) un accès disque, il fait un **appel système** :

- instruction spéciale (TRAP, SVC, INT, etc. suivant le processeur)
- provoque (entre autres) :
  - basculement en mode privilégié
  - déroutement du programme vers une adresse spécifique
- adresse spécifique = système d'exploitation
- **vérification** des paramètres, des droits, etc.
  $\Rightarrow$ pour éviter qu'un utilisateur outrepasse ses droits
- réalisation de l'action demandée par l'appel système

Historique – Temps partagé

Début des années 1970 : terminaux à écran cathodique



Âge d'or des « mainframes »
$\Rightarrow$ stabilisation des systèmes d'exploitation autour des concepts vus précédemment

Bilan

Fonctionnalités offertes par un système d'exploitation

- rentabiliser l'usage de l'ordinateur
  ⇒ utiliser tous les temps morts
- partager un ordinateur entre plusieurs utilisateurs
  ⇒ partager équitablement les ressources matérielles
- faciliter l'accès aux périphériques
  ⇒ offrir une interface avec le matériel
  ⇒ pour tirer parti du parallélisme des périphériques
- permettre à des utilisateurs d'éditer, développer, lancer des programmes

... le tout en offrant les garanties de sécurité nécessaires

Plan

Continuons l'historique

Systèmes d'exploitation marquants :

- Burroughs MCP (1961)
  - conçu pour l'ordinateur Burroughs B5000
  - premier SE programmé en **langage de haut niveau**
  - beaucoup d'éléments innovants (multiprocesseurs, mémoire virtuelle)
- IBM OS/360 (1964)
  - SE « universel » pour les ordinateurs IBM System/360
  - énorme (des dizaines de millions de lignes d'assembleur)
  - **complexe**, beaucoup de retards
  - F. Brooks « The Mythical Man-Month »
- CTSS (**MIT**, 1961)
  - premier système multi-utilisateurs
  - IBM 7094 modifié (2 exemplaires)

Multics

- Multics (MULtiplexed Information and Computing Service)

1963	spécification (MIT)
1964	démarrage du projet
1964	General Electric + Bell Labs rejoignent le MIT
1968	première version « self-hosting »
1969	**Bell Labs** se retirent du projet
1973	annonce commerciale
1985	fin du développement
2000	fermeture du dernier site (Min Défense Canada)

  - Système très ambitieux
    - écrit en langage de haut niveau (PL/1)
    - accès uniforme à la mémoire et aux fichiers
    - système de fichiers hiérarchique
    - édition de liens dynamique
    - reconfiguration matérielle dynamique
    - processus « démons »
    - sécurité (certification B2 en 1985)
  - 84 sites au total (dont 31 universités/centres français)

Unix

- Unix

1969	Bell Labs se retirent du projet Multics récupération d'un PDP-7, écriture en assembleur
1971	achat PDP-11/20 en échange d'un formateur de texte
1972	création du langage C et réécriture d'Unix en C
1973	premières diffusions
1979	Unix v7 (environ 10 000 lignes de C et un peu d'assembleur)
1980	distribution 4BSD
1992	procès AT&T vs BSDi vs Berkeley (fin en 1994)

  - Beaucoup d'idées novatrices
    - simple
    - écrit en langage C
    - séparation nette **noyau** / utilitaires (ou applications)
    - tout est fichier
    - pas de structure interne des fichiers
    - interface utilisateur simple (minimaliste)
    - philosophie Unix (kiss : « keep it simple, stupid »)
    - portable

Le noyau

Philosophie Unix ⇒ approche minimaliste, à l'opposé des systèmes d'exploitation précédents

Le noyau a deux objectifs fondamentaux :

- **partager équitablement les ressources**
- **garantir la sécurité des données**

Le noyau

Partager équitablement les ressources :

- mémoire vive
- temps processeur
- carte réseau
- espace disque
- accès aux disques
- etc.

Le noyau

Garantir la sécurité des données

- un processus ne doit pas accéder aux données d'un autre processus (sauf si autorisé)
- un utilisateur ne doit pas accéder à un fichier non autorisé
- un utilisateur ne doit pas terminer un processus d'un autre utilisateur
- etc.

Le noyau

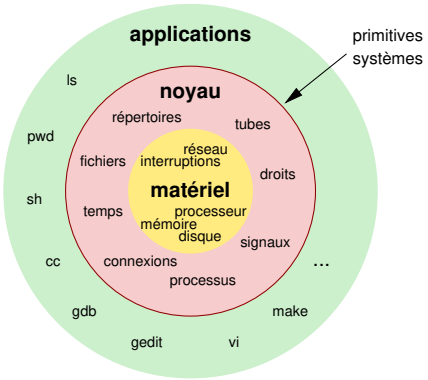
Pour atteindre les deux objectifs fondamentaux :

- le **noyau** s'exécute en **mode privilégié**
  ⇒ il a accès à toutes les ressources matérielles
- les programmes (applications) s'exécutent, dans le contexte de **processus**, en **mode non privilégié**
  ⇒ appels au noyau pour exécuter certaines opérations

Les appels au noyau sont les **primitives systèmes**

- Interface de programmation du noyau
  Application Programming Interface (API)
- Forme : appels de fonction en C
- Exemple :
  ```
  int open (const char *path, int flag, mode_t mode)
  ```


Le noyau



Le noyau

Définition :

Le noyau est constitué de l'ensemble minimum des primitives systèmes nécessaires pour atteindre les deux objectifs fondamentaux

Le noyau

- Démarrage de l'ordinateur : noyau installé en mémoire
  ⇒ il y restera jusqu'à la fin (redémarrage, arrêt, ou crash)
- Le noyau s'exécute en mode privilégié
  - code sensible
  - difficile à développer, à mettre au point
  - plus il est petit, mieux c'est
  - tout ce qui peut être mis ailleurs doit l'être

  Exemple :
  - pour le noyau, un utilisateur = un nombre entier
  - `ls -l` affiche des noms de login
  - ⇒ la conversion « entier ↔ nom » est effectuée par `ls`
- Mise en évidence du concept de noyau
  ⇒ apport majeur d'Unix

Le noyau

Quelles fonctions doivent être des primitives systèmes ?

- un processus ne doit pas avoir le droit d'accéder directement au disque (violation du deuxième objectif)
  ⇒ il faut que le noyau présente une abstraction avec des droits : système de fichiers, propriétaire, droits d'accès
  ⇒ accès au disque par l'intermédiaire de cette abstraction
  ⇒ `open` est une primitive système
- lire l'heure courante ⇒ accès à un périphérique
  ⇒ `time` est une primitive système

Le noyau

Quelles fonctions doivent être des primitives systèmes ?

- `strlen` calcule un nombre d'octets en lisant dans la mémoire du processus courant (donc autorisé)
  ⇒ `strlen` n'est donc pas une primitive système
  Comme elle est utile, on la met [1] dans une **bibliothèque** une fois pour toutes, pour éviter d'avoir à la reprogrammer à chaque fois
- `getlogin` renvoie le nom de login de l'utilisateur
  ⇒ cette fonction récupère le numéro de l'utilisateur avec une primitive système, puis convertit ce numéro en nom en parcourant le fichier `/etc/passwd` (ou en utilisant une base externe LDAP dans le cas de turing)
  ⇒ `getlogin` n'est pas une primitive système

1. Ou plus exactement son code compilé.

Interface utilisateur

Date	Stations de travail	Ordinateurs personnels
1968	Prototype de souris par Douglas Englebart (Stanfort Research Institute)	
1973	Alto (Xerox)	
1981	Display Manager (Apollo Computer)	
1983	W Window System (Stanford)	Lisa (Apple)
1984	X Window System (MIT)	
1985	HP Windows (HP),	Amiga Workbench (Commodore),
	Sunview (Sun)	Windows (Microsoft), etc

La gestion d'une interface utilisateur « graphique » n'est pas du ressort du noyau :

- Souris, écran graphique, écran tactile (tablette, etc.)
  ⇒ périphériques gérés par le noyau
- Système de fenêtrage
  ⇒ applications

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

Retour sur notre interrogation du début :

- Unix (l'original, puis les *BSD) : noyau + applications
- Linux est un noyau
  - des organisations (bénévoles ou commerciales) ont réuni des applications, les ont compilées et ont diffusé, avec le noyau, des **distributions** prêtes à l'emploi
- Windows : noyau + applications
- Android, iOS : noyau (avec support de périphériques tactiles, audio, GSM, GPS, etc.) + applications
- box Internet, montre connectée, voiture, etc. : noyau + applications spécifiques
- etc.

Quelques systèmes atypiques

- systèmes temps réel
- systèmes contraints (ex: Internet des Objets)

Bilan

- Depuis Unix : distinction entre noyau et reste du système d'exploitation
- Objet de cette UE : comprendre le fonctionnement d'un noyau de système d'exploitation

Plan

Interface des primitives systèmes

Primitives systèmes = ensemble de fonctions (appelables en C) pour accéder aux services offerts par le noyau

- À l'origine (Unix v6, 1975) : 43 primitives
- Évolutions ultérieures (années 1980) :
  - Bell Labs, AT&T
  - U. de Berkeley
  - Essor commercial
- Résultat
  - de nombreuses divergences ⇒ incompatibilités
  - les programmes ne sont plus portables
  - les utilisateurs ne sont pas contents
- Il faut normaliser l'existant ⇒ POSIX

POSIX

Comité POSIX de l'IEEE

- IEEE = Institute of Electrical and Electronics Engineers (association américaine, rédige des textes normatifs)
- POSIX = Portable Operating System Interface
- Norme IEEE : IEEE Std 1003.*
- Norme internationale : ISO/IEC 9945
- Première version : 1988
- Version actuelle : 2013

Impact majeur ⇒ aucun système d'exploitation ne peut se permettre de ne pas être compatible POSIX

POSIX

POSIX normalise beaucoup de choses :

- des primitives systèmes et des fonctions de bibliothèque
- des commandes (`sh`, `ls`, `tr`, etc.)
- des extensions (temps réel, threads, sémaphores, etc.)

POSIX ne normalise pas tout :

- une implémentation peut avoir ses propres extensions ⇒ non normalisées ⇒ non portables
- Exemple avec `ls` :
  - POSIX 2013 : 23 options (c'est beaucoup...)
  - GNU (Linux) : 58 options (c'est trop !)
- programmer portable ⇒ respecter POSIX
- utiliser la notice fournie en cours

POSIX

POSIX ne fait pas de distinction entre primitive système et fonction de bibliothèque :

- laisser de la liberté pour les implémentations
- distinction parfois mince
  - Exemple : 6 fonctions pour exécuter un programme (`execv`, `execl`, `execvp`, `execlp`, `execve` et `execle`), une seule est une primitive, les 5 autres sont des fonctions de bibliothèque

POSIX

Principes généraux

- Types POSIX
- Constantes
- Gestion des erreurs
- Paramètres de type « pointeur »

POSIX – Types

- Le noyau Unix manipulait des objets grâce à des entiers
  Problème : portabilité (16/32/64 bits ? implémentations ?)
- POSIX a introduit de nouveaux types (avec `typedef`)
  ⇒ masquer le détail d'implémentation
  ⇒ portabilité des programmes
  Quelques exemples :

`uid_t`	numéro d'utilisateur
`gid_t`	numéro de groupe
`pid_t`	numéro de processus
`mode_t`	permissions
`time_t`	heure courante
`size_t`	taille, résultat de sizeof (définie par ISO C)

POSIX – Types

- Définition des types :
  - taille choisie par l'implémenteur
  - voir fichiers d'inclusion
- Problème : comment afficher des valeurs avec `printf` ?
  - `%d`, `%ld`, `%lld` ? ⇒ non !
  - solution : convertir en `uintmax_t` et utiliser le format `%ju`
    - `j` : taille = celle d'un `intmax_t`/`uintmax_t`
    - `u` : `unsigned` (`d` pour `signed`)
  - exemple :
    ```
    printf ("taille=%ju", (uintmax_t) stbuf.st_size)
    ```


POSIX – Constantes

- Certaines primitives nécessitent de spécifier une opération
- Exemple :

`r = access ("toto", 0)`	le fichier existe-t'il ?
`r = access ("toto", 1)`	puis-je exécuter le fichier ?
`r = access ("toto", 2)`	puis-je modifier le fichier ?
`r = access ("toto", 4)`	puis-je lire le fichier ?

- Il faut se rappeler des valeurs et de leur signification

POSIX – Constantes

- Définition de constantes (dans les fichiers d'inclusion)
  Fichier `unistd.h` :

```
#define R_OK   4   /* Test for read permission */
#define W_OK   2   /* Test for write permission */
#define X_OK   1   /* Test for execute permission */
#define F_OK   0   /* Test for existence */
```

- L'exemple précédent devient :

`r = access ("toto", F_OK)`	le fichier existe-t'il ?
`r = access ("toto", X_OK)`	puis-je exécuter le fichier ?
`r = access ("toto", W_OK)`	puis-je modifier le fichier ?
`r = access ("toto", R_OK)`	puis-je lire le fichier ?

- Utiliser les constantes rend les programmes plus lisibles
- Parfois, les constantes sont moins pratiques (permissions) ou peu répandues

POSIX – Constantes

Certaines constantes ne sont pas constantes...

- arguments sémantiques des primitives : fichiers d'inclusion

Exemples :

`R_OK`	test en lecture pour `access`	`unicode.h`
`O_WRONLY`	ouverture en écriture pour `open`	`fcntl.h`
`S_IFDIR`	type de fichier « répertoire »	`sys/stat.h`
...		

- limites arbitraires : `limits.h`

Exemples :

`PATH_MAX`	taille maximum d'un chemin complet
`LOGIN_NAME_MAX`	taille maximum d'un nom de login
`CHILD_MAX`	nombre maximum de processus fils

- limites dépendant de la configuration du système
  `long sysconf (int paramètre)`

Exemples :

`_SC_LOGIN_NAME_MAX`	longueur maxium des noms d'utilisateur
`_SC_OPEN_MAX`	nombre maximum d'ouvertures de fichier
`_SC_CHILD_MAX`	nombre maximum de processus fils

- limites dépendant du système de fichiers
  `long pathconf (const char *chemin, int paramètre)`

Exemples :

`_PC_LINK_MAX`	nombre maximum de liens sur un fichier
`_PC_NAME_MAX`	taille maximum d'un nom de fichier
`_PC_PATH_MAX`	taille maximum d'un chemin complet

POSIX – Gestion des erreurs

En cas d'erreur, les primitives :

- renvoient -1 en cas d'erreur
  - ... sauf pour quelques très rares exceptions
- placent dans la variable `errno` un code reflétant l'erreur

Fichier `errno.h` :

```
#define EPERM    1    /* Operation not permitted */
#define ENOENT   2    /* No such file or directory */
#define ESRCH    3    /* No such process */
...
extern int errno ;
```

Description de toutes les erreurs possibles pour une primitive
⇒ consulter POSIX ou le `man` de la primitive

POSIX – Gestion des erreurs

Exemple d'utilisation (laborieux) :

```
int fd ;

fd = open ("toto", O_RDONLY) ;
if (fd == -1)
{
    switch (errno)
    {
        case EPERM :
            fprintf (stderr, "Operation␣not␣permitted\n") ;
            break ;
        case ENOENT :
            fprintf (stderr, "No␣such␣file␣or␣directory\n") ;
            break ;
        ...
    }
    exit (1) ;
}
```


POSIX – Gestion des erreurs

Mieux : mettre l'affichage dans une fonction

```
void perror (const char *msg)
{
    fprintf (stderr, "%s:␣", msg) ;
    switch (errno)
    {
        case EPERM :
            fprintf (stderr, "Operation␣not␣permitted\n") ;
            break ;
        case ENOENT :
            fprintf (stderr, "No␣such␣file␣or␣directory\n") ;
            break ;
        ...
    }
}
```

Très utile ⇒ bibliothèque standard :

- `void perror (const char *msg) ;`
- `char *strerror (int numerr) ;`

POSIX – Gestion des erreurs

Exemple d'utilisation (final) :

```
int fd ;

fd = open ("toto", O_RDONLY) ;
if (fd == -1)
{
    perror ("open␣toto") ;
    exit (1) ;
}
```


POSIX – Gestion des erreurs

Recommandations

- toujours vérifier les retours de primitives
- afficher la raison des erreurs

```
void raler (char *msg)
{
    perror (msg) ;
    exit (1) ;
}

...

fd = open ("toto", O_RDONLY) ;
if (fd == -1)
    raler ("open␣toto") ;

...
```


POSIX – Paramètres de type « pointeur »

Certaines primitives retournent des résultats plus complexes qu'un simple entier

- Paramètre de type pointeur (sur un objet à remplir)
- Exemples :
  - `int stat (const char *path, struct stat *stbuf)`
    Retourne dans l'emplacement repéré par `stbuf` les attributs du fichier
  - `pid_t wait (int *status)`
    Place dans l'emplacement repéré par `status` des informations sur la terminaison du processus
  - etc.
- L'emplacement mémoire pointé **doit exister** !

POSIX – Paramètres de type « pointeur »

Exemples d'utilisation :

Correct	Faux

`struct stat stbuf ;`
`if (stat ("toto", &stbuf) == -1)`
`  ...`
	`struct stat *stbuf ;`
	`if (stat ("toto", stbuf) == -1)`
	`  ...`

La variable `stbuf` existe, on passe son adresse	La variable `stbuf` est un pointeur non initialisé, le noyau va écrire le résultat quelque part (où ?) en mémoire

Plan

Tracer les primitives systèmes

Sur de nombreux systèmes, il est possible de suivre les primitives systèmes exécutées par un processus « à la trace »

Commande non normalisée par POSIX :

Linux	commande `strace`
FreeBSD	commande `truss`

Tracer les primitives systèmes

Exemple sur Linux :

```
$ strace cp README test
execve("/bin/cp", ["cp", "README", "test"], 0x7ffd01453380) = 0
[...]
open("README", O_RDONLY)    = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=445, ...}) = 0
open("test", O_WRONLY|O_CREAT|O_EXCL, 0644) = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
read(3, "Lorem␣ipsum␣dolor␣sit␣amet,␣cons"..., 131072) = 445
write(4, "Lorem␣ipsum␣dolor␣sit␣amet,␣cons"..., 445) = 445
read(3, "", 131072)                     = 0
close(4)                                = 0
close(3)                                = 0
[...]
+++ exited with 0 +++
```

Exemple simplifié, sans l'initialisation et la terminaison du processus

Tracer les primitives systèmes

Intérêts de `strace` (ou équivalent) :

- découvrir les primitives
- comprendre le fonctionnement des commandes du système
  - ... si elles sont simples
  - pas la peine d'essayer `cc` ou `doxygen`...
- distinguer les primitives des fonctions de bibliothèque
  - `strace` = espion placé à la frontière du noyau
  - les fonctions de bibliothèque ne sont donc pas affichées
- déboguer ses propres programmes
  - « tiens, c'est curieux, mon programme se termine avant d'arriver à l'appel de la primitive truc »

N'hésitez pas à utiliser cette commande `strace` !

Plan

Plan

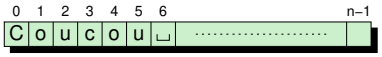
Accès aux fichiers

Un fichier :

- a un nom (en fait, plusieurs...)
- est accessible via un chemin (absolu, relatif)
- possède des attributs :
  - type
  - propriétaire, groupe, permissions
  - taille
  - dates
    - de dernière modification des données
    - date de dernière modification des attributs
    - date de dernier accès
  - emplacement des données sur le disque
- 2 types de fichiers
  - fichiers « réguliers »
  - répertoires
  - ... en réalité, il y en a d'autres (plus tard...)

Accès aux fichiers

Un fichier a une structure simple : suite linéaire d'octets



- innovation d'Unix
  - dans les systèmes antérieurs : les fichiers avaient un type (texte, base de données, etc.)
  - pas de type ⇒ simplification du noyau
- la structure dépend de l'application qui accède au fichier
  - texte : suite de caractères séparés par `\n` (octet 10)
  - binaire exécutable : contient un en-tête qui décrit les différentes parties (code, données, infos de debug, etc.)
  - document LibreOffice : cf application LibreOffice
  - etc.
- notion de « *magic number* »
  - suite d'octets au début d'un fichier pour l'identifier
  - ex: `#!` (script), `%PDF` (fichier PDF), `0xffd8` (JPEG), etc.
  - commande `file`

Accès aux fichiers

- nom de fichier : aucune signification pour le noyau
  - je peux appeler mon exécutable `toto.titi.tata` si j'en ai envie
  - je peux appeler un fichier texte `toto.xls` si j'en ai envie
  - certaines applications attendent un suffixe
    - ex : l'application « compilateur C » suppose que les sources C finissent par « `.c` »
    - ce n'est pas le cas général

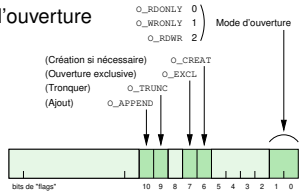
Ouverture de fichier

```
int open(const char *path, int flags)
int open(const char *path, int flags, mode_t mode)
```

- 2 formes pour cette primitive (exception qui confirme...)
  - première forme : ouverture « simple »
  - deuxième forme : ouverture avec création du fichier
    ⇒ `mode` est la permission initiale du fichier (avec application du masque de création, voir plus tard)
- résultat : descripteur d'ouverture (ou -1)
  - utilisé par les autres primitives

Ouverture de fichier

- `flags` : mode d'ouverture



Position des bits purement imaginaire...

- Exemples :
  - `open("toto", O_RDONLY)`
    Ouverture en lecture seule (le fichier doit exister)
  - `open("toto", O_WRONLY | O_CREAT | O_TRUNC, 0666)`
    Création d'un nouveau fichier (ou remise à 0 d'un fichier existant) et ouverture en écriture seule
  - `open("toto", O_RDWR | O_CREAT | O_APPEND, 0666)`
    Création d'un nouveau fichier (s'il n'existait pas déjà) et ouverture en lecture/écriture avec ajout à la fin

Ouverture de fichier

- Quelles permissions mettre lors d'une création ?
  - règle de base : `mode = 0666`
    - si pas de contrainte de sécurité particulière
    - si pas exécutable (sauf si vous écriviez un éditeur de liens)
  - laisser faire le masque de création de fichiers (plus tard)
- Attention : si `mode` non fourni, `open` prendra ce qu'il y a sur la pile à l'endroit attendu
  ⇒ vraisemblablement n'importe quoi

Ouverture de fichier

Trois ouvertures par défaut :

0	entrée standard
1	sortie standard
2	sortie d'erreur standard

Ces descripteurs sont ouverts par le Shell

- Par défaut : le terminal
- Redirection possible depuis/vers un fichier
  ⇒ ne jamais supposer que l'entrée standard est forcément le clavier (ou la sortie standard l'écran)

Fermeture de fichier

Ne pas oublier de fermer les fichiers après utilisation

```
int close(int fd)
```

- fermeture automatique à la terminaison du processus
- bonne pratique : fermer dès que possible
- plus tard (tubes) : fermer dès que possible est **crucial**...
- autant s'habituer à le faire dès maintenant !

Accès au fichier

Une fois le fichier ouvert, on peut lire et écrire :

```
ssize_t read(int fd, void *buf, size_t nb)
ssize_t write(int fd, const void *buf, size_t nb)
```

- retourne le nombre d'octets transférés (ou -1)
  - 0 en fin de fichier
- `fd` : descripteur d'ouverture (retourné par `open`)
- `buf` : emplacement où le noyau écrit (pour `read`) ou lit (pour `write`) les données à transférer
- `nb` : nombre d'octets à transférer
  - nb d'octets transférés : pas forcément celui demandé
  - exemple : `read(fd, buf, 500)` alors que le fichier ne fait que 10 octets ⇒ retour = 10
  - exemple : `write(fd, buf, 500)` alors que le disque est à 10 octets de la saturation ⇒ retour = 10

Accès au fichier

Accès aléatoire :

```
off_t lseek(int fd, off_t offset, int apartir)
```

- Chaque ouverture de fichier possède un *offset*
  - position courante dans le fichier (≥ 0)
  - avancée automatiquement avec `read` et `write`
- `lseek` permet de modifier l'offset en fonction de `apartir`

`SEEK_SET`	déplacer à la position absolue
`SEEK_CUR`	avancer à partir de la position actuelle
`SEEK_END`	avancer à partir de la fin du fichier

- code de retour de `lseek` : offset après modification

Accès au fichier

Exemples :

- `lseek(fd, 1000000, SEEK_SET)`
  Se déplacer à l'offset = 1 million d'octets
- `lseek(fd, -20, SEEK_CUR)`
  Revenir en arrière de 20 octets avant la position actuelle
- `lseek(fd, 300000, SEEK_END)`
  Se déplacer 300 000 octets après la fin actuelle du fichier
  - `read` renverra alors 0
  - `write` pourra écrire de nouveaux octets. Dans ce cas :
    - le système laisse un « trou » dans le fichier
    - en cas de lecture dans le « trou », tout se passe comme si on avait écrit 300 000 fois l'octet 0
    - la taille du fichier n'est pas la place occupée sur le disque !
- `lseek(fd, 0, SEEK_CUR)`
  Où suis-je ?

Plan

Primitives et fonctions de bibliothèque

Deux séries de fonctions pour accéder aux fichiers ?

- primitive systèmes : `open`, `close`, `read`, `write`, `lseek`
- fonctions de bibliothèque : `fopen`, `fclose`, `getc`, `scanf`, `fread`, `putc`, `printf`, `fwrite`, `fseek`, etc.

Duplication de fonctionnalités ?

Pourquoi ?

Exemple avec primitives systèmes

```
int fd1, fd2;
char c;
ssize_t n;

if ((fd1 = open("toto", O_RDONLY)) == -1)
        raler("open toto");

if ((fd2 = open("titi", O_WRONLY | O_CREAT | O_TRUNC, 0666)) == -1)
        raler("open titi");

while ((n = read(fd1, &c, 1)) > 0) {
        if (write(fd2, &c, 1) == -1)
                raler("write");
}

if (n == -1)
        raler("read");

if (close(fd1) == -1)
        raler("close toto");

if (close(fd2) == -1)
        raler("close titi");
```


Exemple avec fonctions de bibliothèque

```
FILE *fp1, *fp2;
int c;

if ((fp1 = fopen("toto", "r")) == NULL)
        raler("fopen toto");

if ((fp2 = fopen("titi", "w")) == NULL)
        raler("fopen titi");

while ((c = getc(fp1)) != EOF) {
        if (putc(c, fp2) == EOF)
                raler("putc");
}

if (ferror(fp1))
        raler("getc");

if (fclose(fp1) == -1)
        raler("fclose toto");

if (fclose(fp2) == -1)
        raler("fclose titi");
```


Primitives et fonctions de bibliothèque

Jeu des différences

primitives systèmes	fonctions de bibliothèque
descripteur = `int`	descripteur = `FILE *`
paramètres de `read`/`write` moins simples	utilisation de `getc`/`putc` simple
uniquement `read` ou `write`	possibilité d'utilisation d'autres fonctions (`printf`, `puts`, etc.)
but = sécuriser les données	but = aide à la programmation
interface de plus bas niveau	les fonctions de bibliothèque utilisent les primitives système (⇒ haut niveau)
code dans le noyau	code ajouté au programme lors de l'édition de liens

Efficacité

Qu'est-ce qui est le plus efficace ?

- approche naïve : les fonctions de bibliothèque appelant les primitives systèmes « équivalentes », elles sont plus lentes
- la réponse est plus complexe...

Efficacité

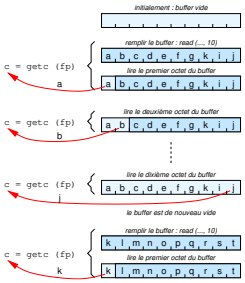
Rappel : déroulement d'une primitive système (exemple `read`)

- instruction spéciale (TRAP, SVC, INT, etc. suivant le processeur)
- provoque (entre autres) :
  - basculement en mode privilégié
  - déroutement du programme vers une adresse spécifique
- vérifications
  - le descripteur d'ouverture est-il ouvert ?
  - l'adresse du buffer est-elle valide ?
  - l'adresse de la fin du buffer est-elle valide ?
  - le nombre à transférer est-il valide ?
- faire l'entrée/sortie (logique)
- recopier les données dans l'espace mémoire du processus
- revenir à l'instruction suivant `read`

Bilan : beaucoup de vérifications, **surtout pour un seul octet**

Bufferisation

Les fonctions d'entrées/sorties de la bibliothèque font de la « **bufferisation** »



- buffer = tableau en mémoire
- premier appel à `getc` : remplissage du buffer
  ⇒ appel à `read` ⇒ vérifications
- après : simple appel de fonction + lecture en mémoire
  ⇒ très efficace
- buffer de taille n ⇒ appel à `read` une fois sur n
- en pratique, $n = 4096$ (p. ex.)

Bufferisation

Bilan :

- si lecture de peu d'octets, les fonctions d'entrées/sorties de la bibliothèque sont plus efficaces que les primitives systèmes
  - moins de vérifications
  - davantage d'appels de fonctions simples
- si lecture de beaucoup d'octets à la fois, les primitives systèmes sont plus efficaces que les fonctions de bibliothèque
  - pas d'overhead dû à une surcouche
  - pas de temps passé pour une bufferisation superflue

À partir de quelle taille de lecture les primitives systèmes sont-elles moins efficaces que `getc` ? ⇒ exercice

Plan

Attributs des fichiers

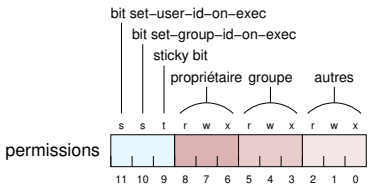
À chaque fichier sont associés des attributs :

- type
- propriétaire et groupe
- permissions
- taille
- dates
  - de dernière modification des données
  - de dernière modification des attributs
  - de dernier accès
- nombre de liens (voir plus tard)
- numéro de périphérique, numéro de fichier
- emplacement des données sur le disque

Le nom n'est pas un attribut du fichier (voir plus tard)

Permissions

Permissions : 12 bits



- 9 bits habituels (3 bits pour propriétaire, groupe, autres)
- bit « sticky » : pour les répertoires, interdit la suppression des fichiers qui s'y trouvent, sauf pour le propriétaire du fichier ou du répertoire (utile pour `/tmp` par exemple)
- bits « set-user-id-on-exec » et « set-group-id-on-exec »
  ⇒ plus tard (chapitre sur la gestion des processus)

Consultation des attributs

Récupération de tous les attributs en une seule opération :

```
int stat(const char *path, struct stat *stbuf)
int fstat(int fd, struct stat *stbuf)
```

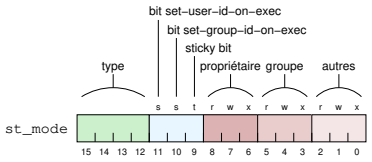
- la structure `stat` contient, en retour, l'ensemble des attributs, parmi lesquels :

`st_mode`	type et permissions
`st_uid`	propriétaire
`st_gid`	groupe
`st_size`	taille en octets
`st_atime`	date de dernier accès
`st_mtime`	date de dernière modification des données
`st_ctime`	date de dernière modification des attributs

- d'autres attributs sont dans cette structure
- ... mais pas tous (localisation des données sur le disque inutile hors du noyau ⇒ pas remontée)

Consultation des attributs

`st_mode` : type et permissions dans le même champ ?



- 12 bits de permissions (ex: `00752` = `rwxr-x-w-`)
  - POSIX définit des constantes :
    `S_IRWXU | S_IRGRP | S_IXGRP | S_IWOTH`
  - ... mais POSIX définit aussi les valeurs numériques (très rare... elles sont plus pratiques à utiliser)
- 4 bits : type

1	répertoire
2	fichier régulier
...	...

Exemple (valeurs imaginaires)

Consultation des attributs

Comment utiliser `st_mode` ?

- Récupérer les permissions : `stbuf.st_mode & 0777`
- Récupérer le type : `stbuf.st_mode & 0xf000`
  - pour tester...
    ... si répertoire `if ((stbuf.st_mode & 0xf000) == 0x1000)`
    ... si fichier `if ((stbuf.st_mode & 0xf000) == 0x2000)`
  - Utilisation des constantes POSIX
    ... si répertoire `if ((stbuf.st_mode & S_IFMT) == S_IFDIR)`
    ... si fichier `if ((stbuf.st_mode & S_IFMT) == S_IFREG)`
  - Encore mieux...
    ... si répertoire `if (S_ISDIR(stbuf.st_mode))`
    ... si fichier `if (S_ISREG(stbuf.st_mode))`

Consultation des attributs

Comment répondre à « puis-je lire/écrire/exécuter le fichier » ?

- `stat` ne permet pas de répondre à la question
  - tester les permissions ne suffit pas
  - il faut d'abord savoir si on est le propriétaire, un membre du groupe, ou un « autre »
- solution :

```
int access(const char *path, int mode)
```

- le paramètre `mode` vaut :

`F_OK`	teste si le fichier existe
`X_OK`	teste l'accès en exécution
`W_OK`	teste l'accès en écriture
`R_OK`	teste l'accès en lecture

- accès autorisé : retour = 0, interdit : retour = -1

Modification des attributs

Pour modifier certains attributs :

- modifier les permissions

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

- modifier le propriétaire ou le groupe

```
int chown(const char *path, uid_t uid, git_t gid)
int fchown(int fd, uid_t uid, git_t gid)
```


Modification des attributs

- modifier les dates :

```
int utime(const char *path, const struct utimbuf *buf)
```

  - Champs de `struct utimbuf` :

`actime`	dernier accès
`modtime`	dernière modification

  - pas de troisième date (modification des attributs)
    ⇒ modifiée par `utime` elle-même

Plan

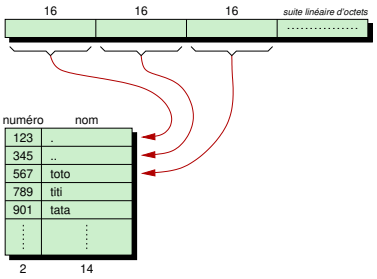
Qu'est-ce qu'un répertoire ?

Un répertoire est un fichier sur le disque

- type particulier : ce n'est pas un fichier régulier
  - certaines opérations sont interdites
    Exemple : `open("repertoire", O_WRONLY)`
  - ⇒ utilisation de primitives spécialisées
- structure particulière
  - c'est toujours une suite linéaire d'octets
  - mais le noyau y met sa propre structure
- contenu : des références à des fichiers (réguliers ou répertoires ou ...)

Qu'est-ce qu'un répertoire

Structure originelle des répertoires Unix (V7, 1977) :



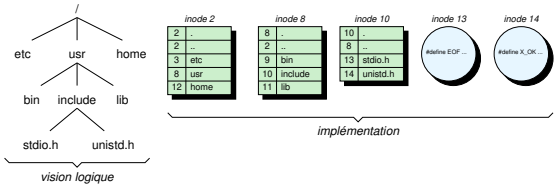
- nom : limité à 14 caractères
- numéro d'inode : numéro de fichier sur le disque
- 2 entrées spéciales : « . » et « .. »

Numéro d'inode

Qu'est-ce qu'un numéro de fichier ?

- chaque fichier (régulier, répertoire) a un **inode**
- l'inode rassemble les attributs d'un fichier
  (≈ ce que retourne `stat` + localisation des données)
- inodes rangés séquentiellement sur une portion du disque
  ⇒ un inode est donc repéré par son numéro
  - convention : inode du répertoire racine = 2
- un numéro d'inode identifie donc un fichier
  ⇒ attributs et contenu

Qu'est-ce qu'un répertoire



- vision logique : arborescence
- vision physique : série d'inodes sur le disque
  ⇒ répertoires et fichiers
- le nom ne fait pas partie des attributs de fichier
  ⇒ un nom n'est qu'une entrée dans un répertoire
- `ls -i` : affiche les entrées (nom + numéro d'inode)

Qu'est-ce qu'un répertoire

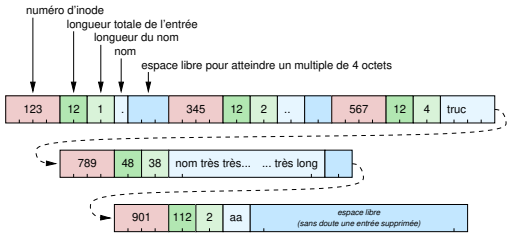
Principales opérations

- `open("/usr/include/stdio.h", ...)`
  - recherche dans les répertoires
  - « traversée de répertoires »
  - nécessite le droit d'**exécution** sur les répertoires
  - chemin absolu : commencer à l'inode 2
  - chemin relatif : commencer à l'inode du répertoire courant
- supprimer une entrée
  - vider l'entrée : numéro d'inode ← 0
  - le reste du répertoire n'est pas décalé (performances)
- ajouter une entrée
  - rechercher une entrée disponible
  - ou agrandir le répertoire si nécessaire

Qu'est-ce qu'un répertoire

1982 : U. Berkeley diffuse BSD 4.2

- amélioration du système de fichiers : performances, fonctionnalités
- augmentation de la taille maximum des noms (≤ 255)



Lecture d'un répertoire

Comment lire le contenu d'un répertoire ?

- originellement : `open("rep", O_RDONLY)`
  - lire 2 octets pour l'inode et 14 pour le nom, etc.
  - ignorer les numéros d'inode nuls
  - recommencer jusqu'à la fin
- apparition du système de fichiers BSD
  - changer tous les programmes (ex : `ls`)
- nouvelle primitive système BSD : `getdirentries`
  - non normalisée par POSIX ⇒ ne pas utiliser
- BSD propose de nouvelles fonctions de bibliothèque :
  - `opendir`, `readdir`, `closedir`
  - normalisées par POSIX ⇒ **à utiliser** !
- nouveaux systèmes de fichiers
  - locaux : lfs, unionfs, zfs, ext*n*fs, etc
  - réseau : nfs, smbfs, etc.

⇒ `opendir` & co : toujours utilisables

Lecture d'un répertoire

```
DIR *opendir(const char *path)
struct dirent *readdir(DIR *dp)
int closedir(DIR *dp)
```

- champs de la `struct dirent` normalisés par POSIX :

`d_ino`	numéro d'inode
`d_name`	nom, terminé par un octet nul

- attention : il peut y avoir d'autres champs, mais ils ne sont pas normalisés par POSIX (ex : `d_type`)
  ⇒ surtout ne pas utiliser !
- `readdir` renvoie successivement toutes les entrées
  - y compris « . » et « .. »
  - adresse renvoyée = variable locale `static` de `readdir`
    ⇒ pas besoin de la libérer

Lecture d'un répertoire

Exemple :

```
DIR *dp;
struct dirent *d;

dp = opendir("/tmp");
if (dp == NULL)
        raler("opendir");

while ((d = readdir(dp)) != NULL) {
        if (strcmp(d->d_name, ".")  != 0 &&
            strcmp(d->d_name, "..") != 0)
                printf("%ld␣%s\n", d->d_ino, d->d_name);
}

if (closedir(dp) == -1)
        raler("closedir");
```


Création et suppression de répertoires

```
int mkdir(const char *path, mode_t mode)
int rmdir(const char *path)
```

- `mkdir`
  - crée automatiquement les deux entrées « . » et « .. »
  - `mode` = permissions du répertoire
    - droit d'exécution : traverser le répertoire
    - règle de base : `mode = 0777` (sauf si conditions spécifiques)
    - laisser faire le masque de création de fichiers (plus tard)
- `rmdir` ne peut supprimer que les répertoires vides
  - à l'exception de « . » et « .. » (qu'on ne peut pas supprimer)
- commandes `mkdir` et `rmdir` : simples appels aux primitives systèmes

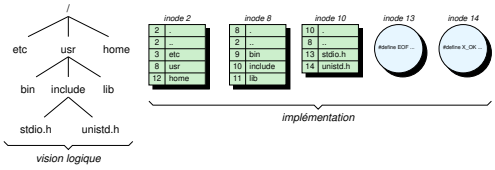
Plan

Liens physiques

En shell :

- Commande shell `ln` : `ln toto titi`
- `int link(const char *old, const char *new)`
- ⇒ ajoute un deuxième nom `titi` au fichier toto

Comment ça marche ? Rappel :

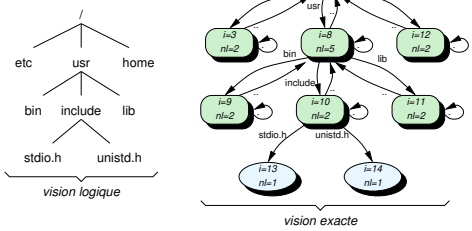


En réalité, la vision logique est biaisée...

Liens physiques

L'arborescence est un graphe orienté G = <S, A> où :

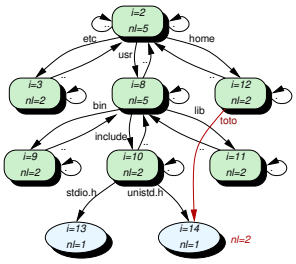
- sommets ∈ S : inodes (répertoires ou fichiers réguliers)
- arcs étiquetés ∈ A : entrées de répertoires = noms



- si on exclut « . » et « .. » : graphe sans cycle
- nl : nombre de liens (`st_nlink` avec `stat`)

Liens physiques – Ajout

- ajouter un arc dans le graphe
- `link("/usr/include/unistd.h", "/home/toto")`



- ajouter un nom augmente le nombre de liens
- le nouveau nom a exactement le même statut que l'ancien
  - il n'y a pas de nom « principal », les 2 sont équivalents

Liens physiques – Suppression

- supprimer un arc dans le graphe
- `int unlink(const char *path)`
- ⇒ décrémente le nombre de liens
- si le nombre de liens = 0
  ⇒ l'espace occupé par le fichier est libéré sur le disque
- `unlink` est la primitive pour supprimer un fichier
  - ne fonctionne pas pour les répertoires (rappel : `rmdir`)

Liens physiques

Limitations des liens physiques :

- le nouveau nom doit être sur le même disque
  - une entrée dans un répertoire ne contient qu'un numéro d'inode
    ⇒ l'inode est cherché sur le même disque
- on ne peut pas créer de lien vers un répertoire
  - sinon, on pourrait créer des cycles dans le graphe
  - difficile de faire un parcours récursif
    (find, recopier une arborescence, tar, etc.)
- l'ancien nom doit exister
  - pas de lien vers un fichier qui n'existe pas encore
  - exemple : un raccourci vers un fichier sur un système de fichiers pas encore monté (montage à la demande)

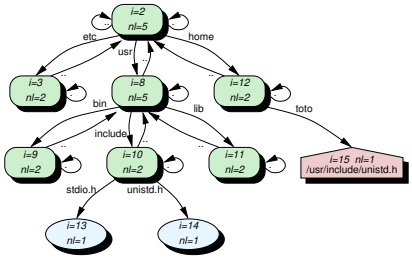
Liens symboliques

1982 : U. Berkeley diffuse BSD 4.2

- liens symboliques
- ex: `ln -s /tmp /home/pda/temp`
- `int symlink(const char *old, const char *new)`
- nouveau type de fichier (en plus de régulier et répertoire)
  - avec `stat` : `S_IFLNK` et `S_ISLNK()`
- un lien symbolique contient le nom de l'objet référencé
  - fichier, répertoire, lien symbolique, ou autre
  - peut même ne pas exister

Liens symboliques – Implémentation

- un lien symbolique contient le nom de l'objet référencé
- nouveau type de fichier



- lorsqu'un lien symbolique est trouvé :
  - si nom absolu : on repart de la racine
  - si nom relatif : on continue à partir du répertoire dans lequel est le lien

Liens symboliques – Lecture d'un lien

```
ssize_t readlink(const char *path, char *buf, size_t taille)
```

- `path` est un fichier de type « lien symbolique »
- le contenu du lien est placé dans `buf`
- au plus `taille` octets placés dans `buf`
  - troncature possible
- renvoie le nombre d'octets placés dans `buf` (ou -1)

Liens symboliques – Retour sur stat

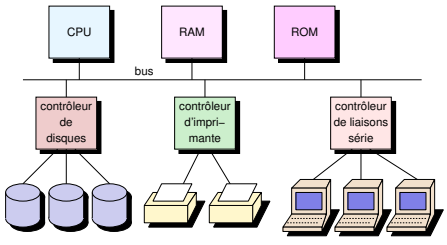
Comment se comporte `stat` avec les liens symboliques ?

- sur un lien symbolique, `stat` lit les attributs du fichier pointé
  - -1 si le lien référence un objet inexistant
- pour savoir si le fichier est un lien symbolique :
  `int lstat(const char *path, struct stat *stbuf)`
  - `lstat` lit les attributs, même s'il s'agit d'un lien symbolique
  - exemple : `tar` archive les liens, pas les fichiers pointés

Plan

Plan

Introduction



Sous Unix, les périphériques sont accédés comme des fichiers
⇒ adage « tout est fichier »

Introduction

- imprimer sur l'imprimante :

```
int fd;

if ((fd = open("/dev/lpr", O_WRONLY)) == -1)
        raler("open");
if (write(fd, "ligne␣a␣imprimer\r\n", 18) == -1)
        raler("write");
if (close(fd) == -1)
        raler("close");
```

- lire le contenu d'un disque dur :

```
uint8_t bloc[512];
int fd;

if ((fd = open("/dev/disk2", O_RDONLY)) == -1)
        raler("open");
if (read(fd, bloc, sizeof bloc) == -1)
        raler("read");
if (close(fd) == -1)
        raler("close");
```

Où est la magie ?

Nouveaux types de fichiers

Deux nouveaux types de fichiers spéciaux (périphériques) :

mode « caractère »	mode « bloc »
avec `stat` : `S_IFCHR` et `S_ISCHR()`	avec `stat` : `S_IFBLK` et `S_ISBLK()`
aurait dû s'appeler : mode « brut »	aurait dû s'appeler : mode « bufferisé »
toute suite d'octets passée à `read` ou `write` est transférée immédiatement	toute suite d'octets passée à `read` ou `write` est bufferisée, avant d'être transférée sur le périphérique
tous les périphériques ou presque	essentiellement les disques durs
périphérique identifié par un couple <*majeur*, *mineur*>	périphérique identifié par un couple <*majeur*, *mineur*>

Fichiers localisés (traditionnellement) dans `/dev`

Numéro de périphérique

Périphériques identifiés par un couple <majeur, mineur>

majeur	numéro de pilote (brut ou bufferisé)
mineur	numéro de périphérique géré par ce pilote (+ autres informations éventuellement)

- Exemple : pilote de disques « sd » (Linux)
  - majeur = 8
  - mineur = adresse du disque (bits ≥ 4) + numéro de partition (bits 0..3)
- type `dev_t` : majeur + mineur

Fichiers périphériques – Création

```
int mknod(const char *path, mode_t mode, dev_t dev)
```

- crée un fichier périphérique
- primitive restreinte à l'administrateur du système
- `mode` : analogue à `st_mode` (type et permissions)
- usage non supporté par POSIX
- primitive désuette : périphériques créés automatiquement

Fichiers périphériques – Primitive stat

Retour sur `stat` :

- si le fichier est un périphérique (bloc ou caractère)
  - champ `st_mode` testé avec `S_ISBLK()` ou `S_ISCHR()`
  - champ `st_rdev` : numéro du périphérique
- pour tous les fichiers
  - réguliers, répertoires, liens symboliques, périphériques, etc
  - le fichier réside sur un disque
  - champ `st_dev` : numéro de périphérique du disque

Plan

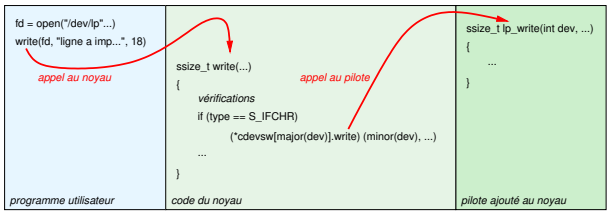
Pilotes de périphérique

- pilote de périphérique :
  - ensemble de fonctions
  - code compilé ajouté au noyau
  - fonctions référencées dans la table des pilotes du noyau
- fonctions différentes suivant le type de pilote :

mode « caractère »	mode « bloc »
fonctions `open`, `close`, `read`, `write`, `ioctl` et interruption	fonctions `open`, `close`, `strategy` et interruption

Pilotes de périphérique

Cheminement d'une requête :



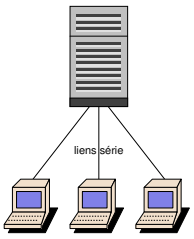
Mode caractère

Interface du pilote :

- fonctions `open`, `close`, `read` et `write` : bien connues
- traitement d'interruption : appelée lorsqu'une interruption est générée à la fin du traitement par le périphérique
- fonction `ioctl` : correspond à la primitive
  `int ioctl(int fd, int requete, .... paramètre)`
  - non normalisée par POSIX (pour les périphériques)
  - opérations spécifiques qui ne rentrent pas dans le modèle read/write
  - exemples :
    - interroger l'imprimante pour connaître son état (en cours d'impression, plus de papier, etc.)
    - éjecter le support (cartouche magnétique, CD, DVD, etc.)
    - changer la vitesse de la liaison série

Mode caractère

Un exemple particulier : le terminal (télétype)



- il est relié par un lien série
  - paramètres : vitesse du lien, nombre de bits, parité
  - raccrocher le modem en fin de connexion
  - etc.
- il a des caractéristiques propres
  - caractères d'effacement, d'arrêt, de suspension, de fin de fichier, etc
  - bufferiser les caractères par ligne ou les envoyer sans attendre
  - nombre de lignes, de colonnes
  - etc.

⇒ commande `stty` pour changer les paramètres (⇒ `ioctl`)

Mode caractère

Un exemple particulier : le terminal (télétype)

- le pilote a pour mission d'acheminer les octets jusqu'au terminal
- certains programmes (ex: `zsh`, `vi`, `more`, etc.) doivent pouvoir en plus effacer l'écran, positionner le curseur à un certain endroit, reconnaître les touches de fonction, etc.
- séquences de contrôle différentes suivant les terminaux
- positionner le curseur à la ligne X et colonne Y : il faut envoyer...
  - ESC [X ; Y f pour un terminal DEC VT100
  - ESC & a Y c X Y pour un terminal HP 2645
  - Note : ESC = l'octet de code 27
- base `termcap`, puis `terminfo` pour l'ensemble des séquences
  - variable d'environnement `TERM` : indique le type de terminal
- les programmes (`zsh`, `vi`, etc.) doivent utiliser cette base
  ⇒ ce n'est pas la mission du pilote

Mode caractère

Même principe pour la plupart des périphériques :

- le pilote achemine les octets jusqu'au périphérique
- ceci ne dispense pas les applications de gérer le protocole de chaque périphérique
  - plusieurs protocoles différents pour les souris
    ⇒ seul le serveur X-Window doit les connaître
  - chaque imprimante dispose de son « langage de contrôle »
    ⇒ tout programme souhaitant imprimer doit disposer d'une collection d'adaptateurs pour les différentes imprimantes (parfois nommés à tort « pilotes »)
  - etc.

Mode bloc

Interface du pilote :

- fonctions `open`, `close` : appelées au montage/démontage du système de fichiers dans l'arborescence
- traitement d'interruption : idem mode brut
- fonction `strategy` : 2 rôles
  - lit un bloc en mémoire, dans le « *buffer cache* » (plus tard)
  - écrit un bloc modifié du « *buffer cache* » vers le disque
  - permet d'implémenter des optimisations
    (ex : algorithme de l'ascenseur)

Note : la plupart des pilotes en mode bloc sont également accompagnés d'un pilote en mode caractère (pour `ioctl`)

Pseudo-périphériques

Un pilote peut offrir un service accessible via `read` ou `write` sans qu'il y ait un vrai périphérique

Exemples :

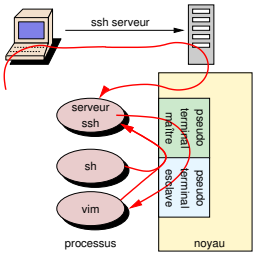
`/dev/null`	poubelle
`/dev/mem`	toute la mémoire de l'ordinateur
`/dev/random`	source d'aléa

Pseudo-périphériques

Cas particulier : les pseudo-terminaux

- beaucoup de programmes sont conçus pour être connectés à un terminal
- comment fait `vi` lorsqu'on est connecté via `ssh` ou via un terminal X-Window ?
  ⇒ il faut simuler un terminal et un lien série
- abstraction : pseudo-terminal
  - paire de pseudo-périphériques : maître et esclave
  - gérés par le même pilote
  - le serveur ssh gère le maître
  - les programmes de la session accèdent à l'esclave : il simule un « vrai » terminal

Pseudo-périphériques

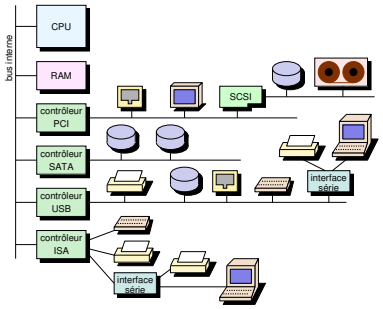


- le serveur ssh ouvre la paire
- tout octet émis vers le maître (resp. esclave) est transmis vers l'esclave (resp. maître)
- le serveur ssh transmet les octets reçus depuis le réseau vers le maître
- les programmes de la session ssh (`sh`, `vi`) sont connectés à l'esclave
- tout changement de paramètre terminal (via `ioctl`) est transmis au serveur ssh

- autres utilisations :
  - fenêtres « terminal » en environnement graphique
  - `script` (enregistrement de session)
  - programmes `screen` et `tmux`

Évolutions

Réalité : malheureusement très complexe...



Exemple : disques connectés via le bus SATA, SCSI, USB ou même via un ancestral bus IDE.

Évolutions

- complexité matérielle accrue
  - gestion des différents niveaux de bus
  - partage de code entre pilotes (exemple : disques)
- dynamicité des périphériques
  - connecter ou déconnecter des périphériques « à chaud »
  - auto-reconnaissance des périphériques

⇒ complexité des pilotes

Plan

Le répertoire /dev

Historiquement, le répertoire `/dev` était peuplé « à la main »

- peu d'ajout ou de suppression de périphériques
- commande `mknod`
- script `MAKEDEV` dans `/dev`

Le répertoire /dev

Évolution ⇒ ajout dynamique de périphériques

- Système de fichier « devfs »
  - création/destruction automatique des fichiers spéciaux
  - nécessite des périphériques capables de s'identifier
    - « plug and play »
- Programme additionnel pour gérer les exceptions
  - je veux un lien `/dev/cédérom` vers le troisième lecteur de CD de mon système
  - je veux pouvoir connecter mon appareil photo sans avoir les droits de l'administrateur

Plan

Plan

Définition d'un processus (haut niveau)

Définition (haut niveau) :

> *un processus est une instance d'un programme en cours d'exécution*

... mais pas n'importe quelle exécution :

- j'exécute `ls /tmp` : le processus correspond à l'exécution du programme `ls` avec les données `/tmp`
- quelqu'un d'autre exécute `ls /tmp` en même temps : ce n'est pas la même exécution, même si c'est le même programme et les mêmes données
  - ce n'est pas le même processus
  - même si le « quelqu'un d'autre », c'est moi

Attributs d'un processus

Un processus possède des attributs :

- état (prêt à tourner, en attente, etc.)
- identificateur de processus (pid)
- identificateur de processus parent (ppid)
- propriétaire (uid), groupe (gid)
- ouvertures de fichiers
- répertoire courant
- terminal de contrôle
- localisation en mémoire
- consommation de temps CPU
- etc.

Définition d'un processus (bas niveau)

Définition (bas niveau) :

un processus est décrit par :

- *un espace mémoire pour le programme et les données*
- *des attributs*
- *un contexte matériel*
  - *registres du processeur*
  - *traduction d'adresses*

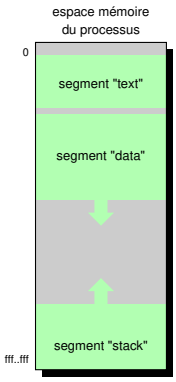
(voir `task_struct` sur `http://www.tldp.org/LDP/tlk/ds/ds.html`)

À chaque fois :

- qu'un processus est retiré du processeur
  - son contexte est sauvegardé (espace mémoire, registres du processeur, etc.)
- qu'un processus est mis sur le processeur
  - son contexte est restauré

Espace mémoire d'un processus

L'espace mémoire d'un processus est découpé en 3 zones :



- segment « text »
  - programme (code compilé)
  - adresse 0 pas utilisée : pourquoi ?
- segment « data »
  - variables globales (+ `static` locales)
  - tas (mémoire allouée par `malloc`)
  - extension explicite (via `malloc`)
- segment « stack » : la pile d'exécution
  - variables locales
  - arguments des fonctions
  - adresses de retour
  - extension implicite (utilisation de la pile)
- d'autres zones peuvent être ajoutées
  - bibliothèques partagées
  - mémoire partagée entre processus
  - ⇒ cf semestre 5

Plan

Identité

```
pid_t getpid(void)      pid_t getppid(void)
uid_t getuid(void)      gid_t getgid(void)
```

- Primitives simples...
- Ne renvoient pas -1 en cas d'erreur (pas d'erreur possible)
- Exemple :

```
pid_t pid, ppid;
uid_t uid;
gid_t gid;

pid = getpid();
printf("je␣suis␣le␣processus␣%d\n", pid);
ppid = getppid();
printf("mon␣pere␣est␣%d\n", ppid);
uid = getuid();
printf("mon␣proprio␣est␣%d\n", uid);
gid = getgid();
printf("et␣mon␣groupe␣est␣%d\n", gid);
```


Identité

```
int setuid(uid_t uid)      int setgid(gid_t gid)
```

- Primitives restreintes à l'administrateur (uid = 0)
- Utilisées lors de l'admission sur le système :
  - `/bin/login`, `sshd` ou équivalent pour X-Window
  - lancé par le processus numéro 1 ou un de ses descendants

Identité

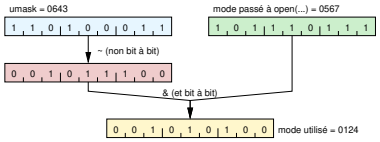
Algorithme de l'admission sur le système :

1. demander login et mot de passe
2. chercher l'entrée dans `/etc/passwd`
   - lignes de la forme :
     `toto:sel+mot-de-passe-chiffré:uid:gid:nom:répertoire:shell`
   - sel et mot de passe chiffré sont mis dans un fichier séparé (`/etc/shadow` sur Linux)
3. chiffrer le mot de passe (avec le « sel » cité dans l'entrée)
4. comparer le mot de passe avec l'entrée
5. si identique
   - générer un nouveau processus
   - `setuid(uid)` / `setgid(gid)` dans ce nouveau processus
   - lancer l'exécution du shell (ou du « window manager »)

Masque de création de fichiers

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

- Création de fichier (`open(... O_CREAT...)`, `mkdir` ou `mknod`)
  - permissions du fichier créé = mode `&` `~` umask



- Recommandation : dans les programmes ⇒ 0666 ou 0777
  - les programmes sont généraux
  - laisser l'utilisateur gérer son niveau de confidentialité
    - commande `umask` du Shell
  - sauf pour les programmes sensibles à la sécurité
- `umask` renvoie l'ancien masque (avant modification)

Répertoire courant

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

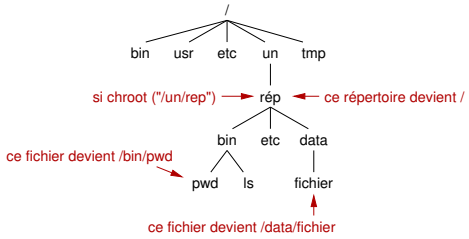
- Modifie le répertoire courant du processus
- Rappel : le répertoire courant est un attribut du processus
  - changer dans un processus n'affecte pas les autres processus
- Pas de primitive système pour récupérer le nom du répertoire courant
  - c'est une fonction de bibliothèque

    ```
    char *getcwd(char *buf, size_t max)
    ```
  - comment fonctionne-t'elle ?

Racine courante

```
int chroot(const char *path)
```

- Modifie la racine courante du processus



- Pas de possibilité de contournement
  - à la nouvelle racine, « `..` » reste au même endroit
  - on ne peut que descendre dans l'arborescence
- Accessible seulement à l'administrateur du système
- Primitive non normalisée par POSIX

Racine courante

`chroot` est un système de confinement :

- restreindre l'environnement à certains fichiers seulement
  - exemple : comptes utilisateurs spécifiques pour accéder à une application seulement
  - exemple : serveur « FTP anonyme »
    service FTP pour distribuer des fichiers, restreint à une portion de l'arborescence
- `chroot` : base des systèmes de « conteneurs » modernes
  - LXC sur Linux, Jails sur FreeBSD
  - complété par d'autres systèmes de confinement
    - visibilité restreinte des processus
    - visibilité restreinte des connexions réseau
    - etc.
  - machines « virtuelles » à moindre coût

Plan

Création des processus

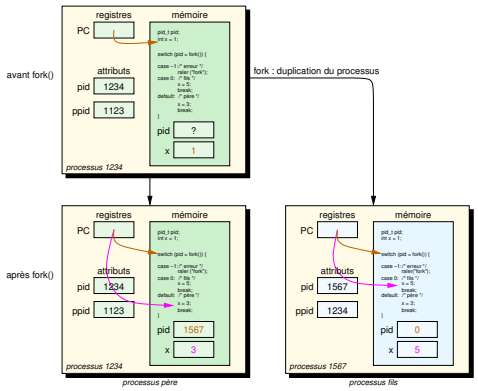
```
pid_t fork(void)
```

Créer un processus ⟺ dupliquer le processus

- Primitive `fork` = photocopieuse à processus
- Processus = mémoire + attributs + contexte matériel
  - (presque) tout est dupliqué
  - mémoire dupliquée ⇒ variables dupliquées
  - les registres du CPU aussi : chaque programme évoluera (registre PC) séparément
- Duplication ⇒ informations **héritées** du processus père
  - répertoire courant, ouvertures de fichiers, etc.
  - sauf quelques attributs : pid, ppid, temps CPU, etc.
- Après une photocopie, on peut écrire sur chaque feuille de papier, ça ne modifie pas l'autre feuille
  - c'est la même chose avec `fork`

Création des processus

```
pid_t pid;
int x = 1;

switch (pid = fork()) {
case -1: // erreur
        raler("fork");

case 0:  // fils
        x = 5;
        break;

default: // père
        x = 3;
        break;
}
```



Après `fork` : les variables `x` et `pid` évoluent séparément

Création des processus

- Particularité de `fork` :
  - un seul appel, deux retours
    - comme `fork` duplique le processus, tout se passe pour le deuxième comme s'il avait lui-même appelé `fork`
- Valeur de retour de `fork` :
  - -1 en cas d'erreur (classique)
  - 0 pour le processus fils
  - une valeur > 0 pour le processus père
    - c'est le pid du fils
- Pas de déterminisme :
  - après `fork`, on ne sait pas si le père ou le fils est remis sur le processeur en premier
  - pas un problème : les processus évoluent indépendamment

Création des processus

- Quel moyen mnémotechnique pour la valeur de retour ?
  - Chaque processus connaît son père
    - attribut ppid, primitive `getppid`
    - information facile à retrouver
    - ⇒ pas besoin de récupérer le pid du père avec `fork`
    - `fork` renvoie donc 0 pour le fils
  - Pas de moyen facile pour récupérer le pid du fils
    - il peut y en avoir beaucoup, lequel renvoyer ?
    - pas d'attribut, pas de primitive
    - ⇒ seul moyen de récupérer le pid du fils = `fork`
    - `fork` renvoie au père le pid du fils

Création des processus

Conseils pour apprivoiser `fork`

- `switch` avec 3 cas : `-1`, `0` et `default`
  - pour être sûr de n'oublier aucun cas
- dans le cas 0 (fils), faire appel à une fonction `fils` et terminer par `exit`
  - isoler le code du fils
  - placer un « cordon sanitaire » afin que le fils n'exécute pas le code prévu pour le père

Création des processus

Exemple :

```
pid_t pid;

switch (pid = fork()) {
case -1: // erreur : ne pas oublier
        raler("fork");

case 0:  // le fils
        fils();  // isoler le fils
        exit(0);  // cordon sanitaire

default: // le père
        ...
}
```


Terminaison des processus

```
void exit(int code)
```

- Termine le processus en cours
- Pas de retour
  - ... et donc pas -1 en cas d'erreur
- Presque toutes les ressources sont libérées
  - mémoire, CPU, etc.
  - cas particulier pour les processus zombies (plus tard)

Terminaison des processus

Argument de `exit`

- code ∈ [0..255]
  - si vous voyez `exit(-1)` dans un programme, c'est que son auteur n'a pas assimilé son cours de système...
- valeur 0 : ok
  - convention utilisée par les shells
  - si le père n'est pas un shell, on peut utiliser une autre convention
- constantes POSIX : `EXIT_SUCCESS` et `EXIT_FAILURE`

Terminaison des processus

En réalité, `exit` est une fonction de bibliothèque

- vide les buffers des fichiers ouverts par `fopen`
- appelle les fonctions enregistrées par `atexit`
- la vraie primitive s'appelle `_exit`
  - jamais appelée directement

Terminaison des processus

```
pid_t wait(int *raison)
pid_t waitpid(pid_t pid, int *raison, int options)
```

- Attend la terminaison d'un des processus fils
  - `wait` attend n'importe quel fils
    - si un processus fils est déjà terminé, pas d'attente
    - si pas de processus fils, renvoie -1
    - s'il y a au moins un processus fils, attente
    - si attente interrompue par un signal, renvoie -1
  - `waitpid` attend un processus spécifique (voir manuel)
- La terminaison peut avoir plusieurs causes :
  - le fils appelle `exit`
  - le fils reçoit un signal
    - CTRL C, violation de segment, etc.
  - ce peut être autre chose qu'une terminaison
    - processus ayant atteint un point d'arrêt avec un débogueur

Terminaison des processus

- Entier pointé par `raison` : raison de la terminaison

	code retour	poids fort	poids faible
processus stoppé en mode trace	identificateur du processus	numéro du signal	0177
processus terminé par exit	identificateur du processus	argument de exit sur 8 bits	0
processus terminé par signal	identificateur du processus	0	numéro du signal (+0200 si core)
wait interrompue par signal	-1	?	?

- POSIX simplifie le travail : macros les plus courantes

Arrêt avec `exit` ?	WIFEXITED()
⇒ si oui, code de retour	WEXITSTATUS()
Arrêt sur signal ?	WIFSIGNALED()
⇒ si oui, numéro du signal	WTERMSIG()

Terminaison des processus

Exemple :

```
pid_t pid;
int raison;

switch (pid = fork()) {
case -1:
        raler("fork");

case 0:  // le fils
        fils();
        exit(0);

default:  // le père
        if (wait(&raison) == -1)
                raler("wait");

        if (WIFEXITED(raison))
                printf("exit(%d)\n", WEXITSTATUS(raison));
        else if (WIFSIGNALED(raison))
                printf("signal %d\n", WTERMSIG(raison));
        else
                printf("autre raison\n");
}
```


Cas particulier – Processus zombie

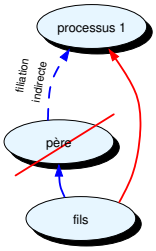
Définition :

Un processus zombie est un processus terminé, dont le père n'a pas encore enregistré la terminaison avec wait

- un processus zombie est « quasiment » terminé
  - presque toutes ses ressources sont libérées...
  - ... sauf le descripteur du processus
    - il contient la raison de la terminaison
    - ainsi qu'un résumé de l'utilisation des ressources
  - sans limitation de durée
  - il reste visible avec `ps`
- lorsque le père utilise `wait` :
  - il collecte les informations nécessaires
  - le descripteur de processus est libéré
    - le processus disparaît alors complètement du système

Cas particulier – Processus orphelin

Que se passe-t'il lorsque le père d'un processus se termine ?



- père se termine ⇒ zombie
- le fils est « reparenté » (ppid ← 1)
- le processus 1 est spécial
  - ne s'arrête jamais
  - (re-)démarre les programmes du système

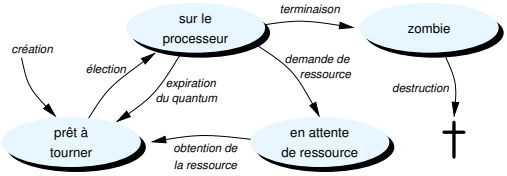
```
liste = lire_fichiers("/etc/init/");
demarrer_toutes_les_taches(liste);
for (;;) {
        pid = wait(&raison);
        ps = chercher_tache(pid, liste);
        if (ps != NULL)
                redemarrer_tache(ps);
}
```

⇒ le fils est immédiatement reparenté
⇒ le statut d'orphelin n'existe donc pas dans le noyau

Note : sur certaines versions de Linux, le parent devient le gestionnaire de la session utilisateur (`systemd`) et non le processus 1

États d'un processus

États d'un processus :



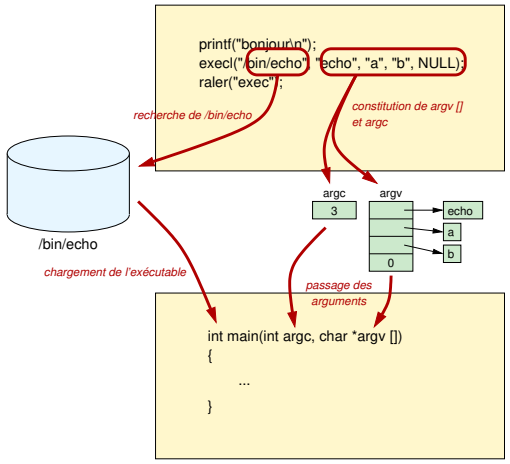
Plan

Exécution d'un fichier

```
int execl(char *path, char *arg, ... )
int execv(char *path, char *argv [])
int execle(char *path, char *arg, ..., char *envp [])
int execve(char *path, char *argv [], char *envp [])
int execlp(char *fichier, char *arg0, ...)
int execvp(char *fichier, char *argv [])
```

- Primitives `exec*` : remplacent le programme du processus courant par un nouveau programme et ses arguments
  - le contexte du processus reste quasiment inchangé
    - attributs inchangés : pid, ppid, uid (sauf exception), umask, répertoire courant, consommation de ressources, etc.
    - attributs modifiés : référence à l'exécutable, ouvertures de fichiers (sauf exception, notamment pour 0, 1 et 2), etc.
  - mémoire initialisée avec le nouveau programme
    - segments text, data et stack
- Valeur de retour = -1 (toujours)
  - retour ⇒ nouveau programme non chargé ⇒ erreur

Exécution d'un fichier



Exécution d'un fichier – Exemple

Pas de retour pour `exec*` ⇒ le plus souvent utilisée avec `fork`

```
switch (fork()) {
case -1:
        raler("fork");

case 0: // fils
        execl("/bin/echo", "echo", "a", NULL);
        raler("execl");

default: // père
        if (wait(&raison) == -1)
                raler("wait");
        if (WIFEXITED(raison) && WEXITSTATUS(raison) == 0)
                printf("ok\n");
        else
                printf("pas ok\n");
}
```

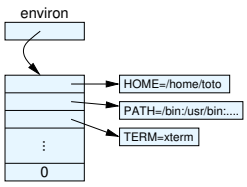
Exécution d'un fichier – Exemple

Autre exemple : algorithme (grossier) du Shell

1. lire une ligne
2. découper la ligne en éléments
3. si éléments [0] ∈ commandes internes
   - alors exécuter la fonction correspondante
   - revenir en 1
4. localiser le fichier éléments [0] dans `PATH`
5. si pas trouvé, alors erreur et revenir en 1
6. `fork` ⇒ fils
   - `exec (` éléments `)`
7. si éléments [end] ≠ `&`
   - alors attendre la fin du fils
8. revenir en 1

Exécution d'un fichier – Environnement

En plus des arguments, `exec*` passe l'environnement :



- `extern char **environ`
- fonction de bibliothèque `getenv`
- environnement modifié par le Shell
  - hérité par tous les processus lancés par le Shell
  - rappel : utiliser `export` en Shell pour exporter une variable d'environnement

Exécution d'un fichier

Six formes pour `exec*` :

- passage des arguments :

`execl`	en liste : `execl(..., "echo", "a", "b", NULL)`
`execv`	en vecteur : `execv(..., tabargv)`

- recherche dans la variable shell `PATH` :

`exec[vl]`	non : `execv("/bin/echo", tabargv)`
`exec[vl]p`	oui : `execvp("echo", tabargv)`

- passage de l'environnement :

`exec[vl]`	implicite : `execv("/bin/echo", tabargv)`
`exec[vl]e`	explicite : `execve("echo", tabargv, tabenvp)`

Une seule de ces formes est une primitive système (laquelle ?)
⇒ les autres sont des fonctions de bibliothèque

Nombres magiques

`exec*` peut exécuter plusieurs sortes de fichiers :

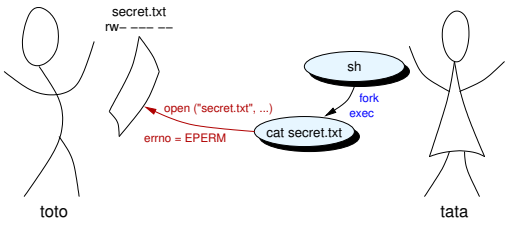
- Fichiers binaires (compilés)
  - plusieurs formats possibles
    - évolution des formats, compatibilité avec anciennes versions
    - sur FreeBSD : mode de compatibilité Linux
- Fichiers interprétés
  - fichiers non directement exécutables
    - recours à un interprète
    - Shell, Awk, Perl, Tcl, Ruby, Python, etc.
  - l'interprète ouvre le fichier et l'« exécute »

Présence d'un « nombre magique » en début de fichier

fichiers binaires	`0x7f 'E' 'L' 'F'` (format ELF sous Linux)
fichiers interprétés	`'#' '!'` (suivi du chemin de l'interprète)

Plan

Droits d'exécution



- `fork` : le fils hérite de l'uid (celui de tata)
- `exec` : ne change pas l'uid (celui de tata)
- le fichier `secret.txt` de l'utilisateur toto ne peut pas être ouvert par un processus appartenant à tata
- ⇒ le système de droits fonctionne bien !

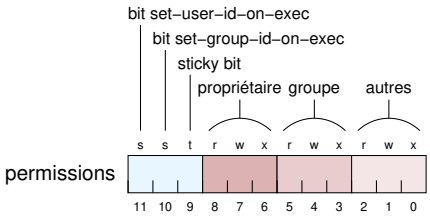
Droits d'exécution – Élévation de privilège

Dans certains cas, il faut pouvoir exécuter un programme avec des droits plus élevés :

- un utilisateur change son mot de passe ⇒ il doit écrire le nouveau mot de passe chiffré dans `/etc/passwd`
  - `/etc/passwd` n'est pas modifiable par l'utilisateur
    ```
    $ ls -l /etc/passwd
    -rw-r--r-- 1 root 12345 Jan 1 1970 /etc/passwd
    ```
- un utilisateur insère la carte SD de son appareil photo ⇒ le système de fichiers sur la carte doit être « monté »
  - l'opération de montage (primitive système `mount`) n'est accessible qu'à l'administrateur
- un utilisateur souhaite utiliser la commande `ping`
  - `ping` accède aux couches réseau de bas niveau et nécessite les privilèges de l'administrateur

Droits d'exécution – Élévation de privilège

Retour sur les permissions de fichiers :



- bit « *set-user-id-on-exec* » (ou bit « suid »)
- bit « *set-group-id-on-exec* » (ou bit « sgid »)

Droits d'exécution – Bit set-user-id-on-exec

- Appel à `exec` : si le bit « suid » est à 1, alors l'uid du processus devient l'uid du propriétaire du fichier
  - autrement dit : commande exécutée avec les droits du propriétaire (de la commande) et non ceux de l'utilisateur
- Exemple :
  - la commande `/bin/passwd` appartient à root
  - dans ses permissions, le bit « suid » est à 1
  - ⇒ le processus peut donc modifier `/etc/passwd`
  - ⇒ l'utilisateur peut changer son mot de passe !

Droits d'exécution – Bit set-user-id-on-exec

- Problème 1 : `/bin/passwd` doit connaître l'uid de l'utilisateur qui change son mot de passe...
  ⇒ 2 notions d'uid distinctes :

uid réel	l'humain derrière son terminal	`uid_t getuid(void)`
uid effectif	l'uid servant à tester les droits	`uid_t geteuid(void)`

Droits d'exécution – Bit set-user-id-on-exec

- Problème 2 : pour certaines opérations, il faut utiliser l'uid réel et non effectif
  - exemple : création de fichier ⇒ propriétaire = uid effectif
  - parfois, il faut repasser temporairement sous l'identité de l'utilisateur réel
    - exemple : pour créer le fichier sous la bonne identité
  - d'où une troisième notion : uid « sauvé »
    - l'uid sauvé permet de sauver l'uid effectif si jamais on le change (avec `int seteuid(uid_t euid)`)
    - permet de passer sous l'identité de l'uid réel, puis de repasser à nouveau sous l'identité privilégiée

Droits d'exécution – Bit set-group-id-on-exec

Application des mêmes principes au groupe :

- bit « set-group-id-on-exec »
- 3 identités
  - gid réel
  - gid effectif
  - gid sauvé

Note :

- `ls` affiche ces bits
- exemple :
  ```
  $ ls -l /usr/bin/passwd /usr/bin/crontab
  -rwsr-xr-x 1 root root    12345 Jan 1 1970 /usr/bin/passwd
  -rwxr-sr-x 1 root crontab 23456 Jan 1 1970 /usr/bin/crontab
  ```

Droits d'exécution – Élévation de privilège

Bits « suid » et « sgid » : changent le niveau de privilège

- le plus souvent : pour élever le niveau
  - même si ça peut arriver de le diminuer
- ce sont des problèmes de sécurité potentiels
  - privilèges ⇒ attention à la programmation !
  - bien vérifier les droits
  - pas de « trou » de sécurité
    ⇒ débordement de tampon, test des primitives, etc.
- limiter le nombre d'exécutables avec ces bits
  - Exemple sur turing.u-strasbg.fr (Ubuntu 14.04) :

bit « suid »	32 fichiers
bit « sgid »	30 fichiers

Plan

Redirections

Le Shell permet de réaliser des redirections :

- `$ wc -l < entree > resultat 2> erreurs`
- Rappel : 3 ouvertures par défaut :

0	entrée standard
1	sortie standard
2	sortie d'erreur standard

- Redirection = modification du descripteur 0, 1 ou 2
- À faire dans le fils (et pas dans le père), avant `exec`

Redirections

Exemple : le shell redirige la sortie standard

1. lire une ligne
2. découper la ligne en éléments
3. si éléments [0] ∈ commandes internes
   - alors exécuter la fonction correspondante
   - revenir en 1
4. localiser le fichier éléments [0] dans `PATH`
5. si pas trouvé, alors erreur et revenir en 1
6. `fork` ⇒ fils
   - `close(1)`
   - `open("toto", O_WRONLY | O_CREAT...)` ⇒ renvoie 1
   - `exec(` éléments `)`
7. si éléments [end] ≠ `&`
   - alors attendre la fin du fils
8. revenir en 1

Redirections

Plusieurs manières de modifier les descripteurs :

1. fermer un descripteur puis ouvrir un nouveau fichier
   - cf exemple précédent
   - par construction, `open` prend le plus petit descripteur disponible

Redirections

Plusieurs manières de modifier les descripteurs :

2. primitive `int dup(int fd)` : duplique une ouverture de fichier
   - exemple :

```
fd = open("toto", O_WRONLY | O_CREAT | O_TRUNC, 0666);
if (fd == -1)
        raler("open");
if (close(1) == -1)
        raler("close 1");
if (dup(fd) == -1)
        raler("dup");
if (close(fd) == -1)
        raler("close");
execv(path, tabargv);
raler("execv");
```

   - intérêt : ouvrir le fichier dans le père, avant `fork`, afin d'éviter de générer un processus en cas d'erreur

Redirections

Plusieurs manières de modifier les descripteurs :

3. primitive `int dup2(int oldfd, int newfd)`
   - exemple :

```
fd = open("toto", O_WRONLY | O_CREAT | O_TRUNC, 0666);
if (fd == -1)
        raler("open");
if (dup2(fd, 1) == -1)
        raler("dup2");
if (close(fd) == -1)
        raler("close");
execv(path, tabargv);
raler("execv");
```

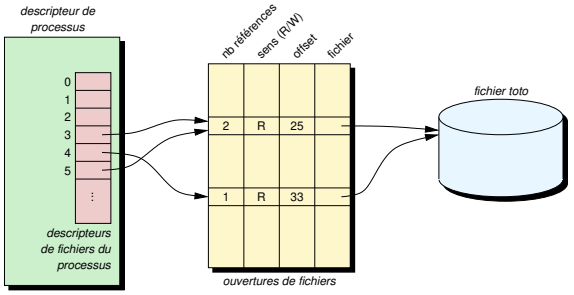
   - intérêt 1 : `dup2` ferme le descripteur de destination si nécessaire
   - intérêt 2 : facilite la sélection du nouveau descripteur

Partage d'ouvertures de fichiers – dup/dup2

Structures de données du noyau :

- une table globale pour toutes les ouvertures de fichiers
- une table par processus pour ses descripteurs

```
fd1 = open("toto", O_RDONLY);
fd2 = open("toto", O_RDONLY);
dup2(fd1, 5);

read(fd1, ..., 10);
read(fd2, ..., 33);
read(5, ..., 15);
```

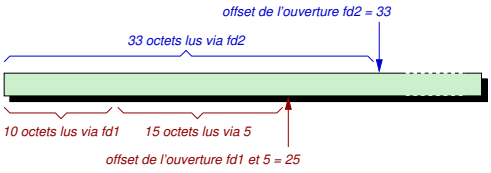


Partage d'ouvertures de fichiers – dup/dup2

Lecture des données dans le fichier :

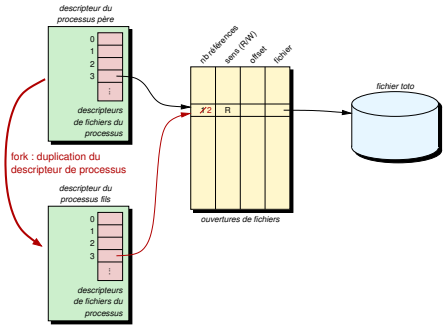
- offset partagé entre fd1 et 5
- offset propre à fd2

```
fd1 = open("toto", O_RDONLY);
fd2 = open("toto", O_RDONLY);
dup2(fd1, 5);

read(fd1, ..., 10);
read(fd2, ..., 33);
read(5, ..., 15);
```



Partage d'ouvertures de fichiers – fork

Que se passe-t'il si `fork` est appelé après une ouverture de fichier ?



⇒ partage de l'ouverture entre les deux processus

Plan

Plan

Introduction

Définition

Un signal est un événement notifié par le noyau à un processus

Exemples :

- événements matériels
  - déconnexion (SIGHUP),
  - appui sur CTRL C (SIGINT)
- événements suite à une action du programme
  - erreur d'adressage mémoire (SIGSEGV)
  - instruction illégale (SIGILL),
  - alarme de processus (SIGALRM),
  - écriture dans un tube sans lecteur (SIGPIPE), etc
- événements sans sémantique associée pour le noyau
  - signaux « utilisateur » (SIGUSR1 et SIGUSR2)
  - signal de terminaison (SIGTERM)
  - signal de terminaison absolu (SIGKILL)

Les signaux sont représentés par des entiers ⇒ SIG*

Introduction – Action par défaut

Notification au processus ⇒ action par défaut du processus

- terminer le processus
  - exemple : CTRL C
  - exemple : SIGSEGV ⇒ erreur d'adressage mémoire
  - certains signaux provoquent la génération d'un fichier core
    - pour l'analyse de la mémoire à postériori
    - exemple : `$ gdb a.out core`
    - peut nécessiter : `$ ulimit -c unlimited`
- ignorer le signal
  - exemple : terminaison d'un fils ⇒ SIGCHLD
- suspendre l'exécution du processus
  - exemple : CTRL Z ⇒ SIGTSTP (terminal stop)
  - ou envoi de SIGSTOP
- reprendre l'exécution du processus
  - Job control : fg/bg en shell ⇒ SIGCONT

Introduction – Action par défaut

Quelques signaux et leurs actions par défaut :

Signal	Action	core	Signification
SIGALRM	Terminer		Alarme de processus
SIGCHLD	Ignorer		Terminaison d'un fils
SIGCONT	Reprendre		fg/bg
SIGFPE	Terminer	oui	Expression invalide (ex: division par 0)
SIGHUP	Terminer		Déconnexion
SIGINT	Terminer		CTRL C
SIGKILL	Terminer		Arme atomique...
SIGPIPE	Terminer		Écriture dans un tube sans lecteur
SIGQUIT	Terminer	oui	CTRL \
SIGSEGV	Terminer	oui	Accès à une case mémoire invalide
SIGSTOP	Suspendre		Suspendre le processus
SIGTERM	Terminer		Demande de terminaison du processus
SIGTSTP	Suspendre		CTRL Z
SIGUSR1	Terminer		Signal sans définition système 1
SIGUSR2	Terminer		Signal sans définition système 2
SIGWINCH	Ignorer		Changement de taille de fenêtre

Introduction – Changement d'action

L'action par défaut n'est pas toujours souhaitable
⇒ une action spécifique peut être associée à chaque signal

- SIG_IGN : ignorer le signal
- SIG_DFL : l'action par défaut
- exécuter une fonction définie préalablement
  - la fonction interrompt l'exécution du programme
  - lorsque la fonction se termine, le programme reprend où il avait été interrompu

Exemples d'utilisation des signaux

- sauvegarder le calcul en cours en cas d'interruption
  - appui sur CTRL C ⇒ SIGINT
  - appeler la fonction programmée pour SIGINT
- interrompre une action sans sortir du programme
  - exemple : CTRL C avec vi
- terminer le programme « proprement »
  - l'utilisateur envoie le signal SIGTERM
  - appeler la fonction programmée pour SIGTERM
    - sauvegarder les données en mémoire, supprimer les fichiers temporaires, etc.
- continuer le programme même après une déconnexion
  - déconnexion ⇒ SIGHUP
  - ignorer le signal
- planifier une action à exécuter dans 3 minutes
  - programmer une alarme ⇒ SIGALRM
  - appeler la fonction programmée pour SIGALRM

Cas particuliers

Deux signaux particuliers : impossible de modifier l'action

- SIGKILL
  - Avec les signaux, il est possible d'exécuter une fonction au lieu de terminer le processus par défaut
  - S'il est possible de programmer une fonction pour chacun des signaux, on peut avoir des processus « immortels »
  - D'où le signal SIGKILL :
    - action = action par défaut ⇒ terminer le processus
    - impossible de modifier cette action
    - il reste toujours un moyen de terminer un processus !
    - ne pas envoyer SIGKILL (= 9) directement à un processus
      ⇒ le processus ne peut pas se terminer « proprement »
      ⇒ faire d'abord des tirs de sommation (ex : SIGHUP, SIGTERM)
- SIGSTOP
  - Analogue à SIGKILL : suspension impérative de processus
  - Ne pas confondre avec SIGTSTP (envoyé suite à CTRL Z)

Plan

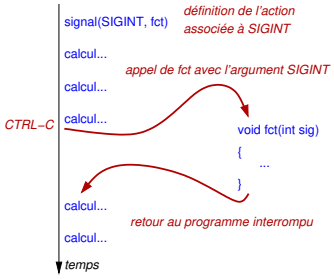
API Unix v7

Ensemble de primitives pour gérer les signaux :

```
void (*signal(int sig, void (*fct)(int sig)))(int sig)
int kill(pid_t pid, int sig)
int pause(void)
unsigned int alarm(unsigned int nsec)
```

- primitives « originelles » (Unix v7, 1977)
  - en réalité plus anciennes
  - mais Unix v7 très largement diffusé
- primitive `signal` obsolète
  - supplantée par `sigaction`, voir API POSIX
  - mais `signal` simple, toujours utilisée
  - et bonne introduction pédagogique
  - et malgré tout, toujours normalisée par POSIX
- les autres primitives sont toujours d'actualité

API Unix v7 – Primitive signal



- `signal` définit l'action à réaliser lorsque le signal **arrivera**
- prototype `fct` fixe
  - numéro de signal passé en argument
  - pas d'autre argument possible

Autres valeurs possibles pour la fonction de `signal` :

`SIG_IGN`	ignorer le signal
`SIG_DFL`	remettre l'action par défaut (plupart des signaux : terminer le processus)

API Unix v7 – Primitive signal

```
void (*signal(int sig, void (*fct)(int sig)))(int sig)
```

ou

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

- `signal` prend en argument : :
  - un numéro de signal pour lequel l'action doit être définie
  - l'adresse d'une fonction (dans le programme)
    - prenant en argument un entier (le numéro du signal reçu)
    - et ne renvoyant rien

    ou bien :
    - `SIG_DFL` : adresse == 0
    - `SIG_IGN` : adresse == 1
- `signal` renvoie l'adresse de l'ancienne fonction
  - ou bien `SIG_ERR` (adresse == -1) en cas d'erreur
    - par exemple si `sig` == `SIGKILL`
- attention : `signal(SIGHUP, fct (5))` passe en argument le **résultat** de l'appel de la fonction `fct`, et non son adresse

API Unix v7 – Primitive signal

Analogie du radio-réveil

- `signal` **n'attend pas** l'arrivée d'un signal
  - `signal` spécifie l'action à effectuer quand le signal **arrivera**
- comme un radio-réveil ou un smartphone :
  - on **règle** la radio à écouter lorsque le réveil **sonnera**
  - on **indique** la chanson à écouter lorsque le réveil **sonnera**

API Unix v7 – Fonction appelée

La fonction appelée doit avoir le prototype suivant :

```
// argument : numéro du signal reçu
void fct(int signum)
{
    // code de la fonction
}
```

Notes :

- pas de possibilité de changer le type de retour
- pas de possibilité de changer les arguments
- avec les options `-Wall -Werror` du compilateur `gcc`, il faut éviter de générer une erreur si l'argument n'est pas utilisé :

```
void fct(int signum)
{
    (void) signum;        // hack : résultat ignoré
    // reste de la fonction qui n'utilise pas signum
}
```

API Unix v7 – Fonction appelée

Attention à la fonction appelée lors de la réception d'un signal :

- L'appel de la fonction interrompt le programme en cours
- Le programme pouvait faire des choses complexes
  - exemple :

```
1   struct gros_compteur {
2           uint32_t grand;
3           uint32_t tresgrand;
4   } c = {0, 0};
5
6   void incrementer(void)
7   {
8           if (c.grand == UINT32_MAX) {
9                   c.grand = 0;
10                  c.tresgrand++;
11          } else {
12                  c.grand++;
13          }
14  }
```

- Problème :
  - si le programme est interrompu entre les lignes 11 et 12
  - et si la fonction utilise la variable `c`

API Unix v7 – Fonction appelée

Recommandations pour la fonction :

- Limiter la fonction à la modification d'une variable
- Tester la variable dans le programme principal
- Utiliser une variable « `volatile sig_atomic_t` »
  - qualificateur `volatile` : empêcher certaines optimisations intempestives
  - type `sig_atomic_t` : variable modifiée en une seule opération ⇒ voir semestre prochain

Autre action dans la fonction ⇒ fonctionnement non garanti

API Unix v7 – Fonction appelée

Exemple :

```
volatile sig_atomic_t arret = 0;

void fct(int sig)
{
    // sig_atomic_t ⇒ modification en "une" opération
    arret = 1;
}

int main(int argc, char *argv [])
{
    int n = 0;

    signal(SIGINT, fct);

    // volatile ⇒ variable relue à chaque iteration
    while (!arret)
        n++;

    printf("%d\n", n);
}
```


API Unix v7 – Signaux et processus

Actions associées aux signaux :

- ce sont des attributs du processus
- héritées avec `fork`
- réinitialisées avec `exec`

API Unix v7

Autres primitives associées aux signaux :

- `int kill(pid_t pid, int sig)`
  - envoie un signal à un processus
- `int pause(void)`
  - suspend l'exécution du programme en attendant l'arrivée d'un signal ⇒ attente *passive*
  - si le signal est ignoré, `pause` ne termine pas
  - `pause` renvoie toujours -1 ⇒ primitive interrompue par un signal
- `unsigned int alarm(unsigned int nsec)`
  - programme l'envoi de `SIGALRM` au processus courant
  - dans l'analogie du radio-réveil, `signal` règle la radio, et `alarm` règle l'heure de réveil

API Unix v7

Recommandations / rappels

- `signal` **n'attend pas** l'arrivée d'un signal
  - ne fait que modifier la fonction
  - ne pas confondre avec `pause`
- appeler `signal` avec l'**adresse** d'une fonction
  - ... et pas son résultat
  - `signal(SIGINT, f)` et non `signal(SIGINT, f(5))`
- en faire **le moins possible** dans la fonction
  - évite les problèmes de concurrence

API Unix v7 – Attention piège !

Exemple : je veux attendre la réception d'un signal :

```
1  volatile sig_atomic_t condition = 0;
2
3  void signal_machin(int signum)
4  {
5      (void) signum;
6      condition = 1;        // l'événement est arrivé
7  }
8
9  int main(...)
10 {
11     signal(SIGmachin, signal_machin);
12     ...
13     if (!condition)       // si l'événement n'est pas arrivé
14         pause();          // alors attendre l'événement
15     ...
16 }
```

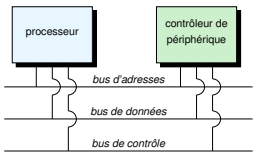
Si le signal arrive entre les lignes 13 et 14 ⇒ attente éternelle

Pas possibile de gérer cela correctement avec l'API v7 !

Plan

Analogie avec les interruptions matérielles

Mécanisme matériel :



- lorsqu'un contrôleur a terminé une requête, il active la ligne d'interruption du bus de contrôle
- lorsque le processeur termine l'exécution de l'instruction courante, il consulte la ligne d'interruption
- si elle est dans l'état « actif », le processeur interrompt le programme en cours
- le contrôleur reste « interruptif » jusqu'à ce qu'il soit interrogé par le processeur

Trois registres du processeur impliqués :

PC	Program Counter (compteur ordinal)
SP	Stack Pointer (pointeur de pile)
SR	Status Register (registre d'état)

Analogie avec les interruptions matérielles

Actions du processeur suite à une interruption :

1. lorsque l'interruption se produit, PC pointe dans le code du processus, SP dans la pile du processus et SR indique qu'on est en mode « non privilégié » (par exemple)
2. le processeur sauvegarde ces registres
3. le processeur modifie ensuite ces registres :
   - SR :
     - passage en mode « privilégié »
     - blocage (masquage) des interruptions
   - PC : initialisé à partir du vecteur d'interruption
   - SP : pointe sur la pile noyau

⇒ tout ceci est effectué par le matériel

Analogie avec les interruptions matérielles

Vecteur d'interruptions :

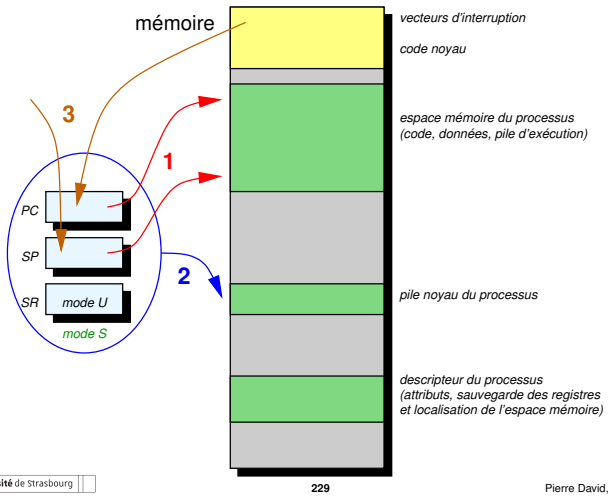
- tableau d'adresses de fonctions internes au noyau
- placé à une adresse fixée pour le processeur
- indexé par le numéro de l'interruption
  - exemple : interruptions clavier, interruptions disque, etc
- initialisé par le noyau au démarrage du système

Analogie avec les interruptions matérielles

Masquage des interruptions :

- empêche le processeur de consulter les interruptions
- mécanisme sélectif (selon le matériel)
  - exemple : masquer ce qui est moins prioritaire que l'interruption courante
  - masquage implicite de l'interruption courante
- masquage ⇒ contrôleur reste « interruptif »
- intérêt : empêcher le noyau de modifier une structure de donnée altérée par le traitement d'une interruption

Analogie avec les interruptions matérielles



Analogie avec les interruptions matérielles

Une fois le contexte (PC, SP, SR) initialisé, le processeur exécute le code du noyau :

1. (en assembleur) sauvegarde du reste du contexte CPU (registres généraux, etc.)
2. (en assembleur) mise en place d'un contexte de pile pour un appel de procédure en langage de haut niveau (ex: C)
3. (en assembleur) branchement à une adresse
4. (en C) vérification de la raison de l'interruption
   - interrogation des contrôleurs de périphériques pour identifier l'origine de l'interruption
5. (en C) action correspondant à l'interruption

Analogie avec les interruptions matérielles

Au retour :

- actions logicielles symétriques en fin d'exception (en C puis en assembleur)
- actions (en matériel) symétriques à la prise en compte de l'exception : instruction spéciale (IRET pour x86, RTE pour 68000)

Analogie avec les interruptions matérielles

Bilan :

	Interruptions	Signaux
Niveau	Matériel	Logiciel
Émetteur	Périphérique	Noyau
Destinataire	Processeur (noyau)	Processus
Masquage	Oui	Non

Plan

API POSIX

Problèmes avec `signal` (API v7) :

- pas de possibilité de masquage des signaux
  - ignorer un signal ⇒ signal perdu
- signal courant pas auto-masqué
  - dépendant de l'implémentation
  - fonction associée au signal interrompue par elle-même...
- action peut-être réinitialisée à l'action par défaut
  - dépendant de l'implémentation
  - deux appuis successifs sur CTRL C ⇒ patatras !

Gestion des signaux non fiable avec `signal` !

API POSIX – Primitive sigaction

POSIX : amélioration des signaux

```
int sigaction(int sig, const struct sigaction *new,
                       struct sigaction *old)
```

- `sigaction` remplace `signal`
- action décrite par une `struct sigaction` :

`sa_handler`	adresse de la fonction (ou `SIG_IGN` ou `SIG_DFL`)
`sa_mask`	masque pendant l'exécution de la fonction
`sa_flags`	comportements particuliers

- masquage implicite du signal reçu pendant l'exécution de la fonction
- pas de modification de l'action associée au signal
- permet de récupérer (ou pas) l'ancienne action

API POSIX – Primitive sigaction

`sa_handler`	adresse de la fonction (ou `SIG_IGN` ou `SIG_DFL`)
`sa_mask`	masque pendant l'exécution de la fonction
`sa_flags`	comportements particuliers

- `sa_handler` : même type de fonction que pour `signal`
- `sa_mask` : signaux supplémentaires à masquer pendant l'exécution de la fonction appelée
  - type `sigset_t` = champ de bits

  - signal $i \in$ ensemble $\Leftrightarrow$ bit i à 1
  - manipulation avec des fonctions de bibliothèque

`sigemptyset`	vide l'ensemble
`sigfillset`	remplit l'ensemble
`sigaddset`	ajout un signal à l'ensemble
`sigdelset`	retire un signal de l'ensemble
`sigismember`	teste si un signal fait partie de l'ensemble

API POSIX – Primitive sigaction

Exemple :

```
void fct(int sig)
{
    ...
}

int main(...)
{
    struct sigaction s;

    s.sa_handler = fct;
    s.sa_flags = 0;                    // souvent
    sigemptyset(&s.sa_mask);           // toujours
    sigaddset(&s.sa_mask, SIGQUIT);    // par ex.

    sigaction(SIGINT, &s, NULL);
    ...
}
```


API POSIX – Primitive sigprocmask

```
int sigprocmask(int comment, sigset_t *new, sigset_t *old)
```

- `sigprocmask` : masque ou démasque des signaux
- pendant le masquage, le signal n'est pas perdu
  - le signal sera traité lors du démasquage
  - attention : un seul bit pour la réception d'un signal
  - $\Rightarrow$ signal envoyé 2 fois : on ne le traitera qu'une fois
- le masque est spécifié par l'ensemble `new`
- valeurs possibles pour `comment` :

`SIG_BLOCK`	signaux $\in$ `new` ajoutés au masque courant
`SIG_UNBLOCK`	signaux $\in$ `new` retirés du masque courant
`SIG_SETMASK`	masque courant $\leftarrow$ `new`

API POSIX – Primitive sigprocmask

Pourquoi/quand utiliser `sigprocmask` ?

- Exemple déjà vu : le programme principal appelle la fonction `incrementer`

```
1  struct gros_compteur {
2          uint32_t grand;
3          uint32_t tresgrand;
4  } c = {0, 0};
5
6  void incrementer(void)
7  {
8          if (c.grand == UINT32_MAX) {
9                  c.grand = 0;
10                 c.tresgrand++;
11         } else {
12                 c.grand++;
13         }
14 }
```

- Problème de concurrence si la fonction associée à un signal utilise la variable `c`
- Il faut empêcher le signal d'être pris en compte lorsque `incrementer` s'exécute $\Rightarrow$ *masquer* le signal

API POSIX – Primitive sigprocmask

```
void incrementer (void)
{
        sigset_t masque, vieux;

        // masquer SIGINT
        sigemptyset(&masque);
        sigaddset(&masque, SIGINT);
        if (sigprocmask(SIG_BLOCK, &masque, &vieux) == -1)
                raler("masquage");

        // début de section critique
        if (c.grand == UINT32_MAX) {
                c.grand = 0;
                c.tresgrand++;
        } else {
                c.grand++;
        }
        // fin de section critique

        // démasquer SIGINT
        if (sigprocmask(SIG_SETMASK, &vieux, NULL) == -1)
                raler("demasquage");
}
```

API POSIX – Primitive sigpending

```
int sigpending(sigset_t *ensemble)
```

- `sigpending` : retourne l'ensemble des signaux en attente
- signaux en attente ⇒ ils sont masqués
  - les signaux sont reçus
  - mais ils sont masqués
  - donc ils sont en attente de traitement
- rappel : signal *i* reçu ⇔ bit *i* à 1
  - un signal n'est mémorisé qu'une seule fois
  - si un signal *i* est reçu *n* fois ($n > 1$), on ne garde qu'un bit

API POSIX – Primitive sigsuspend

```
int sigsuspend(const sigset_t *masque)
```

- `sigsuspend` : généralisation de `pause`
- attend l'arrivée d'un ou plusieurs signaux
- masque ou démasque temporairement (pendant `sigsuspend`) les signaux non désirés à l'aide de `masque`

API POSIX – Primitive sigsuspend

Pourquoi/quand utiliser `sigsuspend` ?

- Exemple : prise de décision suite à un événement

```
1  volatile sig_atomic_t condition = 0;
2
3  void signal_machin(int signum)
4  {
5      (void) signum;
6      condition = 1;        // l'événement est arrivé
7  }
8
9  int main(...)
10 {
11     signal(SIGmachin, signal_machin);
12     ...
13     if (!condition)       // si l'événement n'est pas arrivé
14         pause();          // alors attendre l'événement
15     ...
16 }
```

- Si le signal arrive entre 13 et 14 ⇒ attente éternelle...
  ⇒ il faudrait exécuter les lignes 13 et 14 en section critique tout en autorisant le signal à arriver pendant `pause`

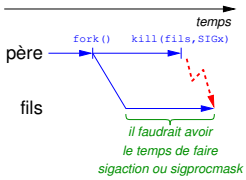
API POSIX – Primitive sigsuspend

```
1  int main(...)
2  {
3      sigset_t masque, vide;
4
5      sigemptyset(&vide);
6      sigemptyset(&masque);
7      sigaddset(&masque, SIGmachin);
8
9      ...
10     // section critique : signal SIGmachin masqué
11     if (sigprocmask(SIG_BLOCK, &masque, NULL) == -1)
12         raler("masquage");
13     if (!condition)           // l'événement ne peut pas arriver ici
14         sigsuspend(&vide);    // section critique levée pendant l'attente
15     // quand on est reveillé, on est toujours en section critique
16     ...
17 }
```

- les lignes 12 à 15 sont exécutées en section critique
  - pendant l'attente, la section critique est levée
  - on ne peut pas être interrompu entre 14 et 15
- ne pas abuser des sections critiques

API POSIX – Attention piège !

Attention aux signaux avec `fork` :



- père envoie un signal au fils ⇒ le fils doit se préparer
  - impossible car non déterminisme de `fork` : le père peut être remis sur le processeur avant que le fils n'ait eu le temps de démarrer
  - seule solution : préparer la réception des signaux dans le père (avant `fork`) afin d'en hériter dans le fils
- problème similaire si le fils envoie un signal au père :
  - si le père se prépare après `fork`, le fils démarrera peut-être trop rapidement !
  - solution : le père doit se préparer avant l'appel à `fork`

API POSIX – Bilan

- API v7 :
  - simple
  - insuffisante pour les cas réels
  - pratique pour le « quick and dirty »
  - utilisation pas à encourager
- API POSIX :
  - adaptée au monde réel
  - similaire aux interruptions matérielles (masquage)
  - plus complexe, plus riche
  - mais aussi plus robuste et plus fiable
  - usage à privilégier

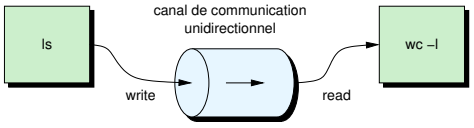
Plan

Plan

Introduction

Les tubes sont une des innovations majeures d'Unix

- exemple en Shell : `$ ls -l | wc -l`



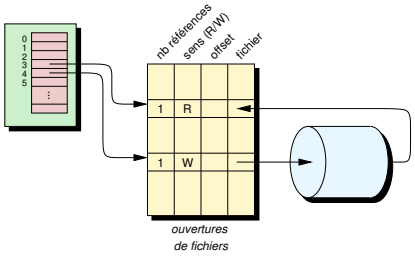
- le processus `ls` écrit dans le tube
  - utilisation de la primitive `write`
  - si `ls` écrit trop vite (⇒ tube plein), `write` attend
- le processus `wc` lit dans le tube
  - utilisation de la primitive `read`
  - si `wc` lit trop vite (⇒ tube vide), `read` attend
- les deux processus tournent en parallèle
  - synchronisation implicite
- pas de limitation sur la quantité de données transférée

Plan

Création des tubes

```
int pipe(int tube [2])
```

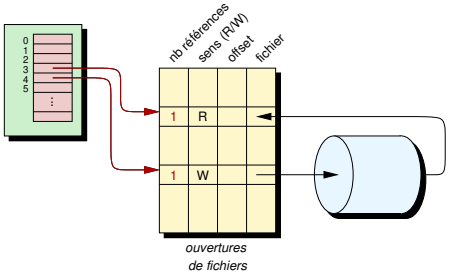
- création du tube et de deux descripteurs
- après appel à `pipe` :



- lecture via `tube[0]`
- écriture via `tube[1]`
- utilité : avec `fork`...

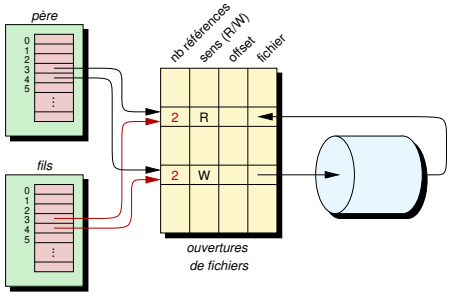
Exemple [1/6]

```
if (pipe(tube) == –1) ...
switch (fork()) {

case –1: ...
case 0:
        close(tube[0]);
        fils(tube[1]);
        close(tube[1]);
        exit(0);
default:
        close(tube[1]);
        pere(tube[0]);
        close(tube[0]);
        if (wait(NULL) == –1) ...
        ...
}
```



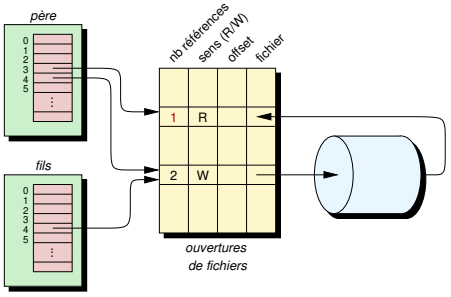
Exemple [2/6]

```
if (pipe(tube) == −1) ...
switch (fork()) {

case −1: ...
case 0:
        close(tube[0]);
        fils(tube[1]);
        close(tube[1]);
        exit(0);
default:
        close(tube[1]);
        pere(tube[0]);
        close(tube[0]);
        if (wait(NULL) == −1) ...
        ...
}
```



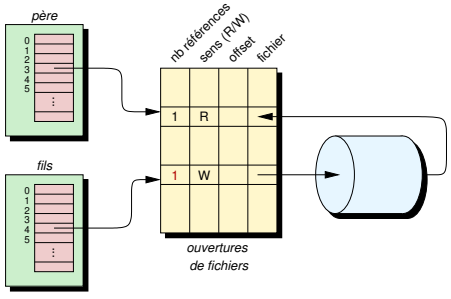
Exemple [3/6]

```
if (pipe(tube) == −1) ...
switch (fork()) {

case −1: ...
case 0:
        close(tube[0]);
        fils(tube[1]);
        close(tube[1]);
        exit(0);
default:
        close(tube[1]);
        pere(tube[0]);
        close(tube[0]);
        if (wait(NULL) == −1) ...
        ...
}
```



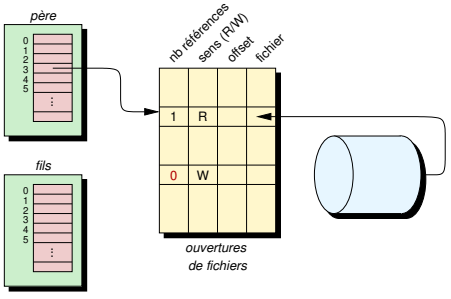
Exemple [4/6]

```
if (pipe(tube) == −1) ...
switch (fork()) {

case −1: ...
case 0:
        close(tube[0]);
        fils(tube[1]);
        close(tube[1]);
        exit(0);
default:
        close(tube[1]);
        pere(tube[0]);
        close(tube[0]);
        if (wait(NULL) == −1) ...
        ...
}
```



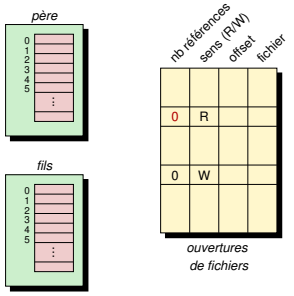
Exemple [5/6]

```
if (pipe(tube) == −1) ...
switch (fork()) {

case −1: ...
case 0:
        close(tube[0]);
        fils(tube[1]);
        close(tube[1]);
        exit(0);
default:
        close(tube[1]);
        pere(tube[0]);
        close(tube[0]);
        if (wait(NULL) == −1) ...
        ...
}
```



Exemple [6/6]

```
if (pipe(tube) == −1) ...
switch (fork()) {

case −1: ...
case 0:
        close(tube[0]);
        fils(tube[1]);
        close(tube[1]);
        exit(0);
default:
        close(tube[1]);
        pere(tube[0]);
        close(tube[0]);
        if (wait(NULL) == −1) ...
        ...
}
```



Plan

Règles de fonctionnement [1/3]

Règles particulières des tubes pour la lecture (`read`) :

- Primitive `read` bloquante
  - `read` renvoie 0 quand tube vide et plus aucun écrivain
- Lectures partielles
  - `read` renvoie ce qui est disponible dans le tube
  - vraisemblablement moins que ce qui est demandé
  - ⇒ être prêt à ce que `read` renvoie moins que demandé

Règles de fonctionnement [2/3]

Règles particulières des tubes pour l'écriture (`write`) :

- Écritures partielles
  - `write` limité par taille maximum des tubes
  - taille maximum variable (même sur le même système)
  - ⇒ être prêt à ce que `write` renvoie moins que demandé
  - exception : si taille demandée ≤ `PIPE_BUF`, alors `write` doit écrire ce qui est demandé
- Écriture dans un tube sans lecteur ⇒ problème !
  - renvoyer `-1` ne suffit pas
    - programmes mal écrits ne testent pas les erreurs...
  - envoi du signal `SIGPIPE` ⇒ terminaison du processus
  - exemple : `$ find / | ./a.out`
    - arrêt « prématuré » de `a.out` ⇒ arrêt automatique de `find` sans continuer à générer des données inutiles

Règles de fonctionnement [3/3]

Règles particulières pour le partage des tubes :

- Il peut y avoir plusieurs lecteurs et plusieurs écrivains
  - situation « normale » (exemple : juste après `fork`)
  - attention à la détection de « fin de fichier »
    - plus aucun écrivain...
    - ⇒ fermer les descripteurs dès qu'ils ne sont plus utilisés
- Écritures simultanées par plusieurs écrivains
  - taille ≤ `PIPE_BUF` : pas de mélange entre écrivains
  - taille > `PIPE_BUF` : mélange possible entre écrivains
  - POSIX spécifie que `PIPE_BUF` vaut 512 octets au minimum
    - `PIPE_BUF` = 512 (FreeBSD) ou 4096 (Linux)

Ne pas oublier...

Principes à respecter

- l'appel à `pipe` doit être fait avant l'appel à `fork`
  - sinon le tube n'est pas hérité
- fermer les descripteurs dès qu'ils ne sont plus utilisés
  - indispensable dans certains cas avec les tubes
  - ⇒ le faire systématiquement évite d'avoir à réfléchir !

Plan

Tubes nommés

```
int mkfifo(const char *path, mode_t mode)
```

- Tubes créés par `pipe` : tubes anonymes
  - doivent être créés par un ancêtre commun aux processus
  - héritage des descripteurs d'ouverture
- Tubes nommés : nom de fichier dans l'arborescence
- Nouveau type de fichier : « *fifo* »
  - avec `stat` : `S_IFIFO` et `S_ISFIFO()`
- Création avec `mkfifo`
  - accès ultérieur avec `open`, `read`, `write`, et `close`
  - ⇒ comme avec n'importe quel fichier régulier
- Règles de fonctionnement : cf tubes anonymes
  - ... après démarrage d'un lecteur et d'un écrivain
  - `read` ou `write` bloqué en attendant l'autre partie

Plan

Redirections

- Comment faire la redirection avec un tube ?
  - exemple : `$ ls -l | wc -l`
- Principe similaire aux redirections classiques
  1. créer un tube anonyme avant le `fork`
  2. dans le processus écrivain (ici `ls`) :
     - 2.1 `dup2(tube[1], 1)`
     - 2.2 `close(tube[0])`
     - 2.3 `close(tube[1])`
  3. dans le processus lecteur (ici `wc`) :
     - 3.1 `dup2(tube[0], 0)`
     - 3.2 `close(tube[0])`
     - 3.3 `close(tube[1])`

Plan

Plan

Mesure du temps

Comment le noyau mesure le temps ?

1. Obtenir l'heure au démarrage
2. Compter le temps qui passe

Mesure du temps

Comment obtenir l'heure au démarrage ?

- Solution 1 : lire l'heure sur le périphérique RTC
  - RTC : *Real Time Clock*
  - horloge matérielle
  - fonctionne sur batterie si courant coupé
- Solution 2 : demander l'heure au démarrage du noyau
  - solution « historique »
  - encore aujourd'hui (exemple : Raspberry PI)
- Solution 3 : laisser le noyau démarrer à une mauvaise date...
  - ... et utiliser le réseau pour synchroniser l'horloge
    - serveur NTP (*Network Time Protocol*)
    - protocole téléphonie mobile

Mesure du temps

Comment compter le temps qui passe ?

- Le noyau doit reprendre la main à intervalle régulier
- Assistance matérielle indispensable
  - mécanisme d'interruption périodique du processeur
  - historiquement : fréquence du secteur électrique
    - fréquence très stable (à l'inverse de la tension)
    - aux États-Unis : 60 Hz ⇒ 60 interruptions par seconde
    - en Europe : 50 Hz ⇒ 50 interruptions par seconde
  - actuellement : composant matériel basé sur le quartz
    - fréquence programmable par le noyau
    - en fonction de la configuration du noyau
    - entre 50 et 1000 fois par seconde (période entre 1 et 20 ms)
- À chaque interruption, incrémenter un compteur
- Lorsque le compteur atteint la fréquence
  - une seconde s'est écoulée
  - incrémenter l'heure du système

Unités de temps

Le noyau utilise deux unités de temps :

1. instant précis dans le temps
2. courte durée (ex : consommation de CPU)

Unités de temps – time_t

Instant précis (à la seconde) dans le temps : type `time_t`

- exemple : heure courante, date de fichier, etc.
- valeur : nombre de secondes depuis « The Epoch »
  - Epoch : premier janvier 1970, 0h0'0", UTC
  - UTC : temps universel coordonné
    (avant 1986 : GMT = heure du méridien de Greenwich)
- l'heure est conservée en UTC
  - indépendamment du fuseau horaire
- `time_t` historiquement sur 32 bits
  - bogue de l'an 2038 (nombre de secondes $\geq 2^{31}$)
  - solution : passer à 64 bits (ex : Linux $\geq$ 3.17)
  - nombreux formats de fichiers avec des dates sur 32 bits
- valeur de type `time_t`...
  - ... facile à tenir à jour par le noyau
  - ... pas facile à lire pour un humain
    - ce n'est pas le problème du noyau

Unités de temps – time_t

Conversion d'un `time_t` en une valeur intelligible :

- problème « intéressant », car prise en compte :
  - du fuseau horaire
  - de l'heure d'été et de l'heure d'hiver
  - des changements de dates des heures d'été et d'hiver
    - ex : pas de changement d'heure avant 1976 en France
  - des années bissextiles
  - des secondes intercalaires
    - variations de la vitesse de rotation de la Terre
    - ex : 1 seconde ajoutée après 23h59'59" le 31/12/2016
- ... mais ne concerne pas le noyau
- ⇒ à la charge des fonctions de bibliothèque
  - `localtime`, `asctime`, `strftime`, etc.

Unités de temps – clock_t

courte durée : type `clock_t`

- exemple : temps CPU consommé par un processus
- valeur : nombre de tops d'horloge (ou « *ticks* »)
- unité dépend de la configuration du noyau
  - POSIX fournit la primitive `long sysconf(int paramètre)`
  - `paramètre` : paramètre de configuration interrogé
  - exemple : `freq = sysconf(_SC_CLK_TCK)`
    ⇒ donne le nombre de tops d'horloge par seconde
- mesure de la consommation CPU :
  - à chaque interruption d'horloge, le noyau incrémente le compteur du processus courant
  - ⇒ consommation approximative

Plan

Heure courante

```
time_t time(time_t *heure)
int stime(time_t *heure)
```

- `time` récupère l'heure courante
  - comme valeur de retour (ou -1)
  - et à l'adresse indiquée
- `stime` modifie l'heure courante
  - primitive réservée à l'administrateur

Note : l'heure courante est parfois appelée « *wall clock* »
(i.e. l'heure qu'on peut lire sur l'horloge murale)

Heure courante – gettimeofday

```
int gettimeofday(struct timeval *tv, struct timezone *tz)
```

- ajout de l'U. de Berkeley
- précision accrue
- contenu de la `struct timeval` :

`time_t`	`tv_sec`	nombre de secondes depuis « The Epoch »
`suseconds_t`	`tv_usec`	nombre de micro-secondes

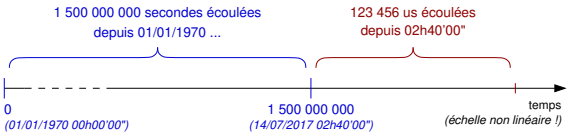
- attention : ce n'est pas parce qu'il y a un champ dont l'unité est la μs que la granularité de la mesure du temps est la μs
- `time` est maintenant devenue une fonction de bibliothèque qui appelle la primitive `gettimeofday`

Heure courante – gettimeofday

Exemple de `struct timeval` :

`tv_sec`	1 500 000 000
`tv_usec`	123 456

$\Leftrightarrow$ 14/07/2017 à 2h40'00" et 123456 μs UTC



Heure courante

Exemple d'utilisation :

```
time_t heure;
struct tm *tm;
char *s1, s2[MAX];
size_t n;

heure = time(NULL);

tm = localtime(&heure);

s1 = asctime(tm);
printf("maintenant : %s\n", s1);

n = strftime(s2, MAX, "le %d/%m/%Y a %H:%M:%S", tm);
if (n == 0)
    printf("s2 est trop petite\n");
else
    printf("maintenant : %s\n", s2);
```

- « primitive système » : `time`
- fonctions de bibliothèque : `localtime`, `asctime` et `strftime`

Plan

Temps CPU

```
clock_t times(struct tms *buf)
```

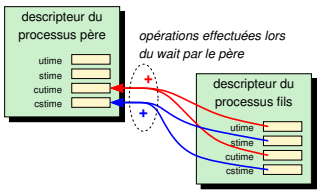
- place la consommation CPU à l'adresse pointée par `buf`
  - contenu de la `struct tms` :

`tms_utime`	consommation CPU du processus en mode utilisateur
`tms_stime`	consommation CPU du processus en mode système
`tms_cutime`	consommation CPU cumulée des fils en mode utilisateur
`tms_cstime`	consommation CPU cumulée des fils en mode système

  - tous les champs sont de type `clock_t`
- retourne le temps réellement écoulé depuis un moment arbitraire dans le passé (ou -1)
  - typiquement le démarrage du noyau
  - pas très utile
  - peut déborder la taille allouée à un `clock_t`
  - bref : valeur de retour à ignorer (sauf pour test d'erreur)...

Temps CPU

Récupération de la consommation CPU :



- état « zombie » permet de transmettre :
  - code de retour (primitive `exit`)
  - consommation CPU agrégée en mode utilisateur
  - consommation CPU agrégée en mode système
- informations transmises lors du `wait` par le père
- on ne peut pas avoir la consommation d'un fils non terminé

Temps CPU

```
int getrusage(int qui, struct rusage *res)
```

- primitive `times` limitée à la consommation CPU
  - ⇒ besoin d'une primitive plus générale pour l'ensemble des ressources consommées par un processus
- paramètre `qui` :

`RUSAGE_SELF`	consommation du processus lui-même
`RUSAGE_CHILDREN`	consommation cumulée des fils

- exemples (non exhaustifs) de champs de `struct rusage` :

`ru_utime`	CPU en mode utilisateur
`ru_stime`	CPU en mode système
`ru_maxrss`	mémoire maximum utilisée
`ru_inblock`	nombre de lectures disques
`ru_outblock`	nombre d'écritures disques

Note : seuls les deux premiers sont normalisés par POSIX

- valeurs remontées du fils vers le père lors du `wait`

Plan

Dates des fichiers

```
int utime(const char *path, const struct utimbuf *ut)
int utimes(const char *path, const struct timeval tv[2])
```

- Rappel des attributs des fichiers :

`st_mtime`	date de dernière modification des données
`st_ctime`	date de dernière modification de l'inode
`st_atime`	date de dernier accès

- Primitive `utime` : modification des dates d'un fichier
- Champs de la `struct utimbuf` :

`actime`	date de dernier accès
`modtime`	date de dernière modification des données

- Pas de dernière modification de l'inode ⇒ pourquoi ?
- Primitive `utimes` : idem `utime`, mais avec des `struct timeval` (comme `gettimeofday`)
  - `tv[0]` : dernier accès
  - `tv[1]` : dernière modification des données

Plan

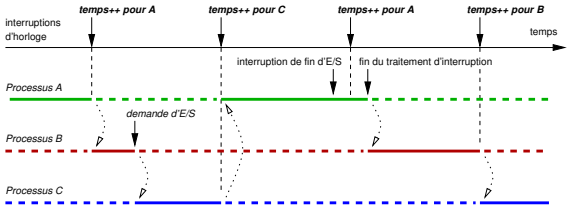
Alarmes de processus

Voir chapitre sur les signaux

Plan

Précision de la mesure du temps

La précision de la mesure de la consommation de temps CPU est approximative



- comptabilisation du temps pour un processus lorsque l'horloge interrompt le CPU
  - pas de prise en compte du temps de B avant son E/S
    $\Rightarrow$ temps imputé à C
  - l'interruption disque est traitée alors que A est sur le CPU
    $\Rightarrow$ temps pour traitement d'E/S de B imputé à A
- $\Rightarrow$ faire plusieurs mesures