S4 2019-2020

Nathan TONNELLE

**Système d'exploitation**

INFO0403

Table des matières

[Introduction Système d'exploitation 5](#_Toc31623697)

[Définition d'un système d'exploitation 5](#_Toc31623698)

[Fonctionnement 5](#_Toc31623699)

[Rôle 5](#_Toc31623700)

[Exemple : disque 5](#_Toc31623701)

[Partage de ressource 5](#_Toc31623702)

[Exemple : 5](#_Toc31623703)

[Historique 5](#_Toc31623704)

[Les processus 7](#_Toc31623705)

[Définition 7](#_Toc31623706)

[Plusieurs 7](#_Toc31623707)

[La concurrence : 7](#_Toc31623708)

[Programmation système 8](#_Toc31623709)

[Exemple : 8](#_Toc31623710)

[Quelques appels system 8](#_Toc31623711)

[Les processus 9](#_Toc31623712)

[Programme : 9](#_Toc31623713)

[Processus : 9](#_Toc31623714)

[La création de processus survient lorsque : 9](#_Toc31623715)

[Identification 9](#_Toc31623716)

[Création : 9](#_Toc31623717)

[pid\_t fork(): 10](#_Toc31623718)

[Etats des processus 10](#_Toc31623719)

[Ordonnancement 10](#_Toc31623720)

[Motivations : 10](#_Toc31623721)

[Equité en temps : 10](#_Toc31623722)

[Définition : 10](#_Toc31623723)

[Type d'ordonnanceur 11](#_Toc31623724)

[Non préemptif 11](#_Toc31623725)

[Préemptif 11](#_Toc31623726)

[Non préemptif 11](#_Toc31623727)

[Préemptif 11](#_Toc31623728)

[Exécution d'un programme 11](#_Toc31623729)

[Programmation Système 11](#_Toc31623730)

[Fonction principale 12](#_Toc31623731)

[Schéma mémoire d'un processeur 12](#_Toc31623732)

[La mémoire d'un processus est composée de : 12](#_Toc31623733)

[Allocation dynamique 12](#_Toc31623734)

[Récupérer et modifier les variables d'environnement 13](#_Toc31623735)

[Exemple de variables d'environnement : 13](#_Toc31623736)

[Exercice : 13](#_Toc31623737)

[Limitation ressources 13](#_Toc31623738)

[Exemple : 13](#_Toc31623739)

[Identifier un processus 14](#_Toc31623740)

[Création de processus 14](#_Toc31623741)

[Exemple : 14](#_Toc31623742)

[Terminaison de processus 14](#_Toc31623743)

[Exemple : 15](#_Toc31623744)

[Chargement d'un programme 15](#_Toc31623745)

[La gestion Mémoire 15](#_Toc31623746)

[Quand la mémoire est insuffisante 16](#_Toc31623747)

[Pour maintenir tous les processus actifs 16](#_Toc31623748)

[2 approches 16](#_Toc31623749)

[Le va et vient 16](#_Toc31623750)

[Gestion mémoire (va et vient) 16](#_Toc31623751)

[Gestion par liste chainée 16](#_Toc31623752)

[Gestion mémoire (va et vient) 17](#_Toc31623753)

[Plusieurs stratégies 17](#_Toc31623754)

[First fit : 17](#_Toc31623755)

[Next fit : 17](#_Toc31623756)

[Best fit : 17](#_Toc31623757)

[Wost fit : 17](#_Toc31623758)

[La mémoire virtuelle 17](#_Toc31623759)

[Technique utilisée : la pagination 17](#_Toc31623760)

[Le défaut de page 17](#_Toc31623761)

[La pagination 17](#_Toc31623762)

[Algorithmes de pagination 18](#_Toc31623763)

[Belady 18](#_Toc31623764)

[Algorithme de la seconde chance 18](#_Toc31623765)

[Les fichiers 19](#_Toc31623766)

[Problèmes posés à un processus 19](#_Toc31623767)

[Solution 19](#_Toc31623768)

[Type de fichiers 19](#_Toc31623769)

[Exemples 19](#_Toc31623770)

[Nom des fichiers 19](#_Toc31623771)

[Accès aux fichiers 19](#_Toc31623772)

[Attributs des fichiers 19](#_Toc31623773)

[Utilisation des répertoires 19](#_Toc31623774)

[Exemple 20](#_Toc31623775)

[Partitionnement 20](#_Toc31623776)

[Disque 20](#_Toc31623777)

[Une partition 20](#_Toc31623778)

[Implémentation des fichiers sur le disque 20](#_Toc31623779)

[Mémorisation des blocs libres 20](#_Toc31623780)

[La fermeture d'un fichier 20](#_Toc31623781)

[Lecteur d'un fichier 20](#_Toc31623782)

[L'écriture d'un fichier 20](#_Toc31623783)

[Exemple 21](#_Toc31623784)

[Se déplacer dans le fichier 21](#_Toc31623785)

[Se déplacer dans un fichier 21](#_Toc31623786)

[Supprimer un fichier 21](#_Toc31623787)

[Obtenir des informations sur un fichier 21](#_Toc31623788)

[Exercice 22](#_Toc31623789)

[Manipulation des répertoires 22](#_Toc31623790)

[Exemple 22](#_Toc31623791)

[Autres opérations sur répertoire 23](#_Toc31623792)

[Exemple 23](#_Toc31623793)

[La concurrence 24](#_Toc31623794)

[Exemple 24](#_Toc31623795)

[Alternance de Peterson 24](#_Toc31623796)

[La concurrence conclusion 25](#_Toc31623797)

[L'interblocage 26](#_Toc31623798)

[Détection d'interblocage 26](#_Toc31623799)

[Les tubes 28](#_Toc31623800)

[Création d'un tube 28](#_Toc31623801)

[Exercice mise en application 28](#_Toc31623802)

[Création de tubes 29](#_Toc31623803)

[Exemple : 29](#_Toc31623804)

[Utilisation de tube vers des programmes externes 29](#_Toc31623805)

[Exemple 30](#_Toc31623806)

[Architecture de tuber 30](#_Toc31623807)

[Les signaux 31](#_Toc31623808)

[Définition 31](#_Toc31623809)

[Génération d'un signal 32](#_Toc31623810)

[Quelques signaux 32](#_Toc31623811)

[Mécanismes 32](#_Toc31623812)

[Envoyer un signal 32](#_Toc31623813)

[Manipulation d'un ensemble de signaux 33](#_Toc31623814)

[Masquer un jeu de signaux 34](#_Toc31623815)

[Exemple : 34](#_Toc31623816)

[Recupérer des signaux bloqués 34](#_Toc31623817)

[Mise en place d'un gestionnaire 34](#_Toc31623818)

[Mécanisme de communication inter processus 34](#_Toc31623819)

[Mécanisme "FiFo" 35](#_Toc31623820)

[Création : 35](#_Toc31623821)

[Exemple : 35](#_Toc31623822)

# Introduction Système d'exploitation

## Définition d'un système d'exploitation

* Interface entre un utilisateur et une machine physique
* Dédié à la gestion des périphériques

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SGBD | Appli de calcul scientifique | … | Traitement de texte | **Application** |
| Interpréteur commande | Compilateur | … | Edition de fichier | **Programme Système** |
| Système d'exploitation | | | |
| Langage Machine | | | | **Matériel** |
| Micro Architecture | | | |
| Périphérique physique | | | |

## Fonctionnement

* Portion de code en mode noyau
* Protégé des modifications apportées par un utilisateur

## Rôle

* Gestion de ressources
* Il offre une vision étendue de la machine

### Exemple : disque

* Vision "haut niveau" : Système de fichiers
* Vision "bas niveau" : surface d'enregistrement tête de lecture, moteur, blocs
* Gestion de ressources :
  + Permettre le partage d'accès
  + Solution pour partager sera dépendante des ressources
  + Equitable
  + Récupérer la connaissance des accès concurrents sur une ressource

### Partage de ressource

Peut se faire sur 2 dimensions :

* + Le temps
  + L'espace

#### Exemple :

* Mémoire : Espace
* Processeur : Temps
* Un ensemble de Processeurs : Temps et Espace

## Historique

* Machine de Charle Babbage (~1800)

Non fonctionnel : Mécanique pas suffisamment précise

* 1ère génération 1945-1955
  + Tube à vide et de tableau d'interrupteur
    - 1950 : introduction de la carte perforées
* 2eme génération 1955-1965
  + Introduction du transistor
    - Accroissement de la fiabilité
    - Introduction du traitement par lots
  + Soumission d'un travail en 4 étapes :
    - Ecriture du programme
    - Soumission à l'opération
    - Introduction des cartes perforées
    - Compilation en fortran
  + Traitement par lots (actuellement : script)
* 3eme génération 1965-1980
  + 2 architectures incompatibles en termes de matériel (unité de données : caractère/mot)
  + IBM : introduction System 360 (-> compatibilité ai niveau logiciel)
  + Introduction Multiprogrammation
  + Introduction Spodage
  + Introduction Temps partagé
  + Multitâche (simple/préemptif)
  + Début d'Unix (-> System V, -> BSD)
* 1980-?? (2010) Ordinateurs personnels
  + Intel 8080
  + MSDOS (Bil Gates Tim Paterson)
  + Lisa, Mac Intosh (Steve Jobs (-> IHM))
  + Windows (d'abord comme surcouche à MSDOS puis comme SE à part entière)
  + Développement UNIX (->Linux)
  + Système d'exploitation Distribué
* 2010-today
  + Dématérialisation
  + Big-Data

# Les processus

## Définition

* Programme en cours d'exécution
* Espace d'adressage propre :
  + Code du programme exécutable
  + Données
  + Ensemble de registres
* Table de processus
  + Un processeur est lié à un autre processus par une relation de parenté
    - Arborescence de processus avec un ancêtre commun : 'int'

## Plusieurs

Processus en concurrence pour l'accès à une ressource :

## La concurrence :

Résoudre le problème de concurrence entre plusieurs processus c'est :

* Sureté de l'exécution de la portion de code durant laquelle le processus accède à la ressource s'appelle la section critique. A un moment donné, au plus un processus exécute en section critique
* Vivacité : un processus qui demande à accéder à sa section critique finit par pouvoir l'exécuter
  + Absence d'interblocage

# Programmation système

## Exemple :

cpt =read(df,&tampon, nboctet);



Retour appelant

Déroulement Noyau

5 Place de code de read() en mémoire



11 Incrémentation CO

4 Appel read

3 Empiler nombre octets

2 Empiler tampon

1 Empiler df



Noyau du SE

Colle de l'appel Sys

Branchement



## Quelques appels system

* Gestion de processus fork(), waitpid(), execve(), …)
* Gestion de fichiers open(), close(), read(), write(), stat(),…)
* Gestion du système de fichier mkdir(), rmdir(), chdir(), link(), unlink(),…)

# Les processus

## Programme :

Suite d'instructions

## Processus :

Programme en cours d'exécution.

Sur un processeur, un utilisateur non averti à l'impression que plusieurs processus s'exécutent en même temps.

La simultanéité résulte du fait que des lots de temps suffisamment courts sont alloués à chaque processus.

On a donc cette idée de "continuité".

La simultanéité réelle provient du fait qu'on a plusieurs processeurs.

* Cette "presque simultanéité" s'appelle la multiprogrammation

### La création de processus survient lorsque :

* Requête utilisateur
* Traitement par lot
* Initialisation du système
* Utilisation de l'appel système fork()

### Identification

* pid (cf commande ps)

Identifier l'utilisateur d'un processus

UID (réel/effectif/sauvé)

Identifier le groupe de l'utilisateur du processus

GID (réel/effectif/sauvé)

### Création :

Les processus sont créés par l'appel system fork() et se terminent :

* De manière normale
* Avec une erreur
* Avec une erreur fatale
* A cause d'un autre processus (via des signaux)

Premier processus créé : int

* Création de processus à l'initiative d'autres processus
* Relation de parenté entre processus (-> Arborescence)
* Visualisation des processus (ps, top, pstree)
* Suppression de processus (kill -9)

Appel system fork() :

pid=342

fork();

pid=??

pid=342

a==0

a==0

a==0

pid: 300

pid: 347

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

A==0

B==0

A==0

B==0

{

int a=0;

int b=0;

fork();

printf("a %d, b %d\n",a,b);

exit(0);

}

## pid\_t fork():

Duplique l'espace mémoire du courant et crée un fil d'exécution dont l'exécution démarre à l'instant où le processus s'est dupliqué.

La valeur retournée (de type pid\_t) est l'identifiant du processus créé dans le processus créant (ou père) et 0 dans le processus créé.

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

{

int a=0;

int b=0;

if(fork()==0)

{

a=1;

b=432;

exit(0);

}

printf("a %d, b %d \n",a,b);

exit(0);

}

## Etats des processus

Sommeil

fork()

Evénement

Interruption



Création



Zombie

Exécution



FIN



SIGCONT

SIGSTOP



Arrêt

## Ordonnancement

### Motivations :

#### Equité en temps :

* Critère pour un "bon" ordonnancement
* Pourcentage utilisation CPU
* Nombre de processus exécutés en un temps donnée
* Temps nécessaire pour exécuter un processus
* Temps d'attente d'un processus dans la file prêt

### Définition :

L'ordonnanceur est un algorithme qui élit un processus. Ce processus a le privilège d'accéder au CPU.

## Type d'ordonnanceur

### Non préemptif

Sélectionne un processus qui s'exécute jusqu'à ce qu'il bloque ou se termine

### Préemptif

Sélectionne un processus qui s'exécute pendant un délai donné

Si le processus est toujours en cours d'exécution, il est suspendu (Etat sommeil) et un autre processus est alors choisi.

### Non préemptif

Ordonnancement

* FIFO : exécution des processus dans l'ordre de soumission
* SJF : exécution des processus les plus courts d'abord
* LJF : exécution des processus les plus longs d'abord

### Préemptif

Ordonnancement :

* SRTN : au plus court reste d'abord
* Roud Robin : on exécute chaque processus durant un laps de temps donné en suivant l'ordre de soumission
* Priorité : on utilise Round Robin par file de priorité

## Exécution d'un programme

* On utilise la famille d'appels Système exec :

Ces appels système chargent en mémoire du processus le contenu d'un programme stocké sue Disque.

int execv (const char \* app, const char \* argv[])

App : chemin complet du programme

Argv : tableau des différents arguments

int execl (const char \* app, const char \* arg1, const char \* arg2, …)

App : chemin complet de l'application

Arg1 ,…,n : arguments à passer. La liste se termine par NULL

Exercice 1 :

Ecrire un programme qui exécute "ls-al"

int main()

{

char \* app="/bin/ls";

char \* arg[]={"ls","-al"};

execv(app,arg);

exit(0);

}

Exercice 2 :

Ecrire un programme qui exécute "ps" puis "ls-al"

int main(){

{

char \* app="/bin/ls";

char \* arg[]={"ls","-al"};

if(fork()==0){

execv(app,arg);

exit(0);

}

}

## Programmation Système

Langage C

## Fonction principale

Int main(int argc, char \* argv[])

argc : nombre de paramètres à la fonction

argv : tableau de chaine de caractères des paramètres

$ ./programme toto 423 arg1

Int t=512;

Char \* s=malloc(…)

sprintf(s,"%d",t);

argc =4

argv[0]="./programme"

argv[1]="toto"

argv[2]="423"

argv[3]="arg1"

## Schéma mémoire d'un processeur

### La mémoire d'un processus est composée de :

* Le segment texte : ce sont les différentes instructions exécutées par le processeur. Le segment est en lecture seule et peut être éventuellement partagé.
* Segment de données initialisées on y retrouve les données initialisées

Int max=100;

* Segment de données non initialisées, elles sont normalement initialisées à 0 ou à NULL

Int sum[100];

* Pile : sert à la sauvegarde d'informations en particulier lors de l'appel d'une fonction.
* Tas : sert à l'allocation dynamique de mémoire
* Arguments et les variables d'environnement

La commande size fournit la taille des différents segments

# Allocation dynamique

* Void \* malloc (size\_t T)

Reserve un emplacement de taille T dont l'adresse du premier octet est retourné.

Void \* ptr;

ptr = malloc (10);

* Void \* calloc (int n, size\_t T)

Fonctionne comme malloc, reserve de la place dans le tas pour contenir n objets de taille T, soit n x T octets

* Void \* realloc (void \* ptr, size\_t T)

Change la taille de la zone de mémoire précédemment allouée. La nouvelle taille est T.

* Void free(void \* p);

Désalloue la zone mémoire dont la première case est située à l'adresse p

# Récupérer et modifier les variables d'environnement

## Exemple de variables d'environnement :

* HOME : chaine de caractère contenant le répertoire de travail de l'utilisation
* LOGNAME : nom de login d'utilisateur
* PATH : ensemble de chemin désignant les répertoires contenant les exécutables
* TERM : type de terminal

1. Char \* getenv(const char \* name);
2. Int putenv(const char \* name);
3. Int setenv(const char \* name, const char \* value, int valur);
4. Void unsetenv(cast char \* name);
5. Récupérer la valeur de la var. d'env name
6. Déclarer une variable d'env name
7. Positionner la variable d'env name à la valeur value
8. Retirer la var d'env name

## Exercice :

Créer une variable d'environnement TRUC. Positionnez sa valeur à "c'est les vacances". Afficher sa valeur, puis supprimer cette variable d'environnement.

Int main (int argc, char \* argv[])

{

putenv("TRUC");

setenv("TRUC","C'est les vacances");

printf("%s\n",getenv("TRUC"));

unsetenv("TRUC");

exit(0);

}

## Limitation ressources

### Exemple :

RLIMIT\_CORE taille max du fichier Core crée en cas d'erreur fatale

RLIMIT\_CPU maximum temps CPU

RLIMIT\_DATA taille max segment de données (init et non init)

RLIMIT\_FSIZE taille max des fichiers ouverts

RLIMIT\_NOFILE nombre max de fichiers ouvert

RLIMIT\_NPROC nombre max de processus fils

RLIMIT\_STACK taille max de la pile

…

struct rlimit{

rlim\_t rlim\_cur; //limite courante

rlim\_t rlim\_max; //maximum pour le système

}

Int getrlimit(int ressource, struct rlimit \* ptr);

Int setrlimit(int ressource, const struct rlimit \* ptr);

## Identifier un processus

pid\_t getpid(); retourne l'id du processus

pid\_t getppid(); retourne l'id du processus père

pid\_t getuid(); retourne l'id de l'utilisateur

pid\_t geteuid(); retourne l'id de l'utilisateur effectif

pid\_t getgid(); retourne l'id du groupe

pid\_t getegid(); retourne l'id du groupe effectif

## Création de processus

pid\_t fork(); créer un processus fils. La valeur de 0 dans le processus fils

Retour est : le pid du processus crée dans le processus père

### Exemple :

int gbb=6;

int main()

{

int var;

pid\_t pid;

var =88;

printf("avant fork\n");

if((pid=fork())==0)

{

glob++;

var++;

printf("%d,%d\n",glob, var);

exit(0);

}

sleep(2);

printf("%d,%d\n", glob, var);

exit(0);

}

## Terminaison de processus

* Void exit(int s);

Effectue un "ménage" et quitte le processus avec le statut s

* Void \_ext(int s);

Quitte le processus et retourne au système avec le statut s

* Int atexit(void (\*fonc) (void));

Spécifie les opérations à réaliser à la terminaison du processus **(voir ce qu'est qu'un pointeur de fonction)**

Pid\_t wait(int \* stat)

Attend la fin d'un processus fils. La variable stat contient la valeur du statut de terminaison SD1 du processus fils. L'appel à wait est bloquant.

### Exemple :

int gbb=6;

int main()

{

int var;

pid\_t pid;

var =88;

printf("avant fork\n");

if((pid=fork())==0)

{

glob++;

var++;

printf("%d,%d\n",glob, var);

exit(0);

}

Wait(NULL);

printf("%d,%d\n", glob, var);

exit(0);

}

## Chargement d'un programme

* Famille de fonction exec (cf cours plus haut)

## La gestion Mémoire

* Monoprogrammation

|  |
| --- |
| ROM (Read Only Memory) |
| Programme utilisateur |

|  |
| --- |
| (ROM) gestion de périphérique |
| Programme utilisateur |
| System d'exploitation |

RAM



|  |
| --- |
| Programmation utilisateur |
| System d'exploitation |

(Random Access Memory)

RAM

RAM

* Multiprogrammation

RAM

|  |
| --- |
| Prog 1 |
| Prog 2 |
| Prog 3 |
| … |
| Prog n |

|  |
| --- |
|  |
|  |
|  |
| … |
|  |



### Quand la mémoire est insuffisante

#### Pour maintenir tous les processus actifs

* Sauvegarde des processus sur disque
* Chargement dynamique

#### 2 approches

Le va et vient :

Chaque processus est considéré dans son intégralité

Exécution et sauvegarde sur disque

La mémoire virtuelle :

Un processus est découpé en page

Il peut donc s'exécuter meme s'il n'est que partiellement en mémoire

#### Le va et vient

* Chargement complet du processus
* Sauvegarde complète éventuelle en cas de manque de mémoire
* Au chargement :

Nécessité de localiser correctement le processus

* Possibilité d'apparition de "trous" mémoire 🡪 Technique de compactage (couteux !)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **P1** | | **P4** | **P5** | | |
|
| **P2** |  | **P7** |  | **P6** |
| **P9** |
| **P3** | | **P8** | |
|

Quand la taille des processus est fixe, le SE alloue exactement la place Mémoire nécessaire

Quand un processus doit augmenter sa taille :

* Soit il y a de l'espace libre contiguë à l'espace mémoire du processus (pas de soucis !)
* Il est contigu à d'autres processus : il faut alors déplacer le processus
* S'il n'y a pas suffisamment de place : attente ou terminaison

#### Gestion mémoire (va et vient)

* Tableau de bits
* La mémoire est répartie en unités d'allocation (UA)
* Chaque UA possède une taille fixe comprise entre quelques octets et plusieurs ko
* A chaque UA correspond un bit dans le tableau

0 : vide 1 : sinon

* Problème : trouver la bonne taille !

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| A |  | B |  | C |  |  |  |
| 0 | 8 | 12 | 16 | 18 | 22 | 24 | UA |

1111 1111

0000 1111

0011 1100

### Gestion par liste chainée

Liste chainée des segments mémoire

Chaque entrée indique :

* L'état du segment (L: libre, 0 : occupé)
* L'adresse à laquelle il débute
* Sa longueur

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 , 0 , 8 |  | L , 8 , 4 |  | 0 , 12 , 4 |

### Gestion mémoire (va et vient)

Où placer les processeurs ?

Dans quel espace libre ?

Peut-on optimiser la taille des espaces libres ?

Optimiser la recherche d'un espace libre ?

#### Plusieurs stratégies

##### First fit :

On parcourt la mémoire depuis le début pour trouver le premier emplacement qui suffit en termes de place

##### Next fit :

On parcourt la mémoire depuis la dernière position pour trouver le premier emplacement qui suffit en termes de place

##### Best fit :

On parcourt l'espace mémoire en cherchant l'espace mémoire dont la taille s'approche le plus de la taille du processus

##### Wost fit :

On parcourt l'espace mémoire pour trouver le plus grand espace libre, si la taille est suffisante, on place le processus "on multiplie "

### La mémoire virtuelle

* La taille d'un programme (instruction, données, pile, …) peut être supérieure à la capacité mémoire physique
* Le système d'exploitation conserve en mémoire des parties d'un programme en cours d'exécution le reste est stocké sur disque

### Technique utilisée : la pagination

|  |  |
| --- | --- |
| Mémoire Rapide de taille m |  |
|  | m <<< M |
| Mémoire Lente de taille M (∞) |  |

Un ensemble de processus est découpé en "pages" mémoire

On doit gérer les échanges de pages entre mémoire rapide et mémoire lente

Pour fonctionner, le système exécute les instructions d'un processus si une partie de ses pages sont en mémoire rapide.

#### Le défaut de page

C'est un accès à une page absente de la mémoire rapide.

Dans ce cas : le SE choisit une page présente en mémoire rapide, l'écrit sur la mémoire lente, et charge la page qui faisait défaut à sa place.

### La pagination

On considère des pages mémoire (qui correspondent à des morceaux de processus).

Une exécution du problème de pagination correspond à une séquence de pages appelées.

Un algorithme est chargé du chargement / déchargement de page entre la mémoire lente et la mémoire rapide.

#### Algorithmes de pagination

* FIFO (First In First Out)
* LIFO (Last in First Out)
* LRU (Least Recently Used)
* FWF (Flush When Full)
* 2ème Chance
* Belady (Théorique)

##### Belady

Algorithme Optimal !

Lors d'un défaut de page, il choisit pour page à décharger (évincer) la page qui sera utilisée le plus tard possible.

6=1,3,4,2,7,2,3,4,1,8,5,6,4,3,8,5,7,4,3,2,1,2,4,1,3,7 On considère une mémoire rapide pouvant contenir 4 pages

##### Algorithme de la seconde chance

* On a une liste chainée des pages mémoire
* A chaque "case" de page mémoire on associe un bit
* Lorsqu'une page est attribuée / chargée, son bit est placé à 1
* Lors d'un défaut de page, à l'aide d'une "tête de lecture" on parcourt les bits de cases de page mémoire :

Si on trouve un "1" on bascule le bit à 0 et on passe à la case suivante.

Si on trouve un bit à 0, on charge la page qui faisait défaut dans cette case (et on repositionne le bit à 1). L'emplacement de la tête de lecture est sauvegardé.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Durée | Soumissions | Priorité |
| J1 | 6 s | t=4 | 5 |
| J2 | 10 s | t=0 | 3 |
| J3 | 8 s | t=2 | 4 |
| J4 | 7 s | t=1 | 4 |
| J5 | 9 s | t=0 | 3 |

|  |  |
| --- | --- |
| FIFO | Non préemptif |
| SJF |
| LJF |
| SRTN |  |
| RR Q=2s | Préemptif |
| Priorité Q=2s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| FIFO (First In First Out) (date de soumission par ordre croissant) | | | | |
| J2 | J5 | J4 | J3 | J1 |



TMA (Temps Moyen Attente) =

TME (Temps Moyen Exécution) =

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SJF (Short Job First) (1ère date de soumission, puis durée par ordre croissant) | | | | |
| J5 | J1 | J4 | J3 | J2 |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| LJF (Longest Job First) (la plus longue durée puis décroissant) | | | | |
| J2 | J5 | J3 | J4 | J1 |



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SRTN (Shortest Time Remaining Next) ( | | | | | |
| J5 | J4 | J1 | J3 | J5 | J2 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RR (Round Robin) (compliqué) | | | | | | | | | | | | | | | | | | | | | |
| J2 | J5 | J4 | J3 | J1 | J2 | J5 | J4 | J3 | J1 | J2 | J5 | J4 | J3 | J1 | J2 | J5 | J4 | J3 | J2 | J5 |



TMA= (SOMME(Temps début exe – Temps soumission))/nb processus

TME= (SOMME(Temps fin exe – Temps soumission))/nb processus

# Les fichiers

## Problèmes posés à un processus

* Mémoriser une grande quantité d'informations
* Persistance des informations en mémoire
* Accès "simplifié" à ces informations par d'autres processus

## Solution

Utiliser des fichiers

## Type de fichiers

* Fichier ASCII
  + Ligne de texte
  + Editable via un éditeur de texte
* Fichiers Binaires

### Exemples

.jpg Binaire

.html ASCII

.docx binaire

.pdf binaire

### Nom des fichiers

* Abstraction des données techniques qui permettent de référencer le contenu du fichier sur le disque
* Unix : l'extension est indicative mais pas imposée
* Dos : extension était définie par le système
  + C-> code source C
  + Jpg -> image
  + Rip -> archive

## Accès aux fichiers

* Accès séquentiel
  + Lecture du périphérique de stockage du début vers la fin
* Accès aléatoire
  + Accès direct à n'importe quel fichier

Aujourd'hui quasiment tous les périphériques d'enregistrement sont en accès aléatoire

## Attributs des fichiers

Informations complémentaires concernant un fichier du système:

* Protection sur le fichier
* Propriétaire
* Date de création
* Taille
* Date de modification
* …

## Utilisation des répertoires

* Hiérarchiser
* Organiser

### Exemple

Cours :

Info401 :

Projet

Tp

Td

cm

Info402

Info403

Minfo401

## Partitionnement

### Disque

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MBR | Table des partition | Partition 1 | Partition 2 | … | Partition n |

### Une partition

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Bloc de boat | Superblock | Gestion espace libre | i-nodes | Répertoire racine | Répertoires et fichiers |

### Implémentation des fichiers sur le disque

* Contigüe
  + Perte d'espace à la suppression
  + Perte de temps à l'accès
* Aléatoire (solution utilisée)
* Des blocks sont de taille fixes

### Mémorisation des blocs libres

* Tableau de bits
* Liste chainée

Primitives de manipulation de fichiers (unix et C)

int open (const char \* path, int flags, mode\_t m);

Ouvre le fichier de nom path et retourne le descripteur de fichier

Flags peut prendre les valeurs

O\_RDONLY

O\_WRONLY

O\_RDWR

O\_APPREND

O\_CREAT

…

N indique les droits sur le fichier (700, 744, …)

## La fermeture d'un fichier

int close (int d);

Ferme le fichier dont le descripteur est d

## Lecteur d'un fichier

ssize\_t read (int d, void \* buf, ssize\_t n);

Tente de lire n octets sur le fichier décrit par d, place les octets lus dans buf et retourne le nb d'octets effectivement lu

## L'écriture d'un fichier

ssize\_t write (int d, const void \* buf, ssize\_t n);

Tente d'écrire les n premiers octets de la zone référencée par buf sur le fichier décrit par d et retourne le nb d'octets effectivement écrits.

## Exemple

Ecrire un programme qui copie le contenu d'un fichier source dans un fichier destination :

#include <unistd.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdlib.h>

int main (int argc, char\* argv[])

{

    int source, dest;

    char buf[255];

    source = open (argv[1], O\_RDONLY, 0);

    dest = open (argv[2],O\_CREAT | O\_WRONLY, 744);

    int lu;

    while((lu = read(source, buf, 255))==255)

    {

        write(dest, buf, lu);

    }

    write(dest, buf, lu);

    close (source);

    close (dest);

    exit (0);

}

## Se déplacer dans le fichier

off\_t | seek (int d, off\_t t, int whence);

Déplace la "tête de lecture du fichier décrit par d de t octets depuis la position whence :

SEEK\_SET début de fichier

SEEK\_ENO fin de fichier

SEEK\_CVR position courante

Retourne le nb d'octets dans la tête de lecture/écriture s'est déplacé

### Se déplacer dans un fichier

lseek

### Supprimer un fichier

int unlink (const char\* pathname)

Supprime l'entrée fichier pathname du système de fichier

### Obtenir des informations sur un fichier

struct stat{

dev\_t st\_dev; //périphérique

ino\_t st\_ino; //num inode

mode\_t st\_mode; //protection

nlink\_t st\_nlink; //nb de liens matériel

uid\_t st\_vid; //uid du propriétaire

gid\_t st\_gid; //gid du propriétaire

dev\_t st\_rdev; //type de périphérique

off\_t st\_size; //taille en octets

blksize\_t st\_blksize; //taille des blocks E/S

blkcnt\_t st\_blocks; //nb de blocks

time\_t st\_atime; //date dernier Accès

time\_t st\_ntime; //date dernier modification

time\_t st\_ctime; //date dernier changement

}

Int stat(const char\* p, struct stat\* b);

Récupère dans b la structure stat associée au fichier p.

### Exercice

Ecrire un programme sui affiche la taille du fichier passé en paramètre.

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

{

struct stat b;

stat(argv[1],&b);

printf("taille du fichier %s : %d\n", argv(1],b.st\_size);

exit(0);

}

## Manipulation des répertoires

DIR \* opendir (const char \* p); //ouvrir un répertoire

int close dir (DIR\* d); //fermer un répertoire

struct dirent \* readdir(DIR \* d); //lecture d'une entrée d'un répertoire

void rewinddir(DIR \* d); //repositionner la tête de lecture d'un répertoire sur la première entrée

struct dirent{

…

char d\_name[256];

…

};

### Exemple

Ecrire un programme qui affiche les fichiers du répertoire.

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

{

struct dirent \*de;

DIR \* D;

D=opendir("./");

while((de=readdir(D))!=NULL)

{

printf("%s\n",de->d\_name);

}

closedir(D);

exit(0);

}

Ecrire un programme qui supprime tous les fichiers d'un répertoire.

* Utiliser les tests sur fichier

S\_ISREG() //mode\_t

S\_ISDIR() s'utilise sur le mode du fichier (cf fonction et structure stat)

S\_ISLINK()

…

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

{

Struct dirent \* de ;

DIR \* D;

D=opendir("./");

While((de=readdir(D))!=NULL)

{

stat(de->d\_name,&b);

if (S\_ISREG(b.st\_mode))

{

Unlink(de->d\_name);

}

}

closedir(D);

exit(0);

}

### Autres opérations sur répertoire

char \* BITE get cwd(char \* b , size\_t s);

int chdir (const char\* p);

int rmdir (const char\* p);

### Exemple

Afficher tous les fichiers de plus de 50 ko du répertoire père.

Int main(int argc, char\* argv[])

{

Struct dirent \* de;

DIR\* D;

Struct stat b;

Chdir("../");

D=opendir("./");

While((de=readdir(D))!=NULL

{

Stat(de->d\_name,&b);

If((S\_ISREG(b.st\_mode))&&(b.st\_size>51200))

{

Printf("fichier %s taille %d \n", de\_dname, b.st\_size);

}

}

Closedir(D);

Exit(0);

}

# La concurrence

Problème se posant à plusieurs processus accédant à une même ressource partagée.

#### Exemple

Soit une variable partagée A et 2 processus de même code :

X<-Lire(A)

X<-X+1

Ecrire(A,X)

Si A=1

Apres exécution des 2 processus on imagine A=3

P1 P2

X=1 X=1

X=2 X=2

Si on exécute les processus de manière séquentielle ((P1 puis P2) ou (P2 puis P1)), A=3

Si P1 et P2 s'exécutent en même temps, peut être A=2

Le problème vient d'autoriser une lecture d'une variable déjà lue mais pas encore écrite

P1 : x<-lire(A)

P1 : x<-x+1

P2 : x<-lire(A)

P2 : x<-x+1

P1 : écrire(A,x)

P2 : écrire(A,x)

De manière plus générale, dès lors que plusieurs processus lisent et écrient sur une meme variable, ce problème est susceptible de survenir. Conflit lecture/écriture et écriture/écriture

On isole la portion de code qui accède a ces variables partagées. On appelle ces morceaux de code la section critique.

Une exécution qui satisfait le problème de concurrence vérifie 2 propriétés :

* La sûreté :

Un seul processus exécute sa portion de code en section critique à un moment donnée

* La vivacité :

Un processus qui souhaite entrer en section critique finira par y entrer

Solution :

* Désactivation des interruptions (pas envisageable !)
* Variables verrou (marche pas !)
* Alternance stricte (marche, mais pas efficace)

## Alternance de Peterson

(Fonctionne pour 2 processus)

Tant que vrai

Action avant SC

Di<-vrai

Tour<-i

Tant que (D(i+1)/2) ∩ (Tour = i)

Rien

Fin TQ

Action SC

Di<-faux

Action après SC

Fin TQ

Soit 2 processus : P0 et P1 de manière séquentielle puis parallèle pour vérifier les 2 propriétés

P1 puis P0

Tour= 1 0

D0= ~~F~~ ~~Vrai~~ faux

D1= ~~F~~ ~~Vrai~~ faux

P0//P1

Tour = 0 1

D0= vrai P1 passe au SC p0 passe en SC

D1= vrai P0 attend P1 attend

**Fonctionne bien**

* Sémaphores

(Introduction par Dijkastra)

Un sémaphore est un drapeau qui est soir levé soit baissé.

Avant d'entrer en SC, un processus attend que le drapeau soit levé, puis il le baisse (opération P()).

En fin de section critique (SC) il le lève (opération V()).

Les opérations P() et V() sont réalisées de manière atomique (c’est-à-dire la lecture et l'écriture de la variable sémaphore ne peuvent pas être interrompues)

* 2 opérations P() ne peuvent pas être exécutées en même temps.

En informatique un sémaphore est un entier :

P() est une lecture, un test et une décrémentation de l'entier du sémaphore. V() est une incrémentation.

Réécrire le code de départ avec la variable A de manière que dans toutes les exécutions, A=3

S1:

P(S)

X<-Lire (A)

X<-x+1

Ecrire(A,x)

V(S)

Les opérations P(S) et V(S) doivent "encadrer" l'accès aux variables partagées (et donc l'accès à la section critique).

On peut aussi utiliser les sémaphores pour réaliser de la synchronisation entre processus.

Si P1 est constitué d'une action A et P2 d'une action B et qu'on souhaite que a précède B.

On utilise un sémaphore **S** initialisé à 0

P …

A

V(S)

…

## La concurrence conclusion

Se produit lors de l'accès à une ressource partagée par plusieurs processus.

Le morceau de code u cours duquel la ressource est accédée s'appelle la section critique.

2 propriétés sont à garantir pour la bonne marche du système.

Sûreté : au plus un processus exécute son code en section critique à un moment donné.

Vivacité : un processus qui souhaite accéder à sa section critique finira par y accéder.

Diverses solutions :

* Désactivation des interruptions
  + Pas envisageable
* Variables basiques verrou
  + Ne fonctionne pas
* Ordonner les demandes
  + Fastidieux et pas très efficace
* Algorithme de Peterson
  + Pour 2 processus.
* Les sémaphores
  + Un sémaphore est une variable particulière qui est accédée par 2 opérations : P() : lecture, test, décrémentation (de manière atomique)

V() incrémentation

P1 S=1 P2

Avant SC Avant SC

P(S) P(S)

SC SC

V(S) V(S)

Après SC Apres SC

Soit une action A dans un processus P1 et B dans un processus P2. On souhaite que A précède B.

P1 S=0 P2

… …

A P(S)

V(S) B

… …

Soit A et B 2 ressources à partager entre 2 processus.

Soit S1 le sémaphore associé à la ressource A et S2 le sémaphore associé à la ressource B.

Et les codes de P1 et P2 :

P1 P(S1) P2 P(S2)

P(S2) P(S1)

SC SC

V(S1) V(S1)

V(S2) V(S2)

# L'interblocage

L'interblocage survient lorsque deux (ou plus) processus obtiennent "une partie des autorisations" (ie un ensemble de sémaphores) et attendent le reste ! (ex : P1 détient A et attend B, P2 détient B et attend A)

Solution : supprimer l'un des processus.

Comment choisir le processus ?

En fonction :

* Du temps d'exécution d'un processus
* Du nombre d'interblocage dans lequel est impliqué le processus

La méthode est "curative".

Les techniques de prévention des interblocages dans un cadre suffisamment général sont restés au stade d'ébauche. (depuis ~1970)

## Détection d'interblocage

En utilisant le graphe des ressources



P2



A

B



P1



Un cycle indique la présence d'un interblocage.

Si un interblocage se produit, le SE choisit un processus à terminer.

P4

P1



A

B



D

C



P2



P3



# Les tubes

Communication entre 2 processus :

* Arguments passés à la ligne de commande pour exécuter un programme

$ls -al

* Parenté :

Int f=10;

If(fork()==0)

* Fichiers :

Read()/write() par 2 processus différents

* Signaux
* Les tubes

## Création d'un tube

int pipe(int T[2])

Crée un tube :

T[0] permet de lire des données précédemment écrites

T[1] permet d'écrire des données

Père :

Write()

Read()

Fils :

Read()

Write()

Close T[0]

fork()



|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| père  T[1]  T[0] | | | |
|  |  |  |  |

Pipe père vers fils

int main(){

pipe(intT[0]);

}



Close T[1]



|  |  |  |  |
| --- | --- | --- | --- |
| T[1]  T[0] | | | |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| fils  T[1]  T[0] | | | |
|  |  |  |  |



Un tube doit servir à communiquer dans un seul sens :

* Du père vers le fils
* Du fils vers le père

Pour communiquer dans les 2 sens => 2 tubes !

## Exercice mise en application

Ecrire un programme qui envoie 2 entiers à un processus fils. Le processus fils affiche la somme.

int main(int argc, char \* argv[]){

    int A,B;

    int T[2];

    scanf("%d",&A);

    scanf("%d",&B);

    pipe(&T[0]);

    if(fork()==0){

        close(T[1]);

        A=B=0;

        read(T[0],&A,sizeof(int));

        read(T[0],&B,sizeof(int));

        printf("%d\n",A+B);

        close(T[0]);

        exit(0);

    }

    close(T[0]);

    write(T[1],&A,sizeof(int));

    write(T[1],&B,sizeof(int));

    wait(NULL);

    close(T[1]);

    exit(0);

}

## Création de tubes

Int pipe(int t[2])

### Exemple :

Ecrire le code où :

Un processus père envoie un entier ou processus fils et le processus fils répond avec 2 fois la valeur de l'entier:

int main (int argc, char \* argv[]){

    int T1[2];//pour les communications du processus père vers le fils

    int T2[2];//l'inverse

    pipe(&T1[0]);

    pipe(&T2[0]);

    if(fork()==0){

        close(T1[1]);

        close(T2[0]);

        read(T1[0],&buf,sizeof(int));

        buf\*=2;

        write(T2[1],&buf,sizeof(int));

        close(T1[0]);

        close(T2[1]);

        exit(0);

    }

    close(T1[0]);

    close(T2[1]);

    scanf("%d",&buf);

    write(T1[1],&buf,sizeof(int));

    read(T[2],&buf,sizeof(int));

    printf("%d\n",buf);

    close(T1[1]);

    close(T2[0]);

    exit(0);

}

## Utilisation de tube vers des programmes externes

FILE \* popen(const char \* cmd, const char \* type)

Int pclose(FILE \* fp);

Ouvre un tube vers le programme cmd. Si le type vaut :

* "r" la sortie du programme est redirigée vers le descripteur retourné
* "w" l'entrée standard du programme appelé est le descripteur retourné

### Exemple

Un programme qui compte le nombre de 'a' que retourne un "ls"

int main(int argc, char \* argv[]){

    cahr c;

    FILE \* f= popen("ls","r");

    int nb=0;

    while(!feof(f)){

        fread(&c,sizeof(char),1,f);

        if(c=='a'){

            nb++;

        }

    }

    pclose(f);

    exit(0);

}

Fermeture d'un tube ouvert avec popen, on utilise pclose(FILE\*)

Ecrire un programme yes

int main(int argc, char \* argv[]){

    cahr y='y';

    FILE \* f= popen(argv[1],"w");

    int nb=0;

    while(1){

        fprint(f,"%c\n",y);

    }

    exit(0);

}

## Architecture de tuber

Mettre en place n processus (1,n) et n tubes de manière à ce que le processus 1 puisse communiquer avec le processus 2. Le processus 2 avec me processus 3 … et le processus n avec le processus 1



|  |
| --- |
| IN OUT |

|  |
| --- |
| IN OUT |



|  |
| --- |
| IN OUT |



|  |
| --- |
| IN OUT |



|  |
| --- |
| IN OUT |



int main(int argc, char \* argv[]){

    int n=atoi(argv[1]);

    int i;

    int \* T=malloc(2\*n\*sizeof(int));

    for(i=0;i<n;i++){

        pipe(&T[2\*i]);

    }

    for(i=0;i<n;i++){

        if(fork()==0){

            exit(0);

        }

    }

}

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tube 1 | | Tube 2 | | Tube 3 | |  | Tube n-1 | | Tube n | |
|  |  |  |  |  |  |  |  |  |  |  |



n



|  |  |  |
| --- | --- | --- |
| Processus | IN | OUT |
| 1 | T[0] | T[3] |
| 2 | T[2] | T[5] |
| 3 | T[4] | T[7] |
| 4 | T[6] | T[9] |
| … | | |
| I | T[2(i-1)] | T[fi+1)/2n] |
| … | | |
| n-1 |  |  |
| n | T[2n-2] | T[1] |

int main(int argc, char \* argv[]){

    int n=atoi(argv[1]);

    int i;

    int \* T=malloc(2\*n\*sizeof(int));

    for(i=0;i<n;i++){

        pipe(&T[2\*i]);

    }

    for(i=0;i<n;i++){

        if(fork()==0){

            IN=T[2\*(i-1)];

            OUT=T[2\*(i-1)];

            ...

            exit(0);

        }

    }

}

# Les signaux

## Définition

Un signal est une interruption logicielle asynchrone

## Génération d'un signal

* Terlinal CTRL^D
* Exception Matérielle (ex Div pour 0)
* Commande kill
* Logiciels qui envoient des signaux

## Quelques signaux

SIGABRT

SIGCHLD

SIGINT

SIGINT

SIGQUIT

SIGTERM

SIGALRM

SIGCONT

SIGSTOP

SIGKILL

SIGSEGN

SIGUSR1

SIGUSR2

SIGBUS

SIGHUP

SIGPIPE

## Mécanismes

Le traitement du signal s'effectue de la manière suivante

void(\*signal(int signo, void(\*fonc)(int)))(int);

Modifie les traitements du signal signo par la fonction fonc

fonc peut prendre les valeurs :

* SIG\_DFL: traitement par défaut
* SIG\_IGN: ignorer le signal
* Une fonction : c'est le traitement à opérer

void handler(int Sig){

    signa(Sig, handler);// attention, la reception du signal Sig entraine un repositionnement à SIG\_DFL

    printf("Signal recu %d\n",Sig);

    return;

}

int main(int argc, char \* argv[]){

    signal(SIGUSR1,handler);

    while(1){

        pause();

    }

    ecit(0);

}

## Envoyer un signal

* Int kill(pid\_t p,int Sig)

La fonction envoie un signal Sig au processus p.

* Int raise(int sig)

⬄ kill(getpif(),Sig)

Autrement dit, raise envoie un signal Sig au processus appelant !

* Int alarm (int nsec)

Envoie un signal SIGALRM après nsec secondes

* Void pause(void)

Suspend l'exécution du processus jusqu'à la prochaine réception d'un signal

Exemple :

Ecrire un programme qui affiche "Bonjour" les 3 premières fois qu'il reçoit un signal SIGUSR1

Ecrire un programme qui ignore SIGKILL

int cpt=0;

void handler(int Sig){

    if(cpt<2){

        Signal(Sig,handler);

    }

    printf("Bonjour");

    cpt++;

    return;

}

int main (int argc, char\* argv[]){

    Signal(SIGUSR1,handler);

    while(1){

        pause();

    }

    exit(0);

}

int main(int argc, char \* argv[]){

    int pid=atoi(argv[1]);

    for(int i=0;i<5;i++){

        kill(pid, SIGUSR1);

        sleep(2);

    }

    exit(0);

}

Ecrire un programme à l'aide de signaux, un programme qui affiche '\*' à chaque seconde écoulées et '/n' à chaque minute écoulées

int cpt=0;

void handler(int Sig){

    signal(Sig,handler);

    alarm(1);

    cpt++;

    printf("\*");

    if((cpt%60)==0){

        printf("/n");

    }

    flush(stdout);

    return;

}

int main(){

    signal(SIGALARM,handler);

    alarm(1);

    while(1){

        pause();

    }

    exit(0);

}

## Manipulation d'un ensemble de signaux

On dispose d'un ensemble de fonctions qui accèdent à un ensemble de signaux

sigset\_t : type définissant un jeu de signaux

int sigemptyset(sigset\_t\*s): 🡪 vide un jeu de signaux

int sigfillset(sigset\_t\*s): 🡪 ajoute tous les signaux au jeu

int sigaddset(sigset\_t\*s,int sig): 🡪 ajoute le signal sig au jeu s

int sigdelset(sigset\_t\*s,int sig): 🡪 supprime le signal sig du jeu s

int sigismember(sigset\_t\*s,intsig): 🡪 sig est il présent dans le je s ?

## Masquer un jeu de signaux

On utilise la fonction sigprocmask:

Int sigprocmask(int how, const sigset\_t\*set, const sigset\_t\*oldset);

Permet de masquer la reception de certains signaux

How :

* SIG\_BLOCK : ajoute les signaux set à l'ensemble des signaux à bloquer (le set U oldset)
* SIG\_UNBLOCK : retire le jeu de signal set au masque actuel (ie oldset \ set)
* SIG\_SETMASK : remplace le masque actuel par set (ie set)

### Exemple :

Ecrire un code qui désactive les signaux SIGUSR1, SIGINT et SIGQUIT, attend 30 secondes, puis qui les réactives

int main (){

    sigset\_t \*s, \*os;

    sigemptyset(s);

    sigaddset(s,SIGUSR1);

    sigaddset(s,SIGINT);

    sigaddset(s,SIGQUIT);

    sigprocmask(SIG\_SETMASK,s,NULL);

    sleep(30);

    sigemptysets(s);

    sigprocmask(SIG\_SETMASK,s, NULL);

    exit(0);

}

## Recupérer des signaux bloqués

Int sigpending(sigset\*s);

Récupère les signaux bloqués par sigprocmask

## Mise en place d'un gestionnaire

struct sigaction{

    void(\*sa\_handler)();

    sigset\_t sa\_mask;

    int sa\_flags;

}

Cette structure définit :

* Sa\_handler : la fonction gestionnaire (SIG\_DFL, ou SIG\_IGN)
* Sa\_mask : le jeu des signaux à bloquer pendant l'exécution du gestionnaire
* Sa\_flags :
  + SA\_RESTART : si la réception d'un signal interrompt un appel system, celui-ci set repris plutôt qu'être mise en échec
  + SA\_RESET : SIG\_DEL est réinstaller comme nouveau gestionnaire

On "attache" cette structure à un signal avec la fonction sigaction:

Int sigaction(int sig, const struct sigation \* act, const struct sigacyion \* oldact);

Installe le handler act🡪sa\_handler() comme gestionnaire pour le signal sig. L'ancien traitement est sauvegardé dans oldact.

## Mécanisme de communication inter processus

Les différents mécanismes :

* fichiers
* signaux
* tubes
* FiFos
* Mécanisme IPC
  + Sémaphores
  + File de message
  + Mémoire partagée
* Sockets

### Mécanisme "FiFo"

Similaire aux tubes, permettent de faire communiquer 2 processus sans lien de parenté connu.

#### Création :

Int mkfifo(const char \* p, mode\_t n);

Créer une entrée de type fifo dans le système de fichier

S'utilise comme un fichier :

On utilise open(…) read(…) write(…) close(…) et unlink(…)

#### Exemple :

Ecrire deux processus A et B, A qui crée le mécanisme fifo et écrit des entiers saisis par l'utilisateur dedans, et B qui lit dans ce mécanisme et affiche ses entiers.

(?? Systemes d'exploitation, A.tanenboum advanced Programmin in Unix Environnement, R.W.Stevans.)

//A

int main(){

    int t;

    mkfifo("tmp",0777);

    int f=open("tmp",O\_WRLONY,0);

    scanf("%d",&t);

    while(t>0){

        write(f,&t,sizeof(int));

        scanf("%d",&t);

    }

    write(f,&t,sizeof(int));

    close(f);

    exit(0);

}

//B

int main(){

    int s;

    int f= open ("tmp",O\_RDONLY,0);

    read(f,&s,sizeof(int));

    while(s>=0){

        printf("%d\n",s);

        read(f,&s,sizeof(int));

    }

    printf("%d\n",s);

    close(f);

    unlink("tmp");

    exit(0);

}