

Licence 2 Informatique

SYSTÈMES 2

Support de cours/TD intégrés et de TD machine

Alain CROUZIL, Jean-Denis DUROU, Jean-Marc PIERSON

 $\{\verb|crouzil,durou,pierson|| @irit.fr|$

http://www.irit.fr/enseignement

Avant-propos Objectif Ce support est destiné à être utilisé durant les séances de cours/TD intégrés et de TD machine. Il ne constitue pas un support complet dans la mesure où, en séance, vous seront donnés des explications, des exemples, des illustrations et des compléments indispensables. Toutefois, il permet de préparer à l'avance les séances. Utilisation Les six premiers chapitres concernent plus spécifiquement les cours/TD intégrés et sont ponctués d'exercices. Le dernier chapitre contient les sujets des exercices qui doivent être faits dans les séances de TD machine. Par ailleurs, l'index situé à la fin du document constitue un outil indispensable pour accéder rapidement à la description d'une fonction à partir de son nom. Signification des pictogrammes Dans la marge ou près de certains titres, apparaissent les pictogrammes suivants: désigne un sujet d'exercice de cours/TD intégrés; 🗷 est un repère destiné à l'enseignant. Bonne lecture!

Table des matières

1	Intr	oduction	7						
	1.1 Rôle du système d'exploitation								
1.2 Machines virtuelles et bon niveau d'abstraction									
1.3 Outils pour l'utilisation des services du système d'exploitation en langage C									
		1.3.1 Interface de programmation POSIX	8						
		1.3.1.1 Définition	8						
		1.3.1.2 Activation	8						
		1.3.2 Communication avec le système d'exploitation en C ANSI	8						
		1.3.2.1 Paramètres d'un programme	9						
		1.3.2.2 Lancement de commandes du système	9						
		1.3.2.3 Sortie d'un programme	9						
		© Exercice 1	10						
		1.3.2.4 Accès aux variables d'environnement	10						
		Service 2	10						
		1.3.2.5 Gestion des erreurs	10						
2			11 11						
2.1 Définitions									
	2.2	- · · · · · · · · · · · · · · · · · · ·	11						
	2.3	- O	11						
	2.4	8	11						
	2.5	1	12						
	2.6	1 1 1	12						
	2.7		13						
	2.8	T	13						
	2.9		14						
			14						
	2.11		14						
		I	14						
			15						
			16						
		1	16						
			17						
		1 1	17						
			17						
			17						
			18						
		© Exercice 6	18						

- 3/60 -

AC-v1.4

3	Ges		21 21							
3.1 Introduction										
		*****	21							
		3.1.2 Mémoire virtuelle	21							
		3.1.3 Objectifs	21							
		3.1.4 Réalisation de la mémoire virtuelle	22							
	3.2 Pagination									
		3.2.1 Transformation des adresses	22							
		3.2.2 Table des pages virtuelles	22							
		3.2.3 Défaut de page	23							
		© Exercice 8	23							
			23							
		3.2.4 Algorithmes de remplacement	23							
© Exercice 10										
Exercice 13										
		Service 14	24							
4	Str	ucture interne du système de fichiers d'Unix	27							
	4.1		27							
	4.2		27							
	4.3	v	28							
		Exercice 15	28							
	4.4		29							
5	Pri	mitives Unix (POSIX.1) de manipulation des fichiers	31							
•	5.1	•	31							
	5.2 Représentation d'un ensemble par un entier									
	5.3	•	31 32							
	0.0		32							
			33							
			33							
			33							

 Sexercice 16
 33

 5.3.5 Contrôle de la position courante
 33

 Sexercice 17
 33

 Exercice 18
 34

 Exercice 19
 34

 5.4 Duplication de descripteur
 34

 Exercice 20
 34

 5.5 Suppression d'un fichier (lien physique)
 35

 5.6 Accès aux caractéristiques d'un fichier
 35

 5.7 Parcours d'un répertoire
 36

 Exercice 21
 36

 5.8 Parcours d'une sous-arborescence
 37

 Exercice 22
 37

Table des matières

Systèmes 2 Systèmes 2 AC-v1.4

4/60

6	Fon	ctions de la bibliothèque standard pour la manipulation des fichiers 3
	6.1	Introduction
	6.2	Ouverture
	6.3	Fermeture
	6.4	Lecture
		6.4.1 Lecture dans un fichier de texte
		6.4.1.1 Lecture d'un caractère
		6.4.1.2 Lecture d'une chaîne de caractères
		6.4.1.3 Lecture formatée
		6.4.2 Lecture dans un fichier binaire
	6.5	Écriture
		6.5.1 Écriture dans un fichier de texte
		6.5.1.1 Écriture d'un caractère
		6.5.1.2 Écriture d'une chaîne de caractères
		6.5.1.3 Écriture formatée
		6.5.2 Écriture dans un fichier binaire
	6.6	Gestion de la position courante
	0.0	6.6.1 Détection de fin de fichier
		6.6.2 Positionnement dans un fichier
		© Exercice 24
		© Exercice 25
		© Exercice 26
	6.7	Réouverture d'un identificateur de fichier
	6.8	Suppression d'un fichier
	6.9	Conseils
	6.10	Remarques sur les fichiers binaires
		© Exercice 27
		© Exercice 28
		© Exercice 29
_	G	And I and American Materials and American Americ
7	Suje	ets des travaux dirigés sur machine 4 Séance 1
		Exercise 2
		Exercice 3
		Séance 2
		Exercice 4
		Exercice 5
		Séance 3
		Exercice 6
		Exercice 7
		Exercice 8
		Séance 4
		Exercice 9
		Exercice 10
		Exercice 11
		Séance 5

Exercice 12	 	 	 	 	 	. 56
Erronoico 12						5.6

Table des matières

6/60

ndex																									59
		Exercice	16	٠		٠	٠	٠		٠	٠		٠	٠	٠		٠	٠		٠	٠		٠	•	57
		Exercice	15																						57
	Séance	6			 																				57
		Exercice	14						 																56
		Exercice	13																						56

Chapitre 1 Introduction

1.1 Rôle du système d'exploitation

Le système d'exploitation fournit à chaque utilisateur un environnement de travail adapté à ce qu'il veut faire, c'est-à-dire une machine virtuelle qui est une version simplifiée (abstraction) de l'architecture matérielle.

Les fonctions de l'environnement d'exécution de la machine virtuelle sont :

- la gestion de l'information :
 - structuration des programmes, des données manipulées par ces programmes, ainsi que des objets manipulés par l'utilisateur;
 - o conservation des fichiers et leur catalogue;
 - \circ désignation des objets (comment on identifie un fichier, comment on le distingue d'un autre) :
 - o gestion de l'espace d'adressage (permet de stocker, dans la mémoire principale, l'information).
- la gestion de la communication pour permettre l'accès:
 - à des machines virtuelles éloignées ou non, à des machines réseaux, c'est-à-dire permettre l'accès à d'autres machines virtuelles, qu'elles soient locales ou distantes;
 - o à des supports externes (disques durs...), via des entrées-sorties ;
 - o à des objets, des composants partageables;
 - $\circ\,$ à des services ;
 - o à des fichiers, qu'ils soient distants ou non.
- $\bullet \ la \ gestion \ de \ l'exécution$ pour permettre :
 - de lancer des exécutions mais aussi de suivre ces exécutions de programmes, qu'elles soient faites en séquence, en parallèle, en concurrence sur l'accès à une ressource;
 - o de composer et d'enchaîner les programmes ;
 - o de synchroniser des exécutions (quand on fait de la programmation parallèle).

1.2 Machines virtuelles et bon niveau d'abstraction

- Niveau de l'application particulière :
 - $\circ\,$ interaction Homme-machine (IHM), logiciels spécifiques
- Niveau fichiers:
 - o langage de commande (ex: shell de Bourne);
 - o interpréteur du langage de commande (ex:/bin/sh).

AC - v1.4 - 7/60 - Sustèmes 2

8/60 1 – Introduction

• Niveau langage machine:

- o création de l'exécutable et de son image en mémoire (sa représentation dans la mémoire):
- o processus: création, états, fin;
- entrées-sorties, interruptions.
- Niveau langage évolué : C, C++, Java...

1.3 Outils pour l'utilisation des services du système d'exploitation en langage C

Les services du système d'exploitation peuvent être invoqués:

- par l'intermédiaire d'une interface graphique;
- grâce à un interpréteur de commande;
- depuis un programme écrit dans un langage évolué.

Dans la suite de ce cours, les services du système Unix seront invoqués depuis des programmes écrits en langage C:

- de manière directe, en utilisant la bibliothèque Unix (l'interface de programmation POSIX) qui donne accès aux « primitives Unix »;
- de manière indirecte, en utilisant la bibliothèque standard qui offre des outils portables sur tous les systèmes d'exploitation, à la seule condition de disposer d'un compilateur C ANSI.

1.3.1 Interface de programmation POSIX

1.3.1.1 Définition

POSIX (Portable Operating System Interface for unix) est une norme internationale d'« interfaçage » des programmes avec les systèmes d'exploitation. Elle est basée sur le système Unix. La première partie de cette norme, « System Application Program Interface (API) [C Language] », POSIX.1, décrit les fonctions utilisables dans un programme pour interagir avec le système d'exploitation, en utilisant la syntaxe du langage C. Ces fonctions constituent un sous-ensemble portable (95 fonctions) des fonctions présentes dans les bibliothèques « système » des différentes variantes d'Unix. La norme a repris et étendu les fonctionnalités de la bibliothèque standard (stdio.h, time.h, ...) et a apporté des fonctionnalités nouvelles (gestion de l'arborescence des fichiers, des processus, ...). Toutes ces fonctionnalités peuvent se regrouper en huit parties: paramètres du système (configuration, base de noms, ...), environnement d'un processus (état, temps d'exécution, ...), lancement et terminaison des processus, interaction avec la hiérarchie des fichiers, entrées-sorties de bas niveau (« primitives » d'entrées-sorties), signaux, tubes anonymes, interaction avec le terminal. Viennent se rajouter six parties optionnelles: entrées-sorties asynchrones, gestion des files de messages, processus légers (threads), ordonnancement des processus, sémaphores, gestion avancée de la mémoire.

1.3.1.2 Activation

Pour garantir que le compilateur C vérifie qu'un programme respecte la norme POSIX.1, il faut placer, *avant* les inclusions de fichiers d'en-tête (les fichiers d'extension .h), la directive : #define _POSIX_C_SOURCE 1

1.3.2 Communication avec le système d'exploitation en C ANSI

Nous verrons plus loin comment faire appel indirectement au système de gestion des fichiers avec la bibliothèque standard du langage C. Mais il est aussi possible d'effectuer des interactions simples

Systèmes 2

en utilisant cette bibliothèque.

1.3.2.1 Paramètres d'un programme

Il est possible de récupérer les paramètres (arguments) qui sont passés au programme au moment du lancement de son exécution. Pour cela, il faut modifier l'en-tête de la fonction principale et utiliser: int main(int argc, char *argv[])

- où <u>argc</u> contient le nombre de paramètres effectivement passés au programme augmenté de 1 pour le nom du programme exécutable :
 - <u>argv</u> est un tableau de <u>argc</u> pointeurs vers les chaînes de caractères qui constituent les paramètres effectifs du programme.

Exemple: si prog est le nom du programme exécutable correspondant au fichier source prog.c et si on lance son exécution en tapant la ligne de commande: prog 3794 3.14 chaîne alors on a:

Ces paramètres peuvent être utilisés dans le programme, sachant qu'il peut être nécessaire d'effectuer des conversions de chaînes de caractères en nombres grâce par exemple aux fonctions atoi ou atof (déclarées dans stdlib.h) ou bien à la fonction sscanf.

Exemple:

```
int n;
sscanf(argv[1],"%d",&n);
```

Sous Unix, l'interface POSIX permet de rajouter un paramètre à l'en-tête de la fonction main:

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

Le paramètre enve est un tableau de chaînes de caractères contenant les variables d'environnement au moment de l'exécution, sous la forme variable=valeur.

Toutefois, l'accès aux valeurs des variables d'environnement est plus simple si l'on utilise la fonction getenv qui est décrite à la page 10 et qui fait partie de la bibliothèque standard.

1.3.2.2 Lancement de commandes du système

La fonction

AC-v1.4

#include <stdlib.h>

int system(const char *commande)

permet de lancer l'exécution de la commande <u>commande</u>. Si le paramètre <u>commande</u> est égal à NULL, la fonction **system** retourne une valeur non nulle s'il existe au moins une commande du système qui lui est accessible. Si le paramètre <u>commande</u> est différent de NULL, la valeur de retour dépend de l'implémentation.

1.3.2.3 Sortie d'un programme

Il y a deux manières de terminer l'exécution d'un programme et de retourner au système un entier appelé $code\ de\ retour$:

 en utilisant, dans la fonction main, l'instruction return <u>code_retour</u>;

 en faisant appel, depuis n'importe où dans le programme, à la fonction #include <stdlib.h> void exit(int code retour)

Remarques:

- L'exécution de l'opérateur return dans une fonction autre que main permet d'interrompre l'exécution de la fonction en question et de rendre la main à la fonction appelante.
- Les conventions des commandes Unix préconisent qu'un programme retourne 0 quand tout s'est bien passé ou un entier positif permettant de décrire une situation anormale.
- Sous Unix, la fonction exit fait appel à la primitive:

```
#include <unistd.h>
_exit(int code_retour)
```

♠ Exercice 1

Écrire un programme C ANSI qui affiche à l'écran ses arguments (un argument par ligne).

1.3.2.4 Accès aux variables d'environnement

```
La fonction
#include <stdlib.h>
char *getenv(const char *nom_var)
```

retourne l'adresse du début de la chaîne de caractères correspondant à la valeur de la variable d'environnement <u>nom.var</u>. Si la variable n'existe pas, cette fonction retourne NULL.

```
Exemple : printf("LOGNAME=%s\n",getenv("LOGNAME"));
affiche le nom d'utilisateur (par exemple 12inf200).
```

♠ Exercice 2

Écrire un programme C ANSI qui prend en paramètre le nom d'une variable d'environnement et qui affiche à l'écran sa valeur.

Remarque: sous l'interpréteur, on peut obtenir l'affichage des variables d'environnement et de leurs valeurs en lançant la commande env.

1.3.2.5 Gestion des erreurs

En cas d'erreur, les fonctions de la bibliothèque standard qui font appel indirectement au système (comme les fonctions de gestion des entrées-sorties sur les fichiers) ainsi que la plupart des fonctions de l'API POSIX positionnent la variable globale errno. Le fichier errno. h contient des messages d'erreur associés aux valeurs de la variable errno. La fonction de la bibliothèque standard:

```
#include <stdio.h>
void perror(const char *message)
```

permet d'afficher sur stderr la chaîne passée en paramètre suivie du caractère :, suivi éventuellement du message associé à l'erreur.

```
Exemple :
int fd;
fd=open("fich",O_RDONLY);
if (fd==-1) perror("fich");
Si le fichier fich n'existe pas, perror va afficher un message ressemblant à:
fich : No such file or directory
Remarque : la fonction open est décrite de manière détaillée à la page 32.
```

Chapitre 2 Processus

2.1 Définitions

Un processus est l'instance d'un programme mis sous forme exécutable en mémoire. C'est aussi le déroulement de l'exécution de cette instance par la machine virtuelle :

- niveau utilisateur;
- niveau système.

Mais encore, c'est aussi un objet actif qui modifie son état et ses données en utilisant la machine virtuelle: un processus peut être représenté par son état, son contexte et son espace d'adressage.

2.2 Création d'un processus et exécution

≠ 1

2.3 Image d'un processus sur disque

 \not 2

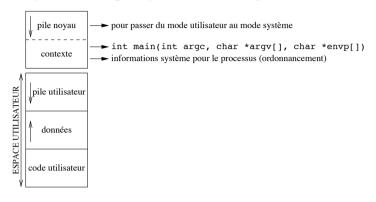
2.4 Image d'un processus en mémoire virtuelle

₡∋ 3

AC - v1.4 - 11/60 - Systèmes 2

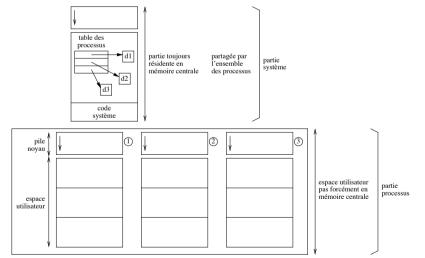
12/60 2 - Processus

2.5 Espace d'adressage du processus vu d'un système



2.6 Description du processus (process status)

- identificateur du processus (pid);
- identificateur du créateur du processus (ppid);
- zone mémoire allouée + pointeur de pile d'exécution
- informations d'ordonnancement (priorités);
- valeur du compte temps.



2.7 - Ordonnancement 13/60

- Les processus sont en concurrence.
- À un utilisateur sont associés plusieurs processus.
- Tout processus peut créer un processus sarborescence de processus: tout le système repose sur ce concept.

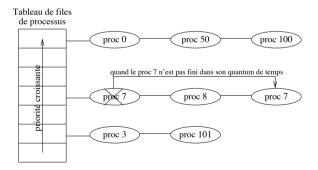
4

2.7 Ordonnancement

- Choix du processus en exécution (attribution du processeur);
- Pour combien de temps : notion de quantum de temps pendant lequel un processus peut s'exécuter.

≠ 5

2.8 Files des processus prêts

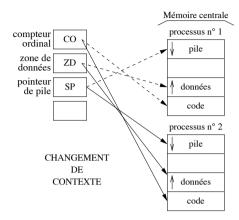


Remarque : la priorité est fonction de l'utilisateur, du fait qu'il s'agit d'un processus système ou pas.

AC-v1.4 Systèmes 2

14/60 2 - Processus

2.9 Registres de l'unité centrale



Lorsqu'un processus est élu, les registres de l'unité centrale reçoivent les valeurs du nouveau processus. Quand un processus est enlevé de l'unité centrale, il passe de l'état élu (actif) à un autre état (zombie, bloqué...) et son état est sauvegardé dans un PCB (process control block). Le PCB contient ce qu'il faut pour relancer un processus (quand il repasse dans l'état élu):

- état du processus :
- valeurs des registres de l'unité centrale (UC), dont le CO (adresse de la prochaine instruction à exécuter);
- $\bullet\,$ informations sur l'ordonnancement du processus (priorité) ;
- informations sur la mémoire utilisée (notamment les pages utilisées);
- informations sur les entrées-sorties (E/S), comme les fichiers ouverts.

2.10 Activité de l'unité centrale

€ 6

Remarque: les appels systèmes peuvent concerner:

- la gestion des processus: fork, exec, malloc...
- la gestion des fichiers (E/S): read, open, stat, fcntl...
- la gestion du système de fichiers: mkdir, chdir, mount...
- la gestion des protections: chmod, chown...
- la gestion du temps: times...
- \bullet la gestion des signaux : kill, sigaction...

2.11 Gestion des processus avec les primitives de la bibliothèque Unix

2.11.1 Identification du processus courant

Un processus qui crée un autre processus est appelé le processus père et le processus créé est le processus fils. Un processus appartient à un groupe de processus. Un processus a un propriétaire qui

appartient à un groupe d'utilisateurs.

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

- pid_t getpid(void) retourne l'identificateur du processus courant (pid).
- pid_t getppid(void) retourne l'identificateur du processus père du processus courant (ppid).
- pid_t getpgrp(void) retourne l'identificateur du groupe (pgrp) de processus auquel appartient le processus courant.
- char *getlogin(void) retourne l'adresse d'une chaîne de caractères contenant le nom de l'utilisateur (login) connecté sur le terminal contrôlant le processus courant ou NULL si cette information n'est pas disponible.
- uid_t getuid(void) retourne l'identificateur de l'utilisateur (uid) propriétaire du processus courant.
- gid_t getgid(void) retourne l'identificateur du groupe d'utilisateurs (gid) du propriétaire du processus courant.

Remarques.

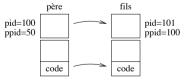
pid_t, uid_t et gid_t sont des synonymes de types entiers définis dans sys/types.h. Vous pouvez obtenir votre identificateur et celui de votre groupe avec la commande shell id.

2.11.2 Création

#include <sys/types.h>
#include <unistd.h>
pid_t fork(void)

La fonction fork permet de créer un processus fils. Le processus créé (fils) est une copie quasi-conforme (un clone) du processus appelant (père). En effet, le fils hérite, par copie (duplication), de tout le contexte du père, c'est-à-dire de la plupart de ses attributs: environnement (PATH, SHELL...), priorité, numéro de groupe, segment de pile, segment de données, fichiers ouverts... Il partage le code du père. Le processus fils débute son existence dans la fonction fork. À partir de là, les deux processus s'exécutent « en parallèle ». Les principales différences entre les deux processus sont:

- leurs identificateurs,
- la valeur retournée par fork:
 - o dans le contexte du père : la valeur -1 en cas d'échec, l'identificateur du fils sinon,
 - o dans le contexte du fils : la valeur 0.
- la localisation des segments de pile et de données: ces segments sont privés; à la création du processus fils, ils contiennent les mêmes informations; par la suite, ils évoluent indépendamment.
- les descripteurs de fichiers (chaque processus possède ses propres descripteurs qui pourraient partager des ressources).

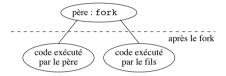


Après la création les deux processus sont indépendants

AC-v1.4 Systèmes 2

Schémas classiques de création d'un processus fils

```
if ((pid=fork())==-1)
                                          switch (pid=fork())
{ /* Code exécuté par le père
     en cas de problème
                                          case -1: /* Code exécuté par le père
                                                      en cas de problème
                                                    . . .
else if (pid)
                                                    hreak.
    { /* Code exécuté par le père */
                                          case 0 : /* Code exécuté par le fils */
    }
                                                    break:
     وي [ و
                                          default: /* Code exécuté par le père */
    { /* Code exécuté par le fils */
```



Remarque : la création du processus fils peut échouer par exemple si la table des processus est pleine (plus le droit de créer des processus) ou si la mémoire virtuelle est pleine.

♠ Exercice 3

Écrire un programme qui crée un processus fils. Après la création, le processus père doit afficher son propre identificateur. Le processus fils doit afficher l'identificateur de son père ainsi que son identificateur.

Ø 7

2.11.3 Terminaison d'un processus

```
#include <stdlib.h>
void exit(int compte_rendu)
```

La fonction exit termine l'exécution du processus courant et transmet à son père un compte rendu dans l'octet de poids faible de compte_rendu: 0 (ou EXIT_SUCCESS) pour une terminaison normale, ou un entier dans l'intervalle [1,255] (par exemple EXIT_FAILURE) en cas de problème.

Un processus qui a terminé son traitement s'autodétruit. Ceci conduit :

- à fermer tous les fichiers ouverts du processus
- à réveiller son père (si le père est en attente de la terminaison d'un fils)
- à notifier au père sa terminaison par la transmission d'un compte rendu.

Un processus terminé se retrouve dans l'état de zombie tant que son père n'est pas terminé et qu'il n'a pas pris connaissance de la terminaison de son fils, c'est-à-dire tant qu'il n'a pas attendu la terminaison de son fils (voir la fonction wait ci-dessous). Quand un processus crée des processus fils, il est important qu'il attente par la suite leurs terminaisons. Dans le cas contraire, les fils restent sous la forme de zombies durant toute la durée de l'exécution de leur père.

2.11.4 Attente de la terminaison d'un fils

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
pid_t wait(int *pcirconstance)
```

La fonction wait endort le processus appelant jusqu'à la terminaison de l'un de ses fils (ou l'arrivée d'un signal non ignoré). Elle retourne :

- le pid du fils qui s'est terminé et, si elle est différente de NULL, place à l'adresse <u>pcirconstance</u> un entier qui décrit la circonstance de cette terminaison. Si cet entier s'appelle <u>circonstance</u>, la macro WEXITSTATUS(<u>circonstance</u>) permet alors de récupérer le code de retour transmis par le fils (grâce à la fonction exit ou à l'instruction return dans la fonction main).
- la valeur -1 si les terminaisons de tous les fils ont déjà été prises en compte ou s'il n'y a pas de fils.

2.11.5 Endormissement temporaire d'un processus

```
#include <unistd.h>
unsigned int sleep(unsigned int sec)
```

La fonction sleep endort le processus appelant durant \underline{sec} secondes. Si le processus est réveillé par un signal avant l'écoulement des \underline{sec} secondes, sleep retourne le nombre de secondes qu'il restait à attendre. Sinon, elle retourne 0.

Exercice 4

Écrire un programme qui permet à quatre processus frères, numérotés 1, 2, 3 et 4, de constituer avec le père, numéroté 0, un anneau unidirectionnel. Entre ces processus, on définit la relation de successeur de la manière suivante :

- successeur(i)=i+1 pour i=0,1,2,3.
- successeur(4)=0

Chaque processus fils doit afficher \mathbb{N} fois (\mathbb{N} étant une constante symbolique définie dans le programme) un message accompagné de son identificateur de processus et il retourne un compte rendu égal à son numéro (pas son identificateur). Le père attend, avant de se terminer, la fin de tous ses fils et, à chaque terminaison d'un fils, il affiche l'identificateur du fils terminé ainsi que le code de retour de ce fils.

2.11.6 Primitives de recouvrement

La famille de primitives exec permet à un processus de charger en mémoire un nouveau code exécutable. Après un exec, le contrôle de l'exécution est donné au programme lancé. Le code et les données étant écrasées, il est impossible de revenir au programme dans lequel se trouve l'appel à la fonction de la famille de primitives exec. Le processus garde son identificateur et donc la plupart de ses caractéristiques : il n'y a pas de création de processus.

Les fonctions de la famille diffèrent principalement par la manière dont on passe en paramètres les informations concernant le nouveau code qui va être exécuté. Par exemple, la fonction:

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

```
int execl(const char *désignation_fichier, const char *\underline{\operatorname{arg}}, \underline{\dots}, NULL) permet de charger en mémoire le programme défini par:
```

- désignation_fichier qui est la désignation du fichier qui contient le code à exécuter;
- les paramètres suivants qui sont des chaînes de caractères correspondant aux paramètres de la fonction main, c'est-à-dire argv[0], argv[0]...
- le dernier paramètre qui doit être NULL (ou (char *)0).

 $\label{lem:example:pour remplacer} \textit{Exemple:} \text{ pour remplacer le code du processus courant par celui de la commande } 1s -1, il suffit d'utiliser:}$

```
execl("/bin/ls","ls","-l",NULL);
   Une variante intéressante est la fonction:
#include <unistd.h>
int execlp(const char *nom_fichier, const char *arg, ..., NULL)
```

qui se comporte comme la fonction execl à l'exception du premier paramètre <u>nom_fichier</u> qui est le nom du fichier exécutable que l'on veut lancer et qui sera recherché dans les répertoires contenus dans la variable d'environnement PATH.

```
Exemple : l'appel précédent peut donc être remplacé par :
execlp("ls", "ls", "-l", NULL);
```

Si tout se passe bien, les fonctions de la famille **exec** ne retournent jamais puisque c'est le nouveau code qui remplace l'ancien. Si une erreur se produit (exécutable inexistant par exemple), alors elles retournent -1.

Attention: la bibliothèque standard propose la fonction system qui a été décrite au paragraphe 1.3.2.2, page 9, et qui permet de lancer l'exécution de la commande. Mais la fonction system crée un processus fils car elle lance un nouveau shell pour interpréter la commande.

♠ Exercice 5

```
Que produit l'exécution du fragment de programme suivant:
printf("Maintenant :\n");
execlp("date","date",NULL);
```

♠ Exercice 6

```
Soit le programme suivant:
#include <stdio.h>
#include <unistd.h>
int main(void)
{
   printf("%d\n",(int)getpid());
   return 0;
}
```

printf("Terminé\n");

Si pid est le nom du fichier exécutable produit par la compilation de ce programme, que produit l'exécution du programme suivant :

AC-v1.4

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
   printf("%d\n",(int)getpid());
   printf("%d\n",execlp("pid","pid",NULL));
   return 0;
}
```

AC-v1.4 Sustèmes 2 Sustèmes 2

Voici un exemple d'utilisation des primitives fork, wait et exec dans lequel les contrôles d'erreur ainsi que les inclusions des fichiers d'en-tête ont été omis :

```
int main(void)
{
  int pid=fork();
  if (pid==0)/* on est dans le fils */
    execl("/bin/ls","ls","-1",NULL);
  else
    if (pid>0)/* on est dans le père */
        wait(NULL); /* le père attend la fin du fils */
    exit(0); /* terminaison du père (le fils n'exécute pas cette instruction) */
}
```

Exercice 7

Écrire un programme C qui réalise un travail similaire à celui de l'interpréteur de commandes lorsqu'on lui demande d'exécuter la séquence de commandes suivante :

1s -1

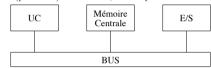
pwd

L'interpréteur doit créer un processus fils, lui faire exécuter la commande ls -1 et attendre la fin de ce processus fils. Puis, il crée un nouveau processus fils, lui fait exécuter la commande pwd et attend sa fin.

Chapitre 3 Gestion de la mémoire : mémoire virtuelle et allocation non contiguë

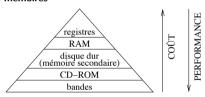
3.1 Introduction

Pour qu'un programme (processus) soit exécuté, il faut qu'il soit en mémoire centrale.



Tous les processus ne rentrent pas en mémoire centrale car elle a une taille limitée.

3.1.1 Hiérarchie des mémoires



3.1.2 Mémoire virtuelle

C'est le support de l'ensemble des informations potentiellement accessibles. La mémoire virtuelle est utilisée pour les raisons suivantes :

- $\bullet \ \ rapport \ co\^ut/performance \, ;$
- \bullet les processus trop gros ne peuvent par rentrer en mémoire centrale \Longrightarrow chargement partiel + chargement à la demande.

La mémoire virtuelle permet d'utiliser de la mémoire secondaire comme si c'était de la RAM (mémoire centrale).

3.1.3 Objectifs

- fournir un espace d'adressage indépendant de la mémoire physique;
- pouvoir exécuter des programmes dont la taille excède la taille de la mémoire physique.

AC - v1.4 - 21/60 - Systèmes 2

22/60 3 – Gestion de la mémoire : mémoire virtuelle et allocation non contiguë

3.1.4 Réalisation de la mémoire virtuelle

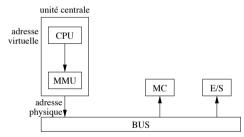
- représentation physique : mémoire centrale + mémoire secondaire ;
- gestion basée sur la pagination, l'allocation par blocs, la segmentation.

3.2 Pagination

Le mécanisme de pagination permet d'allouer aux processus un espace de mémoire physique (la mémoire centrale ou RAM) constitué de parties non contiguës. La mémoire (l'espace d'adressage) physique est découpée en blocs de taille fixe appelés pages physiques, cadres, cases ou frames. La mémoire qui est directement adressée par les processus, appelée mémoire virtuelle, est elle-même découpée en blocs de taille fixe appelés pages virtuelles ou, plus simplement, pages. Pour simplifier, la taille d'une page virtuelle est la même que celle d'une page physique.

3.2.1 Transformation des adresses

Une page virtuelle peut correspondre à une page physique quelconque. Le mécanisme de pagination assure à travers le matériel (MMU: memory management unit) la correspondance entre une adresse de la mémoire virtuelle et une adresse de la mémoire physique. Ce mécanisme utilise la table des pages qui est associée à chaque processus et qui est, elle-même, stockée en mémoire physique.



 ${\it adresse \ virtuelle} = \underbrace{{\it adresse \ de \ page}}_{} + \underbrace{{\it d\'eplacement \ dans \ la \ page}}_{}$

adresse physique = adresse de cadre + déplacement dans le cadre

3.2.2 Table des pages virtuelles

	Présente	Modifiée	Protection	n° page physique
0	oui			0
1	oui			3
2	non			
3	oui			2
4	non			
5	non			
6	oui			1

Systèmes 2

3.2.3 Défaut de page

La mémoire virtuelle peut être de taille très importante et beaucoup plus grande que la taille de la mémoire physique. À un instant donné de l'exécution d'un processus, toutes ses pages ne sont pas obligatoirement en mémoire physique; certaines peuvent être conservées sur la mémoire secondaire (le disque). Quand un processus accède à une page qui n'est pas en mémoire physique, on dit qu'il y a un défaut de page. L'instruction est interrompue et le système doit alors allouer un cadre à cette page. Si aucun cadre n'est libre, il réquisitionne un cadre occupé par une autre page : c'est le remplacement de page. Différents algorithmes peuvent être utilisés pour déterminer quelle page va être remplacée. Si la page remplacée a été modifiée par rapport à sa version sur la mémoire secondaire, elle doit être réécrite en mémoire secondaire. Enfin, l'instruction qui a provoqué le défaut de page peut être exécutée. Ce mécanisme s'appelle la pagination à la demande.

Exercice 8

Une mémoire virtuelle a une taille de page de 1024 mots, huit pages virtuelles et quatre pages physiques. Sa table des pages est la suivante:

Page virtuelle	Page physique
0	3
1	1
2	pas en mémoire physique
3	pas en mémoire physique
4	2
5	pas en mémoire physique
6	0
7	pas en mémoire physique

- 1. Donnez la liste des adresses qui provoqueront un défaut de page.
- Quelles sont les adresses physiques correspondant aux adresses virtuelles 0, 3728, 1023, 1024, 1025, 7800, 4096?

Exercice 9

AC-v1.4

Dans un système paginé, le surcoût dû à un défaut de page est de 20 microsecondes et le temps d'accès à un mot de la mémoire centrale de 500 nanosecondes. Calculer le temps d'accès moyen d'un tel système, en fonction du taux de défaut de page, sachant que la table des pages est entièrement en mémoire centrale. Quel est le taux maximum de défaut de page qui maintient le temps d'accès moyen en dessous de 1,2 microsecondes?

3.2.4 Algorithmes de remplacement

Quand une page n'est pas en mémoire centrale, le système doit :

- la chercher en mémoire secondaire et la charger en mémoire centrale;
- s'il n'y a plus de place en mémoire centrale:
 - o choisir un cadre en mémoire centrale;
 - o éventuellement, le sauvegarder en mémoire secondaire;
 - o le remplacer par la page recherchée.

Il existe plusieurs stratégies pour déterminer quelle est le cadre qui va être remplacé. Par exemple:

OPT – L'algorithme de remplacement OPT, ou algorithme optimal, consiste à remplacer la page à laquelle on accédera le plus tard (dont la prochaine référence est la plus tardive). Cet algorithme n'est pas applicable en pratique car on ne peut pas prévoir les accès futurs, mais il permet de comparer les performances des autres algorithmes à ses performances qui sont les meilleures.

RANDOM – L'agorithme de remplacement RANDOM consiste à choisir au hasard la page qui va être remplacée.

- FIFO L'algorithme de remplacement FIFO (first in first out), ou algorithme chronologique de chargement, consiste à remplacer la page qui est en mémoire depuis le plus longtemps. Le matériel doit donc maintenir une file des numéros de pages dans l'ordre où elles ont été chargées en mémoire
- LRU L'algorithme de remplacement LRU (least recently used), ou algorithme chronologique d'utilisation, consiste à remplacer la page qui n'a pas été utilisée depuis le plus longtemps (dont la dernière référence est la plus ancienne). Le matériel doit donc maintenir la liste des pages ordonnées par leur dernier moment d'utilisation.

Évidemment, les remplacements entraînent un surcoût.

♠ Exercice 10

24/60

Une mémoire centrale comprend 25 cadres. Cinq processus s'exécutent simultanément. Chaque processus dispose de 5 cadres et produit la liste de références (en numéros de pages):

processus
$$i + 1 : 0 + 5i, 1 + 5i, 2 + 5i, 3 + 5i, 4 + 5i, 0 + 5i, 1 + 5i, \dots$$
 avec $0 \le i \le 4$.

L'algorithme de remplacement utilisé est LRU et la mémoire est intialement vide.

- 1. Quel est le taux de défaut de page provoqué par cette exécution?
- 2. On introduit un 6ème processus, analogue aux 5 autres quant à la suite de références produites (pour i = 5). On attribue maintenant à chaque processus 4 cadres. Calculer le nouveau taux de défaut de page.
- 3. Décrire sur cet exemple le phénomène de l'écroulement.

© Exercice 11

Un programme a un espace de mémoire virtuelle de 512 mots. On considère la suite d'adresses virtuelles: 34, 123, 145, 510, 456, 345, 412, 10, 14, 12, 234, 236, 412.

- Donner la suite des numéros de pages virtuelles référencées, sachant qu'elles comportent 100 mots.
- Le programme dispose de 300 mots en mémoire centrale. Calculer le taux de défaut de page pour les algorithmes FIFO. LRU et OPT.

♠ Exercice 12

On a constaté que le nombre d'instructions exécutées entre deux défauts de page est directement proportionnel au nombre de cadres alloués à un programme. Si on double la mémoire disponible, on double le temps entre deux défauts de page consécutifs. Supposez qu'une instruction normale requiert 1 microseconde et qu'un défaut de page en requiert 2001. Si le programme s'exécute pendant 60 secondes et s'il fait 15000 défauts de page, quelle serait sa durée d'exécution si on avait deux fois plus de mémoire?

Exercice 13

S'il faut une microseconde pour exécuter une instruction et si le traitement d'un défaut de page entraı̂ne un surcoût de n microsecondes, donner la formule du temps d'exécution moyen d'une instruction s'il se produit un défaut de page toutes les k instructions.

♠ Exercice 14

Le taux d'activité moyen du processeur est donné par :

$$taux = t/(t + pT) = 1/(1 + pS)$$

3.2 – Pagination 25/60

où t est le temps moyen d'exécution d'une instruction, T le temps moyen de traitement d'un défaut de page, p la probabilité globale (sur l'ensemble des processus) d'un défaut de page. Le rapport S=T/t est du même ordre de grandeur que le rapport du temps de transfert d'une page depuis le disque sur le temps d'accès en mémoire principale ; sa valeur est comprise entre 1000 et 10000.

- 1. Que peut-on dire de p pour que le taux reste voisin de 1?
- 2. Comment évolue le taux en fonction de p?

AC – v1.4 Systèmes 2

Chapitre 4 Structure interne du système de fichiers d'Unix

4.1 Introduction

Un fichier Unix est une suite finie d'octets. Il est matérialisé par des blocs du disque. À chaque fichier est associée une petite table appelée *inode* (ou nœud d'index).

Un inode contient des informations concernant le fichier ainsi que 10 adresses (numéros) de blocs de données et 3 adresses de blocs d'adresses (pointeurs à simple, double et triple indirection).

4.2 Structure de base du système de fichiers

n° de bloc	Contenu
0	Bloc de boot (+ identification du disque)
1	Superbloc:
	• date de dernière mise à jour du SF
	• nombre de fichiers que peut contenir le SF
	• taille du SF
	• pointeur sur la liste des blocs libres
2	Table des inodes (informations concernant les fichiers)
1 :	
k	
k+1	Blocs de données
:	

La taille d'un bloc dépend du système (exemples : 512 octets, 8192 octets, ...). Le nombre d'inodes (le nombre maximum de fichiers) est déterminé à la création du SF.

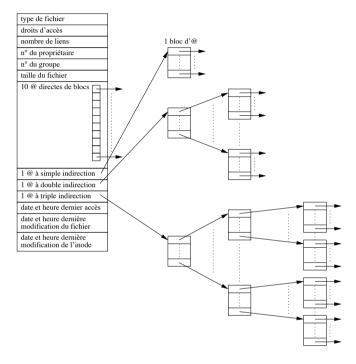
AC - v1.4 - 27/60 - Systèmes 2

4 – Structure interne du système de fichiers d'Unix

4.3 Structure d'un inode

28/60

Chaque inode est repéré dans la table par son numéro (index). Le numéro 2 est la racine du SF. La commande 1s -1i affiche les numéros d'inodes.



♠ Exercice 15

Compléter les zones laissées en pointillés dans le texte suivant.

- \bullet Si taille (fichier) $\leq \ldots$ blocs, les adresses des blocs de données sont entièrement contenues dans l'in ode.
- Si taille(fichier) > . . blocs, le système attribue au fichier un nouveau bloc du disque contenant les adresses (numéros) des blocs de données.

Supposons qu'un bloc peut contenir 256 adresses de blocs.

- Alors, si taille (fichier) \leq blocs, on n'a pas besoin d'utiliser les pointeurs à double et triple in direction.
- De même, si taille(fichier) ≤ blocs, on n'a pas besoin du pointeur à triple indirection.
- La taille maximale d'un fichier est donc, dans ce cas, de blocs.

4.4 Contenu des répertoires

Un répertoire est un fichier qui contient le nom des éléments contenus dans ce répertoire ainsi que leurs inodes. Par exemple, le contenu d'un répertoire contenant les fichiers fich1.txt et fich2.txt et le répertoire REP peut être représenté par la table suivante:

Nom	N° inode
	8186182
	1784306
REP	8333283
fich1.txt	8186211
fich2.txt	8186327

Chapitre 5 Primitives Unix (POSIX.1) de manipulation des fichiers

5.1 Introduction

On appelle « primitives » ou « appels système » des services offerts par le noyau et qui sont activés par des appels de fonctions. On ne s'intéresse ici qu'au sous-ensemble que l'on appelle parfois « entréessorties de bas niveau » qui permettent d'interagir avec le système de gestion des fichiers. On ne verra que quelques fonctions et quelques paramètres (voir L3, M1, etc.).

5.2 Représentation d'un ensemble par un entier

En langage C, on peut représenter un ensemble par un entier dont les bits positionnés à 1 dans sa représentation binaire indiquent les éléments présents. Chaque élément est représenté par une puissance de 2.

Exemple:

```
• Définition du type
```

typedef unsigned char Soupe; /* Ensemble pouvant comporter jusqu'à 8 éléments */

• Définition des éléments

```
#define PATATES 1 /* 00000001 */
#define CAROTTES 2 /* 00000010 */
#define NAVETS 4 /* 00000100 */
#define DIGNONS 8 /* 00001000 */
#define FEVES 16 /* 00010000 */
#define CELERI 64 /* 01000000 */
#define CURGE 128 /* 10000000 */
```

• Déclaration d'un ensemble

Soupe s;

- Construction d'un ensemble: opérateur | (OU logique bit à bit) s=CAROTTES|NAVETS|OIGNONS|POIREAUX; /* 00101110 */
- Détermination de l'appartenance d'un élément à un ensemble : opérateur & (ET logique bit à bit)

```
if (s&OIGNONS)
printf("Il y a des oignons dans la soupe.\n");
```

AC - v1.4 - 31/60 - Systèmes 2

32/60 5 - Primitives Unix (POSIX.1) de manipulation des fichiers

Cette technique est par exemple utilisée au niveau des paramètres de la fonction open qui fait partie de la bibliothèque Unix de manipulation des fichiers.

5.3 Primitives d'entrées-sorties

5.3.1 Ouverture d'un fichier

```
#include <sys/types.h> /* types implémentés différemment selon la version d'Unix */
#include <sys/stat.h> /* constantes symboliques pour les droits d'accès */
#include <fcratl.h> /* constantes symboliques pour les différents types d'ouverture */
int open(const char *chemin, int mode)
int open(const char *chemin, int mode, mode, tdroits)
```

- chemin: chaîne de caractères contenant la désignation du fichier.
- <u>mode</u>: entier obtenu par la combinaison de constantes symboliques grâce à l'opérateur « ou bit à bit » (1) permettant de décrire le mode d'ouverture du fichier. Il doit comporter l'une des constantes suivantes:

```
o O_RDONLY: ouverture en lecture
```

- o O_WRONLY: ouverture en écriture
- o O_RDWR: ouverture en lecture/écriture

auxquelles on rajoute n'importe quelle combinaison des constantes suivantes:

```
o O_APPEND: concaténation des données en fin de fichier
```

- o O_CREAT : création du fichier lorsqu'il n'existe pas
- O_EXCL: provoque une erreur si le fichier existe (open retourne -1)
- o O_TRUNC: efface le contenu du fichier s'il existe
- o O_SYNC: écriture immédiate sur disque en vidant les tampons
- 0 .

• <u>droits</u>: si O_CREAT est utilisé, il faut fournir un troisième paramètre qui indique les droits du fichier créé. Il est construit grâce à l'opérateur | et aux constantes suivantes:

```
o S_IRUSR: lecture autorisée pour le propriétaire
```

- o S_IRGRP: lecture autorisée pour le groupe
- S_IROTH: lecture autorisée pour les autres
- o S_IWUSR: écriture autorisée pour le propriétaire
- o S_IWGRP: écriture autorisée pour le groupe
- o S_IWOTH: écriture autorisée pour les autres
- o S_IXUSR: exécution autorisée pour le propriétaire
- o S_IXGRP: exécution autorisée pour le groupe
- o S_IXOTH: exécution autorisée pour les autres
- o S_IRWXU: lecture, écriture et exécution autorisées pour le propriétaire
- o S_IRWXG: lecture, écriture et exécution autorisées pour le groupe
- o S_IRWXO: lecture, écriture et exécution autorisées pour les autres

Remarque : S_IRWXU == S_IRUSR|S_IWUSR|S_IXUSR

La fonction open retourne le descripteur interne du fichier, c'est-à-dire l'indice (entier naturel) dans la table des descripteurs associée au processus ou -1 en cas de problème.

Remarque: les descripteurs de fichiers correspondant aux unités standard sont les constantes STDIN_FILENO, STDOUT_FILENO et STDERR_FILENO définies dans unistd.h et qui valent respectivement 0, 1 et 2. Ces trois unités standard sont automatiquement ouvertes au lancement du programme.

5.3.2 Fermeture d'un fichier

#include <unistd.h>

int close(int fd)

Cette fonction ferme le fichier de descripteur $\underline{\underline{t}}\underline{d}$ (en fait, elle libère le descripteur $\underline{\underline{t}}\underline{d}$). Elle retourne 0 si la fermeture s'est bien passée ou -1 en cas d'erreur (descripteur incorrect par exemple).

5.3.3 Lecture dans un fichier

#include <unistd.h>

ssize_t read(int fd, void *p, size_t nboctets)

Cette fonction essaie de lire $\underline{\mathtt{nbottets}}$ octets dans le fichier de descripteur $\underline{\mathtt{fd}}$ et les range à partir de l'adresse p. Elle retourne le nombre d'octets effectivement lus.

Remarque : size_t est un type entier retourné par l'opérateur sizeof et ssize_t est un type entier retourné par la fonction read.

5.3.4 Écriture dans un fichier

#include <unistd.h>

ssize_t write(int fd, const void *p, size_t nboctets)

Cette fonction essaie d'écrire <u>nboctets</u> octets situés à partir de l'adresse \underline{p} dans le fichier de descripteur \underline{fd} . Elle retourne le nombre d'octets effectivement écrits.

Exercice 16

Écrire un programme C qui effectue la copie du contenu d'un fichier par blocs de 512 octets. Ce programme prend deux arguments: le nom du fichier source et le nom du fichier destination. Si le fichier destination n'existe pas, il doit être créé avec le droit en lecture pour le propriétaire et le groupe et le droit en écriture pour le propriétaire. S'il existe, son contenu doit être « écrasé ». Le programme doit gérer les codes de retour en cas d'erreur (mauvais nombre de paramètres, fichier inaccessible, etc.).

5.3.5 Contrôle de la position courante

#include <sys/types.h>

#include <unistd.h>

 $\texttt{off_t lseek(int } \underline{\texttt{fd}}, \ \texttt{off_t } \underline{\texttt{decalage}}, \ \texttt{int } \underline{\texttt{origine}})$

Cette fonction modifie la position courante d'un fichier de descripteur <u>fd</u>. La nouvelle position est située à <u>decalage</u> octets de l'origine qui peut être: SEEK_SET (début), SEEK_CUR (position courante) ou SEEK_END (fin). Elle retourne la nouvelle position courante exprimée en nombre d'octets depuis le début du fichier ou -1 en cas d'erreur.

♠ Exercice 17

Retour à une position mémorisée

On suppose que le descripteur fd est associé à un fichier ouvert en lecture.

- Écrire l'instruction permettant de mémoriser dans une variable position la position courante dans le fichier.
- 2. Écrire l'instruction permettant de revenir à une position mémorisée dans la variable position.

AC – v1.4 Systèmes 2

Exercice 18

34/60

Détermination de la taille d'un fichier

Écrire l'instruction permettant de récupérer dans une variable taille la taille (nombre d'octets) d'un fichier ouvert en lecture et de descripteur fd.

♠ Exercice 19

Accès direct au contenu d'un fichier binaire

On suppose que le descripteur \mathtt{fd} est associé à un fichier binaire ouvert en lecture et contenant des int. Écrire une fonction qui retourne le $\mathtt{n}^{\mathrm{ième}}$ int de ce fichier.

5.4 Duplication de descripteur

Chaque processus possède une table de descripteurs de fichiers. Il est possible de dupliquer un descripteur de fichier. Les deux descripteurs désigneront alors le même fichier. Ils pourront être utilisés de manière interchangeable.

```
#include <unistd.h>
int dup(int fd)
int dup2(int fdsource, int fddestination)
```

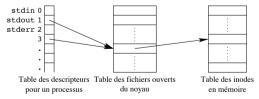
La fonction dup utilise le premier descripteur libre pour le nouveau descripteur. La fonction dup2 duplique le descripteur <u>fdsource</u> dans le descripteur <u>fddestination</u> après l'avoir fermé si nécessaire. Ces fonctions retournent le nouveau descripteur ou -1 en cas de problème (dans ce cas, la fonction perror peut être utilisée).

Exemple: redirection de la sortie standard vers un fichier.

```
int d;
d=open("fich", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
if (d=-1)
    perror("Échec car ");
else
{
    printf("Bonjour\n");
    dup2(d,1); /* ou bien dup2(d, STDOUT_FILENO); ou bien close(1); dup(d); */
    printf("Bonsoir\n");
}
```

Lors de l'exécution de cet extrait de programme, Bonjour est affiché à l'écran et Bonsoir est écrit dans le fichier fich.

Remarque : pour faire une redirection avec écriture en fin de fichier (équivalente à l'utilisation de >> en shell), il suffit de remplacer O_TRUNC par O_APPEND.



♠ Exercice 20

Écrire un programme C qui lance l'exécution d'une commande en réalisant les redirections du type : commande < fich_entree">commande < fich_entree >> fich_sortie

Systèmes 2

Ce programme prend trois arguments: la commande, le nom du fichier vers lequel doit être redirigée l'entrée standard et le nom du fichier vers lequel doit être redirigée la sortie standard (avec ajout en fin de fichier). L'exécution de la commande doit être déléguée à un processus fils. Les redirections doivent être réalisées par duplication des descripteurs de fichiers. Si le fichier de sortie n'existe pas, il doit être créé avec le droit en lecture pour tous et le droit en écriture pour le propriétaire. Le programme doit gérer les codes de retour en cas d'erreur (mauvais nombre de paramètres, fichier inaccessible, etc.).

5.5 Suppression d'un fichier (lien physique)

```
#include <unistd.h>
int unlink(char *designation)
Cette fonction retourne 0 en cas de succès ou -1 en cas d'échec.
La fonction remove de la bibliothèque standard fait appel à unlink.
```

5.6 Accès aux caractéristiques d'un fichier

Il est possible d'accéder aux informations contenues par l'inode d'un fichier grâce aux fonctions suivantes :

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *NomFich, struct stat *pInfos)
int fstat(int DescFich, struct stat *pInfos)
```

AC - v1.4

Ces fonctions récupèrent des informations contenues dans l'inode d'un fichier et les stockent dans la structure dont l'adresse est passée en paramètre. La fonction stat prend en entrée le nom du fichier alors que la fonction fstat utilise le descripteur du fichier, ce qui sous-entend que le fichier doit avoir été ouvert. Ces fonctions retournent -1 en cas de problème ou 0 sinon. La structure struct stat contient (au moins) les champs suivants:

```
struct stat
  dev_t st_dev; /* Id du système de fichiers contenant le fichier
 ino_t st_ino; /* N° de l'inode du fichier
 mode_t st_mode; /* Type et mode du fichier
 nlink_t st_nlink; /* Nb de liens physiques sur le fichier
 uid_t st_uid; /* Id de l'utilisateur propriétaire du fichier
 gid_t st_gid; /* Id du groupe du propriétaire du fichier
 off_t st_size; /* Taille du fichier en octets
 time_t st_atime; /* Instant du dernier accès au fichier
                   /* (création, lecture, écriture,
                   /* lecture des caractéristiques,...)
 time_t st_mtime; /* Instant de la dernière modification des données
 time_t st_ctime; /* Instant de la dernière modification des caractéristiques */
                   /* (création, écriture, changement des droits,
                   /* changement de propriétaire, ...)
};
```

- Le champ st_mode contient plusieurs informations dont le type du fichier et ses droits d'accès.
 - Pour obtenir le type du fichier, on fait appel aux macros suivantes (on suppose que Infos est une variable de type struct stat):

```
\begin{split} & \texttt{S\_ISREG(Infos.st\_mode)} \neq 0 \text{ si c'est un fichier ordinaire,} \\ & \texttt{S\_ISFIFO(Infos.st\_mode)} \neq 0 \text{ si c'est un fichier spécial tube,} \\ & \texttt{S\_ISCHR(Infos.st\_mode)} \neq 0 \text{ si c'est un fichier spécial caractère,} \\ & \texttt{S\_ISBLK(Infos.st\_mode)} \neq 0 \text{ si c'est un fichier spécial bloc,} \\ & \texttt{S\_ISDIR(Infos.st\_mode)} \neq 0 \text{ si c'est un répertoire.} \end{split}
```

 $\circ\,$ Pour obtenir les droits d'accès, on utilise l'opérateur & et les constantes correspondantes.

Exemple: (Infos.st_mode & S_IRUSR) $\neq 0$ si le propriétaire a le droit en lecture sur le fichier.

• Les champs st_atime, st_mtime et st_ctime contiennent des instants (dates et heures) sous la forme d'entiers (nombre de secondes écoulées depuis le 01/01/1970, c'est-à-dire le « big-bang » Unix). Pour les afficher sous une forme plus lisible, on peut utiliser la fonction standard ctime: #include <time.h>

```
char *ctime(const time_t *pTemps)
Exemple : printf("%s",ctime(&Infos.st_atime));
```

5.7 Parcours d'un répertoire

Le parcours d'un répertoire peut se faire en utilisant les directives : #include <sys/types.h>

```
#include <sys/types.n>
#include <dirent.h>
et en utilisant les fonctions suivantes:
```

- DIR *opendir(const char *DesignationRep)
 ouvre le répertoire de nom DesignationRep et retourne son identificateur ou NULL en cas de problème.
- struct dirent *readdir(DIR *Rep)
 retourne un pointeur vers une structure qui représente l'élément courant ou NULL en cas de
 problème ou si la fin du répertoire est atteinte, et passe à l'élément suivant. La structure
 struct dirent contient au moins le champ d_name, de type tableau de caractères, qui est
 le nom de l'élément (fichier ou répertoire) sous la forme d'une chaîne de caractères.

Attention : la fonction readdir peut ou non, selon les implémentations (c'est le cas sur marine), retourner les éléments de nom "." et ".."

- void rewinddir(DIR *IdRep)
 remet la position courante au début du répertoire d'identificateur IdRep.
- int closedir(DIR *IdRep)
 ferme le répertoire d'identificateur IdRep et retourne -1 en cas de problème ou 0 sinon.

Exercice 21

36/60

```
Écrire une fonction d'en-tête:
```

```
long TailleRepCour(void)
```

qui retourne le nombre total d'octets occupés par les fichiers présents dans le répertoire courant (les éventuels sous-répertoires ne doivent pas être examinés).

5.8 Parcours d'une sous-arborescence

La fonction de parcours d'une sous-arborescence de racine Rep doit être récursive et suivre l'algorithme général suivant :

```
Parcourir(Rep)
{
   Pour tout Element de Rep
   Si Element est différent de "." et de ".."
   Si Rep/Element est un répertoire
        Parcourir(Rep/Element);
   Sinon
        Traiter(Rep/Element);
}
```

L'algorithme précédent nécessite de construire des désignations, c'est-à-dire de manipuler des chaînes de caractères. Pour cela, on utilise les fonctions de manipulation des chaînes de caractères de la bibliothèque standard.

On peut utiliser la fonction strcpy qui permet de copier une chaîne de caractères et la fonction strcat qui permet de concaténer deux chaînes de caractères.

Exemple:

```
#include <string.h>
...
char ch1[512],ch2[512];
strcpy(ch1,"Il fait "); /* ch1 contient "Il fait " */
strcpy(ch2,ch1); /* ch1 contient "Il fait " et ch2 contient "Il fait " */
strcpy(ch1,"beau "); /* ch1 contient "beau" et ch2 contient "Il fait " */
strcat(ch2,ch1); /* ch2 contient "Il fait beau" et ch1 contient "beau" */
```

Pour comparer deux chaînes de caractères, on peut utiliser la fonction strcmp qui retourne 0 quand les deux chaînes passées en paramètres sont identiques ou un entier non nul quand elles sont différentes.

♠ Exercice 22

Écrire une fonction qui affiche à l'écran les désignations de tous les fichiers contenus dans la sous-arborescence dont le répertoire racine est passé en paramètre.

Supposons que l'on se trouve dans la configuration suivante : le répertoire TEST contient le fichier fich.txt, le répertoire REP1 et le répertoire REP2; le répertoire REP1 contient les fichiers fich11.txt et fich12.txt; le répertoire REP2 contient le fichier fich2.txt. Si le répertoire TEST est passé en paramètre à la fonction, elle doit alors afficher:

```
TEST/REP1/fich11.txt
TEST/REP1/fich12.txt
TEST/REP2/fich2.txt
TEST/fich.txt
```



Chapitre 6 Fonctions de la bibliothèque standard pour la manipulation des fichiers

6.1 Introduction

Ce chapitre décrit les outils de gestion des fichiers de la bibliothèque standard du langage C ANSI. Même si, au bout du compte, tout est binaire, on distingue généralement deux types de fichiers :

- les fichiers de texte: chaque octet qui les compose représente un caractère et ces caractères sont organisés en lignes (le contenu du fichier apparaît de manière lisible dans la fenêtre d'un éditeur de texte);
- les fichiers binaires: ce sont tous les autres fichiers (leur contenu apparaît sous la forme de caractères illisibles dans la fenêtre d'un éditeur).

La différence entre les deux types de fichiers réside dans la manière dont les informations y sont codées. Exemple: l'entier 24 est codé par:

- 00110010 00110100 dans un fichier de texte: un octet pour le code ASCII du caractère 2, qui vaut 50, et un octet pour le code ASCII du caractère 4, qui vaut 52;
- \bullet 00000000 00000000 00000000 00011000 dans un fichier binaire si les entiers sont codés sur quatre octets.

Même si ce n'est pas une obligation, on n'utilise généralement pas les mêmes fonctions pour réaliser des entrées-sorties dans les fichiers de texte et les fichiers binaires.

Les fonctions de gestion des fichiers nécessitent la directive #include <stdio.h>.

6.2 Ouverture

La manipulation d'un fichier va se faire à travers un identificateur de fichier (aussi appelé flux ou stream) de type FILE *.

La fonction

AC - v1.4

FILE *fopen(const char *nom, const char *mode)

ouvre le fichier de désignation $\underline{\mathtt{nom}}$ dans le mode $\underline{\mathtt{mode}}$ et retourne son identificateur ou NULL s'il y a un problème.

Le mode est une chaîne de un, deux ou trois caractères:

- le premier caractère indique si le fichier doit être ouvert en lecture (r pour read), en écriture (w pour write) ou en écriture en fin de fichier (a pour append):
 - \circ r: si le fichier existe, on est positionné au début ; sinon, l'ouverture échoue et fopen retourne NULL ;
 - o w: si le fichier existe, son contenu est écrasé; sinon, le fichier est créé;

- 39/60 - Systèmes 2

0/60 6 – Fonctions de la bibliothèque standard pour la manipulation des fichiers

- o a: si le fichier existe, on est positionné à la fin; sinon, le fichier est créé;
- le deuxième caractère, qui peut être ajouté au premier, est le signe + qui précise que l'on souhaite effectuer une mise à jour (lecture et écriture) :
- le troisième caractère indique si le fichier doit être considéré comme un fichier de texte (t pour text), qui est le type par défaut, ou comme un fichier binaire (b pour binary).

```
Exemple: FILE *id_fich;
    id_fich=fopen("data.txt","rt");
    if (id_fich==NULL) perror("data.txt ");
```

Remarque: trois identificateurs de fichiers particuliers sont disponibles et n'ont besoin ni d'être ouverts, ni d'être fermés: stdin (entrée standard), stdout (sortie standard) et stderr (sortie standard des erreurs).

6.3 Fermeture

La fonction

```
int fclose(FILE *id_fich)
```

ferme le fichier d'identificateur $\underline{\mathtt{id_fich}}$ et retourne 0, si tout se passe bien, ou \mathtt{EOF} s'il y a un problème. Remarques :

- Il ne faut fermer un fichier que s'il est ouvert.
- Il est conseillé de fermer un fichier dès que son traitement est terminé.

6.4 Lecture

6.4.1 Lecture dans un fichier de texte

6.4.1.1 Lecture d'un caractère

La fonction

```
int fgetc(FILE *id_fich)
```

lit le caractère courant dans le fichier d'identificateur <u>id.fich</u> et retourne ce caractère ou EOF si la fin

Remarques : la macro getc effectue la même chose que fgetc et la macro getchar effectue la même chose que fgetc(stdin).

6.4.1.2 Lecture d'une chaîne de caractères

La fonction

```
char *fgets(char *ch, int n, FILE *id_fich)
```

lit au maximum $\underline{n}-1$ caractères dans le fichier d'identificateur $\underline{id.fich}$, s'arrête si elle rencontre le caractère '\n', les range (y compris l'éventuel '\n') dans la chaîne d'adresse \underline{ch} en complétant par le caractère '\0'. Elle retourne l'adresse de la chaîne ou NULL s'il y a un problème ou si la fin de fichier a été atteinte.

6.4.1.3 Lecture formatée

La fonction

```
int fscanf(FILE *id_fich, const char *format[,liste_d_expressions])
```

se comporte de la même manière que scanf mais effectue la lecture dans le fichier d'identificateur id_fich au lieu du clavier.

Sustèmes 2 AC - v1.4

6.5 - Écriture 41/60

6.4.2 Lecture dans un fichier binaire

```
La fonction
```

```
size_t fread(void *pt, size_t taille, size_t nb_infos, FILE *id_fich)
lit dans le fichier d'identificateur idfich nb_infos informations (blocs d'octets) de taille octets
chacunes et les range en mémoire à l'adresse pt (la zone mémoire doit avoir été préalablement allouée).
Elle retourne le nombre d'informations effectivement lues. Ce nombre peut être inférieur à nb_infos
si la fin du fichier est atteinte.
```

6.5 Écriture

6.5.1 Écriture dans un fichier de texte 6.5.1.1 Écriture d'un caractère

La fonction

```
int fputc(int c, FILE *id_fich)
```

écrit dans le fichier d'identificateur id_fich le caractère c (convertit en unsigned char) et retourne le caractère écrit ou EOF s'il y a un problème.

6.5.1.2 Écriture d'une chaîne de caractères

La fonction

```
int fputs(const char *ch, FILE *id_fich)
```

écrit dans le fichier d'identificateur id.fich la chaîne d'adresse ch et retourne EOF s'il y a un problème ou une valeur négative dans le cas contraire.

6.5.1.3 Écriture formatée

La fonction

```
int fprintf(FILE *id_fich, const char *format[,liste_d_expressions])
se comporte de la même manière que printf mais effectue l'écriture dans le fichier d'identificateur
id_fich au lieu de l'écran.
```

Remarque: on utilise très souvent fprintf pour afficher les messages d'erreur sur stderr.

```
Exemple: FILE *f;
                                                                                      4 11
          f=fopen("toto.txt","rt");
          if (f==NULL) fprintf(stderr, "Fichier toto.txt inexistant.\n");
```

6.5.2 Écriture dans un fichier binaire

La fonction

```
size_t fwrite(void *pt, size_t taille, size_t nb_infos, FILE *id_fich)
écrit dans le fichier d'identificateur id.fich nb_infos informations (blocs d'octets) de taille octets
chacunes situées en mémoire à l'adresse pt. Elle retourne le nombre d'informations effectivement
```

écrites. Ce nombre peut être inférieur à nb_infos dans le cas d'un disque saturé.

6.6 Gestion de la position courante

À chaque fichier ouvert est associé un « pointeur de fichier » qui donne la position courante dans le fichier, c'est-à-dire le rang du prochain octet à lire ou à écrire. Après chaque opération de lecture ou

AC - v1.4Systèmes 2

6 - FONCTIONS DE LA BIBLIOTHÈQUE STANDARD POUR LA MANIPULATION DES FICHIERS

d'écriture, ce « pointeur » est automatiquement incrémenté du nombre d'octets transférés. On parle d'accès séquentiel au fichier.

6.6.1 Détection de fin de fichier

La fonction

```
int feof(FILE *id_fich)
```

retourne une valeur non nulle si la fin du fichier d'identificateur id_fich est atteinte, ou 0 sinon.

Attention: la valeur de retour de feof n'est valide qu'après avoir effectué au moins une lecture.

Remarque: dans le cas d'une lecture dans un fichier binaire, la valeur de retour de la fonction fread peut remplacer avantageusement l'appel à la fonction feof.

Exercice 23

Exemple de lecture séquentielle d'un fichier de texte

En n'utilisant que les fonctions de la bibliothèque standard, écrire un programme qui parcourt le fichier dont le nom est passé en paramètre. On suppose qu'il s'agit d'un fichier de texte contenant des entiers séparés par au moins un caractère séparateur (espace, tabulation, nouvelle ligne...). Le programme doit lire de manière séquentielle chaque entier et l'afficher à l'écran.

6.6.2 Positionnement dans un fichier

Il est possible d'accéder à un fichier de manière directe (accès direct) en modifiant sa position courante grâce à la fonction

int fseek(FILE *id_fich, long decalage, int depart)

- id_fich est l'identificateur du fichier;
 - decalage est le nombre d'octets dont on veut se décaler par rapport à la position depart;
 - depart peut être égal à:
 - o SEEK_SET ou 0 : début de fichier,
 - o SEEK CUR ou 1: position courante dans le fichier.
 - o SEEK END ou 2: fin de fichier.

La fonction fseek retourne 0 si tout s'est bien passé ou -1 sinon, en particuler dans le cas où on essaie de revenir en arrière avant le début du fichier. Néanmoins, quand on essaie d'avancer au-delà de la fin du fichier, fseek retourne 0 (tout se passe comme si on « piétinait » à la fin du fichier).

```
La fonction
void rewind(FILE *id_fich)
place le pointeur du fichier d'identificateur id_fich à son début.
Remarque: rewind(f): \Leftrightarrow fseek(f.0.SEEK SET):
long ftell(FILE *id_fich)
```

retourne la position courante du fichier d'identificateur id_fich ou -1 s'il y a un problème.

4 12

Retour à une position mémorisée

On suppose que l'identificateur f est associé à un fichier ouvert en lecture. En n'utilisant que les fonctions de la bibliothèque standard:

- 1. Écrire l'instruction permettant de mémoriser dans une variable position la position courante
- 2. Écrire l'instruction permettant de revenir à une position mémorisée dans la variable position.

Exercice 25

Détermination de la taille d'un fichier

En n'utilisant que les fonctions de la bibliothèque standard, écrire l'instruction permettant de récupérer dans une variable taille la taille (nombre d'octets) d'un fichier ouvert en lecture et d'identificateur f.

♠ Exercice 26

Accès direct au contenu d'un fichier binaire

On suppose que l'identificateur f est associé à un fichier binaire ouvert en lecture et contenant des int. En n'utilisant que les fonctions de la bibliothèque standard, écrire une fonction qui retourne le $n^{\rm ième}$ int de ce fichier.

6.7 Réouverture d'un identificateur de fichier

```
La fonction
```

FILE *freopen(const char *nom, const char *mode, FILE *id_fich)
ferme le fichier associé à l'identificateur <u>id_fich</u> et ouvre, en l'associant à l'identificateur <u>id_fich</u>, le
fichier nommé <u>nom</u> en mode <u>mode</u>. Cette fonction retourne l'identificateur du fichier ou NULL s'il y a
un problème.

Exemple: redirection de la sortie standard vers un fichier.

```
/* Affichage d'un message sur stdout */
void Bonjour(void)
{
    printf("Bonjour !\n");
}
...
Bonjour(); /* Affiche Bonjour ! à l'écran */
freopen("fich.txt","wt",stdout);
Bonjour(); /* Écrit Bonjour ! dans le fichier fich.txt */
...
```

6.8 Suppression d'un fichier

La fonction

int remove(const char *nom_fichier)

 $supprime \ le \ fichier \ de \ nom \ \underline{{\tt nom_fichier}} \ et \ retourne \ 0 \ ou \ une \ valeur \ non \ nulle \ s'il \ y \ a \ un \ problème.$

6.9 Conseils

Le tableau ci-dessous fait une synthèse des principales fonctions généralement utilisées pour effectuer des lectures, des écritures et pour détecter la fin de fichier en fonction du type de fichier.

		Lecture	Écriture							
Type fichier	Mode ouverture	Fonctions	Détection fin de fichier	Modes ouverture	Fonctions					
Texte	"rt"	fgetc, fgets fscanf	feof	"wt", "at"	fputc, fputs fprintf					
Binaire	"rb"	fread	Valeur retournée par fread	"wb", "ab"	fwrite					

6.10 Remarques sur les fichiers binaires

- Accès plus rapide au contenu d'un fichier binaire:
 - o pas de conversion des valeurs numériques;
 - o taille fixe des informations, donc pas de séparateur à gérer.
- \bullet Accès direct plus naturel. Par exemple, si on veut lire les 5 derniers entiers d'un fichier :

fseek(id_fich,-5*sizeof(int),SEEK_END);

Avec un fichier de texte, c'est beaucoup plus compliqué...

Remarque: les fonctions fseek et ftell sont surtout utiles pour des fichiers binaires.

- Mais, problèmes de portabilité: pour un type donné, la représentation peut varier au niveau:
 - o du nombre d'octets occupés (format);
 - du codage;
 - o de l'ordre des octets (little endian / big endian).

Exercice 27

Écrire une fonction d'en-tête:

```
int Text2Bin(char NomFichSource[], char NomFichDest[])
```

qui lit les entiers contenus dans le fichier de texte NomFichSource et les écrit, sous forme de int, dans le fichier binaire NomFichDest. On suppose que, dans le fichier de texte, les entiers sont séparés par au moins un caractère séparateur. Si le fichier NomFichDest n'existe pas, il doit être créé. S'il existe, son contenu doit être « écrasé ». Cette fonction doit retourner le nombre d'entiers qui ont été convertis ou un nombre négatif en cas d'erreur.

♠ Exercice 28

Écrire une fonction d'en-tête:

```
int Bin2Text(char NomFichSource[], char NomFichDest[])
```

qui effectue la conversion inverse de la fonction Text2Bin, c'est-à-dire qui lit les int contenus dans le fichier binaire NomFichSource et les écrit, un par ligne, dans le fichier de texte NomFichDest. Si le fichier NomFichDest n'existe pas, il doit être créé. S'il existe, son contenu doit être « écrasé ». Cette fonction doit retourner le nombre d'entiers qui ont été convertis ou un nombre négatif en cas d'erreur.

♠ Exercice 29

Écrire une fonction d'en-tête:

```
int Nieme(char NomFich[], int n)
```

qui retourne le n^{ème} int du fichier binaire NomFich ou un nombre négatif en cas d'erreur (on suppose que les entiers contenus dans le fichier sont positifs).

Chapitre 7 Sujets des travaux dirigés sur machine

SÉANCE 1

Processus Unix: terminaux, processus et signaux en shell de Bourne

Introduction

Gestion des terminaux

Un terminal correspond à une fenêtre dans laquelle s'exécute un interpréteur de commandes. Chaque terminal est représenté par un fichier spécial situé sous le répertoire /dev. La commande tty donne le chemin d'accès à ce fichier. Un terminal a un certain nombre de caractéristiques qui peuvent être affichées grâce à la commande stty -a. Parmi ces caractéristiques, les caractères de contrôle permettent d'associer une action à une combinaison de touches du type Ctrl-touche (c'est-à-dire qu'il faut maintenir la touche Ctrl enfoncée et appuyer sur la touche touche). Par exemple, par défaut, la combinaison de touches Ctrl-c permet d'interrompre l'exécution d'un programme. En fait, cela provoque l'envoi du signal SIGINT au processus qui s'exécute au premier plan (voir plus loin). Pour pouvoir utiliser une autre combinaison de touches pour réaliser la même action, on peut faire appel à la commande:

stty action '^touche'

L'action correspondant à Ctrl-c est intr.

Gestion des processus

Un processus est un programme en cours d'exécution. La commande ps permet d'obtenir des informations sur les processus (par défaut, uniquement ceux qui sont contrôlés par le terminal). Elle accepte, entre autres, les options suivantes:

- -e: affiche les informations sur tous les processus;
- -a: affiche les informations sur tous les processus sauf ceux qui ne sont pas associés à un terminal;
- -f: affiche des informations détaillées:
- -1: affiche des informations très détaillées;
- -u "<u>liste d'utilisateurs</u>" : affiche les informations sur les processus appartenant aux utilisateurs mentionnés.

Sur marine, les informations très détaillées affichées sont les suivantes:

- F: obsolète:
- S: état des processus

0: en train de s'exécuter sur le processeur;

AC - v1.4 - 45/60 - Systèmes 2

7 – Sujets des travaux dirigés sur machine

```
S: endormi ou bloqué (attend un événement pour continuer);
     Z: zombie (terminé alors que son père n'a pas encore pris connaissance de sa terminaison):
     T: stoppé (par le signal SIGSTOP par exemple);
UID: numéro ou nom du propriétaire du processus;
PID: numéro du processus;
PPID: numéro du processus père;
C: obsolète:
PRI: priorité (plus le nombre est grand, plus la priorité est basse);
NI: gentillesse (plus le nombre est grand, moins l'accès du processus au processeur est fréquent);
ADDR: adresse mémoire du processus;
SZ: taille du processus en Ko (code + données + pile):
WCHAN: nom de la fonction du noyau du système dans laquelle le processus est endormi (bloqué);
STIME: heure de lancement du processus;
TTY: terminal de contrôle du processus:
TIME: temps d'exécution cumulé du processus;
CMD: commande qui a donné naissance au processus.
```

La commande top permet d'afficher des informations concernant les processus qui utilisent le plus le processeur. Ces informations sont mises à jour régulièrement. Sur marine, la commande

top -Uutilisateur permet de n'afficher des informations que sur les processus appartenant à utilisateur.

La commande sleep <u>sec</u> suspend l'exécution pendant <u>sec</u> secondes. Par exemple, (sleep 30; echo "C'est fini !")& affiche au bout de 30 secondes le message C'est fini !

Gestion des jobs

46/60

La plupart des interpréteurs de commandes intègrent un « contrôleur des jobs ». Pour le shell de Bourne, il faut explicitement l'activer en lançant la commande jsh.

Une commande ou un enchaînement de commandes (pipeline) constituent un job. Un job peut être dans l'un des états suivants:

- Au premier plan: il peut effectuer des lectures et des écritures sur son terminal de contrôle. À un instant donné, il peut y avoir au plus un job au premier plan.
- En arrière-plan: il ne peut pas effectuer de lecture sur le terminal mais il peut effectuer des écritures si les caractéristiques du terminal le permettent. À un instant donné, il peut y avoir plusieurs jobs en arrière-plan. Lancer un job en arrière-plan laisse l'interpréteur disponible pour lancer une autre commande: ceci n'est pas le cas quand on lance un job au premier plan.
- Stoppé: il est dans un état suspendu.

Chaque job lancé dans un shell reçoit un numéro (entier positif) qui peut servir à l'identifier. L'identification d'un job peut se faire grâce au caractère ¼ immédiatement suivi du numéro du job.

La commande jobs permet d'afficher le numéro, l'état et la commande correspondant aux jobs du shell courant.

Le caractère & en fin de commande permet de créer un job en arrière-plan.

La frappe de la combinaison de touches Ctrl-z permet de stopper le job au premier plan en lui envoyant le signal SIGSTOP.

La commande stop %id_job stoppe l'exécution d'un job en arrière-plan.

La commande bg [%id_job] permet de reprendre l'exécution en arrière-plan d'un job stoppé. Si bg est appelée sans argument, c'est le job courant (celui qui vient d'être stoppé) qui est considéré.

La commande fg [%id_job] permet de reprendre l'exécution au premier plan d'un job stoppé. Si fg est appelée sans argument, c'est le job courant (celui qui vient d'être stoppé) qui est considéré.

Remarque: la commande kill accepte un identificateur de job en argument.

Signaux et processus

Le mécanisme des signaux permet de gérer des événements pouvant survenir dans la vie d'un processus. Un signal peut être envoyé à un processus:

- lors d'une erreur interne au processus (erreur arithmétique, violation de protection mémoire, ...);
- par la frappe d'une combinaison de touches (par exemple, la combinaison Ctrl-c effectuée depuis un terminal envoie le signal SIGINT au processus qui a été lancé depuis ce terminal au premier plan);
- par un autre processus (commande ou fonction kill).

Le mécanisme des signaux permet par exemple de gérer la synchronisation entre processus. L'utilisation classique des signaux vous sera présentée en L3. Ici, nous allons seulement voir un exemple de prise en compte de l'arrivée d'un signal dans un script shell.

Émission d'un signal

Par défaut, les caractéristiques d'un terminal permettent d'envoyer un certain nombre de signaux au processus lancé au premier plan. Par exemple, la combinaison de touches Ctrl-\ envoie le signal SIGQUIT.

La commande kill -<u>signal</u> <u>pid</u> envoie le signal <u>signal</u> au processus d'identificateur <u>pid</u>.

La commande kill -l affiche la liste des signaux disponibles.

Réception d'un signal

Quand un processus recoit un signal, il a trois comportements par défaut selon le signal recu:

- ignorer le signal,
- terminer son exécution,
- terminer son exécution en créant un fichier core¹ (c'est par exemple le cas avec la combinaison de touches Ctrl-\ qui envoie le signal SIGQUIT).

On peut changer ce comportement par défaut en « déroutant » le signal (ce n'est pas possible pour tous les signaux, comme par exemple SIGKILL et SIGSTOP). En shell, un tel déroutement est possible grâce à la commande trap.

La commande trap [[commande] signal [...]] permet au shell courant (en général, à un script) d'exécuter la commande commande à l'arrivée des signaux désignés. Cette désignation peut être numérique (à chaque signal est associé un entier positif) ou symbolique (voir kill -1). Après l'exécution de la commande, l'exécution du shell reprend là où elle avait été interrompue sauf si la commande exit est rencontrée. Si la commande exit est rencontrée. Si la commande exit est est est signaux désignés reprennent leur comportement original (qu'ils avaient au lancement du shell). Si le numéro du signal est 0, alors la commande est exécutée à la sortie du shell (à la fin de l'exécution du script). La commande trap sans argument affiche la liste des commandes associées à chaque numéro de signal qui est dérouté.

Attention: le comportement de cette commande peut varier d'un shell à l'autre.

48/60

7 - SUJETS DES TRAVAUX DIRIGÉS SUR MACHINE

Exercices

Exercice 1

- Afficher les informations détaillées, puis très détaillées, concernant tous les processus vous appartenant.
- 2. Afficher le nombre total de processus issus de la commande mozilla présents sur marine.
- Lancer un interpréteur de shell de Bourne en activant le contrôleur des jobs. Lancer la commande xterm en arrière-plan et mettre de côté la fenêtre obtenue.
- Lancer l'éditeur asedit au premier plan, puis stopper son exécution. Enfin, reprendre son exécution en arrière-plan.
- Vérifier l'état de vos processus en affichant les informations très détaillées concernant les processus du terminal. Vérifier l'état de vos jobs.
- 6. Stopper l'exécution de l'éditeur et refaire les mêmes vérifications que dans la question précédente.
- 7. Reprendre l'exécution de l'éditeur au premier plan. Quitter l'éditeur et la fenêtre xterm.

Exercice 2

- Depuis un autre terminal (obtenu par exemple par la commande xterm&), lancer la commande top afin de vérifier l'état de vos processus tout au long des manipulations de cet exercice (la touche q permet de terminer l'exécution de la commande top).
- Écrire un script en shell de Bourne qui affiche l'heure en boucle sur stdout. Pour cela, on utilisera une boucle infinie (while true...) et on fera une pause d'une seconde entre chaque affichage.
- Vérifier l'effet de l'envoi du signal SIGINT (signal numéro 2) sur l'exécution de ce script en utilisant les touches Ctrl-c.
- 4. Associer l'envoi du signal SIGINT à la combinaison de touches Ctrl-a. Relancer le script précédent et vérifier les effets de cette association.
- 5. Compléter le script de telle sorte que, lorsqu'il recevra le signal SIGINT, un message soit affiché sur stdout et le script continue à s'exécuter. Pour arrêter son exécution, on pourra lui envoyer le signal SIGQUIT (signal numéro 3) avec la combinaison de touches Ctrl-\.
- Depuis un autre terminal, vérifier les effets de l'envoi des signaux SIGINT, SIGQUIT et SIGKILL (signal numéro 9) sur le script précédent.

Exercice 3

La commande trap peut être utilisée pour gérer l'effacement des fichiers temporaires dans les scripts. En effet, par défaut, si l'utilisateur interrompt l'exécution du script en tapant la combinaison de touches Ctrl-c, les éventuels fichiers temporaires créés par le script ne seront pas effacés.

- 1. Écrire un script qui lit un texte tapé au clavier, l'écrit dans un fichier temporaire, l'affiche sur stdout et supprime le fichier temporaire. La lecture et l'écriture du texte dans le fichier temporaire se font ligne par ligne. Chaque ligne est lue sur stdin et écrite à la fin du fichier temporaire. La lecture doit s'arrêter dès que l'utilisateur tape une ligne vide. Exécuter le script en le terminant normalement, puis en l'interrompant avec Ctrl-c. Vérifier dans les deux cas la présence ou l'absence du fichier temporaire à l'issue de l'exécution.
- 2. Modifier le script précédent de telle sorte :
 - qu'il retourne 0 en cas de fin normale et 1 en cas d'interruption par le signal SIGINT,
 - qu'il efface le fichier temporaire en cas de fin normale mais aussi en cas d'interruption par le signal SIGINT.

Depuis l'interpréteur de commandes, vérifiez le code retourné par le script avec la commande echo \$?

^{1.} Un fichier **core** contient des informations sur le processus au moment de sa terminaison et peut être analysé par certains outils de mise au point.

Séance 2 49/60

SÉANCE 2

Processus Unix: interface de programmation POSIX

Conseils pour compiler les programmes C

Pour limiter le nombre d'erreurs à l'exécution, vous pouvez vous inspirer de l'exemple suivant : gcc -ansi -pedantic -Wall -Werror monprog.c -o monprog

dans lequel "-ansi -pedantic" demande à gcc de vérifier que le programme respecte rigoureusement la norme ANSI, "-Wall" demande à gcc d'être plus scrupuleux dans ses vérifications et d'afficher des messages d'avertissement (warnings) dès qu'il a un doute et "-Werror" demande à gcc de considérer les warnings comme des erreurs et donc de ne pas produire le fichier exécutable s'il y a des warnings.

Pour éviter de devoir taper toutes ces options à chaque compilation, vous pouvez définir un alias en ajoutant dans votre fichier .cshrc la ligne suivante:

alias gcc "gcc -ansi -pedantic -Wall -Werror"

Pour prendre en compte cette modification, lancez depuis l'interpréteur la commande : $\verb|source|$. cshrc Pour effectuer la même compilation que dans l'exemple précédent, il suffira alors de taper :

gcc monprog.c -o monprog

Exercices

Exercice 4

- Écrire une fonction qui affiche toutes les informations du processus courant (cf. paragraphe 2.11.1, page 14).
- 2. Écrire un programme permettant de tester cette fonction.
- 3. Écrire une nouvelle version de ce programme de telle sorte que:
 - il crée un processus fils;
 - il affiche les informations (en utilisant la fonction précédente) concernant le père et le fils:
 - le père et le fils affichent un message juste avant de se terminer en indiquant leur code de retour:
 - le père attende, avant de se terminer, la terminaison du fils et affiche le code de retour du fils.

Exercice 5

- Écrire un programme permettant de lancer l'exécution des commandes qui lui sont passées en paramètres. Pour chaque commande, le programme doit:
 - créer un processus fils qui doit lancer l'exécution de la commande grâce à la fonction execvp;
 - attendre que ce fils soit terminé;
 - passer à la commande suivante.

Des messages doivent être affichés à l'écran afin de suivre l'exécution du programme (voir exemple ci-dessous). Chaque message doit être précédé de l'identificateur du processus produisant cet affichage. Voici un exemple d'exécution dans lequel lancer est le nom de la version exécutable du programme demandé:

```
> lancer "sleep 5" pwd date
[1207] J'ai délégué sleep 5 à 1208. J'attends sa fin...
[1208] Je lance sleep 5 :
```

AC-v1.4 Systèmes 2

7 – Sujets des travaux dirigés sur machine

50/60

```
[1207] 1208 terminé.
[1207] J'ai délégué pwd à 1209. J'attends sa fin...
[1209] Je lance pwd:
/users/linfg/linfg0
[1207] 1209 terminé.
[1207] J'ai délégué date à 1210. J'attends sa fin...
[1210] Je lance date :
Thu Jan 7 19:43:35 MEST 2010
[1207] 1210 terminé.
[1207] J'ai fini.
La syntaxe de la fonction execvp (famille exec) est la suivante:
#include <unistd.h>
int execvp(const char *nom_fichier, char *const argv[])
où argy doit être un tableau de pointeurs de caractères semblable à celui qui est utilisé en
paramètre de la fonction main, mais avec NULL comme dernier élément.
Afin de construire un tel tableau correspondant à une commande stockée sous la forme d'une
chaîne de caractères, vous pouvez utiliser la fonction suivante:
#define NBMOTSMAX 20
/* Construction d'un tableau de pointeurs vers le début des mots d'une chaine
 * de caractères en vue de l'utilisation de la fonction execvp.
 * Retourne le nombre de mots trouves.
int Decoupe(char Chaine[], char *pMots[])
  char *p:
  int NbMots=0:
  p=Chaine; /* On commence par le debut */
  /* Tant que la fin de la chaine n'est pas atteinte et qu'on ne deborde pas */
  while ((*p)!='\0' && NbMots<NBMOTSMAX)
    while ((*p)==' ' && (*p)!='\0') p++; /* Recherche du debut du mot */
    if ((*p)=='\0') break; /* Fin de chaine atteinte */
    pMots[NbMots++]=p; /* Rangement de l'adresse du 1er caractere du mot */
    while ((*p)!=' ' && (*p)!='\0') p++; /* Recherche de la fin du mot */
    if ((*p)=='\0') break; /* Fin de chaine atteinte */
    *p='\0'; /* Marquage de la fin du mot */
    p++; /* Passage au caractere suivant */
  pMots[NbMots]=NULL: /* Derniere adresse */
  return NbMots;
Par exemple, si la commande est stockée dans le tableau de caractères Chaine sous la forme
d'une chaîne de caractère (donc terminée par le caractère '\0'), alors il suffit de déclarer un
tableau de pointeurs de caractères:
char *pMots[NBMOTSMAX+1];
```

Sustèmes 2 AC - v1.4

```
d'appeler la fonction:
Decoupe(Chaine,pMots);
et de lancer l'exécution avec:
execvp(pMots[0],pMots);
```

2. Écrire une nouvelle version du programme précédent dans laquelle les processus fils sont tous créés (le père n'attend pas la terminaison d'un fils pour créer le suivant) et, ensuite, le père attend la terminaison de chacun de ses fils.

7 – Sujets des travaux dirigés sur machine

SÉANCE 3

52/60

Gestion de fichiers: bibliothèque Unix

Introduction

On désire manipuler des fichiers binaires dans lesquels on stocke des fiches contenant des informations sur des personnes. Le type associé à une fiche sera le suivant :

```
#define LONG_MAX_NOM 20
typedef struct
{
   char Nom[LONG_MAX_NOM+1];
   int Age;
   int NbEnfants;
} Infos;
```

Exercices

Exercice 6

Création séquentielle du fichier

En utilisant les fonctions de la bibliothèque Unix, écrire une fonction d'en-tête:

int Creation(char NomFichier[])

qui effectue la création du fichier dont le nom est passé en paramètre à partir de données tapées au clavier. Les grandes étapes à suivre sont les suivantes :

- ouverture du fichier,
- saisie et écriture dans le fichier des différentes fiches,
- fermeture du fichier.

Cette fonction doit retourner un entier décrivant la manière dont la création s'est passée : 0 si tout s'est bien passé ou un entier positif décrivant le problème rencontré (utiliser des constantes symboliques).

Écrivez une fonction principale (main) qui vous permet de tester la fonction Creation et que vous ferez évoluer durant la séance.

Afin de vérifier le fonctionnement de votre fonction Creation, il peut être utile d'afficher le contenu d'un fichier binaire avec la commande od (n'hésitez pas à taper man od): l'option -c permet d'afficher chaque octet du fichier comme un caractère; l'option -t d1 permet d'afficher chaque octet du fichier comme un entier en base 10.

Exercice 7

Consultation du fichier par accès direct

En utilisant les fonctions de la bibliothèque Unix, écrire une fonction d'en-tête:

int Consultation(char NomFichier[])

qui permet à l'utilisateur de consulter une ou plusieurs fiches contenues dans le fichier dont le nom est passé en paramètre. La fonction doit effectuer une boucle de consultation dans laquelle on demande à l'utilisateur le numéro de la fiche qu'il souhaite consulter (la première fiche portant le numéro 1) afin de se positionner directement au niveau de la fiche à lire dans le fichier et à afficher à l'écran. Les grandes étapes à suivre sont les suivantes:

- ouverture du fichier,
- affichage du nombre de fiches contenues dans le fichier,

AC-v1.4 Systèmes 2

Séance 3 53/60

• consultation en boucle (saisie du numéro d'une fiche, positionnement direct en début de fiche, lecture de la fiche, affichage du contenu de la fiche),

• fermeture du fichier.

Cette fonction doit retourner un entier ayant le même rôle que celui de l'exercice 6.

Exercice 8

Terminez l'écriture de la fonction principale (main) permettant de tester les deux fonctions précédentes en affichant un menu. On suppose que le nom d'un fichier contient au plus 30 caractères. Si un problème est survenu lors de l'exécution d'une fonction, le programme doit retourner au système d'exploitation le code retourné par la fonction appelée.

7 - Sujets des travaux dirigés sur machine



Gestion de fichiers: bibliothèque Unix

Exercice 9

54/60

Écrire un programme C qui affiche des informations sur les fichiers ou répertoires dont les noms sont passés en paramètres. Pour un fichier, les informations suivantes doivent être affichées sur une ligne : la désignation du fichier, la chaîne de caractères fichier, sa taille et la date et l'heure de dernière modification des données du fichier. Pour un répertoire, les mêmes informations doivent être affichées sur une ligne à l'exception de la chaîne de caractères fichier qui doit être remplacée par la chaîne répertoire.

Exemple:

```
> tp12ex1 tp12ex1.c TEST TEST/fich.txt

tp12ex1.c : fichier 1093 octets Thu May 27 16:50:34 2004

TEST : répertoire 120 octets Thu May 27 14:36:52 2004

TEST/fich.txt : fichier 33 octets Thu May 27 14:36:52 2004
```

Remarque: pour aligner les informations, vous pouvez utiliser les formats %-20s pour la désignation et %81d pour le nombre d'octets (le caractère – permet d'obtenir un affichage cadré à gauche).

Exercice 10

Compléter le programme de l'exercice 9 pour que, dans le cas où aucun paramètre n'est passé au programme, il affiche les informations sur tous les fichiers et répertoires contenus dans le répertoire courant. En revanche, si des paramètres sont passés au programme, il doit avoir le même comportement que le programme de l'exercice 9.

Remarque : le répertoire courant peut être désigné par la chaîne de caractères ".".

Exemple:

> cd TEST
> ../tp12ex2

 .
 : répertoire
 120 octets Thu May 27 14:36:52 2004

 .
 : répertoire
 752 octets Thu May 27 17:18:35 2004

 REP1
 : répertoire
 112 octets Thu May 27 14:37:53 2004

 REP2
 : répertoire
 80 octets Thu May 27 14:38:09 2004

 fich.txt
 : fichier
 33 octets Thu May 27 14:36:52 2004

Exercice 11

En vous inspirant de la fonction de l'exercice 22, page 37, écrire un programme qui affiche les nombres d'octets de tous les fichiers contenus dans la sous-arborescence dont le répertoire racine est passé en paramètre. Cet affichage devra être de la forme suivante:

```
> cd ..
> tp12ex3 TEST
TEST/REP1/fich11.txt : 39 octets
TEST/REP1/fich12.txt : 56 octets
Total TEST/REP1 : 95 octets
Test/REP2/fich2.txt : 74 octets
Total TEST/REP2 : 74 octets
```

AC-v1.4 Systèmes 2

Séance 4 55/60

TEST/fich.txt : 33 octets

TOTAL TEST : 202 octets

La version exécutable sous marine de ce programme est contenue dans le fichier: /users/linfg/linfg0/S4/tp4ex3

56/60

7 – Sujets des travaux dirigés sur machine



Gestion de fichiers: bibliothèque standard

Exercice 12

Écrire un programme C ANSI qui prend le nom d'un fichier de texte en paramètre et affiche le contenu de ce fichier à l'écran en faisant précéder chaque ligne de son numéro, la première ligne portant le numéro 1. On suppose qu'une ligne contient au plus 80 caractères. On utilisera la fonction fgets, décrite à la page 40, pour effectuer la lecture du fichier ligne par ligne. Le programme doit gérer les codes de retour en cas d'erreur (mauvais nombre de paramètres, fichier inaccessible, etc.).

Exercice 13

Écrire un programme C ANSI qui lit le contenu d'un fichier de texte et qui le recopie dans un autre fichier de texte en remplaçant toutes les occurrences d'un caractère par un autre. Ce programme prend trois paramètres : une chaîne contenant deux caractères (le caractère à remplacer suivi de son remplaçant), le nom du fichier de texte source, le nom du fichier de texte destination. Si le fichier destination n'existe pas, il doit être créé. S'il existe, son contenu doit être « écrasé ». Le programme doit gérer les codes de retour en cas d'erreur (mauvais nombre de paramètres, premier paramètre incorrect, fichier inaccessible, etc.).

Exercice 14

Écrire un programme C ANSI qui compte le nombre de mots contenus dans un fichier de texte dont le nom est passé en paramètre. On suppose qu'un mot est constitué d'au plus 50 caractères. La lecture de chaque mot sera réalisée en faisant appel à la fonction fscanf avec le format %s. Le programme doit gérer les codes de retour en cas d'erreur (mauvais nombre de paramètres, fichier inaccessible, etc.).

AC - v1.4

Séance 6 57/60

SÉANCE 6

Gestion de fichiers : bibliothèque standard

Exercice 15

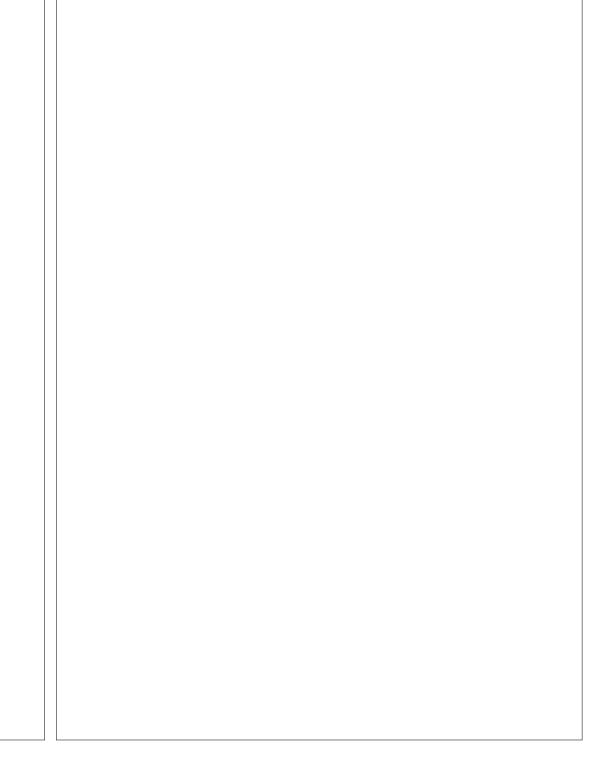
Écrire une nouvelle version du programme des exercices 6, 7 et 8, page 52, en remplaçant, pour les entrées-sorties sur les fichiers, tous les appels aux primitives Unix par des appels aux fonctions de la bibliothèque standard.

Exercice 16

Rajouter au programme précédent les fonctions suivantes :

- modification d'une fiche;
- $\bullet\,$ suppression d'une fiche.





Index

Systèmes 2

&, 46 _exit, 10 bg, 46 close, 33	getlogin, 15 getpgrp, 15 getpid, 15 getppid, 15 getuid, 15
closedir, 36	id, 15
ctime, 36	,
	jobs, 46
dup, 34	1-:11 47
dup2, 34	kill, 47
env, 10	ls, 28
EOF, 40	lseek, 33
exec1, 17	
execlp, 18	main, 9
execvp, 50	O_APPEND, 32
exit, 10, 16	O_CREAT, 32
	O_EXCL, 32
fclose, 40	O_RDONLY, 32
feof, 42	O_RDWR, 32
fg, 46	O_SYNC, 32
fgetc, 40	O_TRUNC, 32
fgets, 40	O_WRONLY, 32
FILE *, 39	open, 32
fopen, 39	opendir, 36
fork, 15	1 ,
fprintf, 41	PATH, 18
fputc, 41	perror, 10
fputs, 41	ps, 45
fread, 41	
freopen, 43	read, 33
fscanf, 40	readdir, 36
fseek, 42	remove, 43
fstat, 35	return, 9
ftell, 42	rewind, 42
fwrite, 41	rewinddir, 36
gcc, 49	S_IRGRP, 32
getchar, 40	S_IROTH, 32
getenv, 10	S_IRUSR, 32
getgid, 15	S_IRWXG, 32

- 59/60 -

AC-v1.4

Index

```
S_IRWXU, 32
S_ISBLK, 36
S_ISCHR, 36
S_ISDIR, 36
S_ISFIFO, 36
S_ISREG, 36
S_IWGRP, 32
S_IWOTH, 32
S_IWUSR, 32
S_IXGRP, 32
S_IXOTH, 32
S_IXUSR, 32
SEEK_CUR, 42
SEEK_END, 42
SEEK_SET, 42
sleep, 17, 46
{\tt st\_atime},\,36
st\_ctime, 36
st_mode, 35
st_mtime, 36
stat, 35
stderr, 40
STDERR_FILENO, 32
\operatorname{stdin}, 40
STDIN_FILENO, 32
stdout, 40
{\tt STDOUT\_FILENO},\,32
stop, 46
strcat, 37
{\tt strcmp},\,37
strcpy, 37
stty, 45
{\tt system},\,9,\,18
top, 46
trap, 47
unlink, 35
wait, 17
WEXITSTATUS, 17
write, 33
```

60/60

S_IRWXO, 32

Systèmes 2 AC – v1.4