

Version en ligne
OFFERTE !
pendant 1 an

+ QUIZ 

Programmation système

Maîtrisez les appels système Linux
avec le langage C

Nouvelle édition



informatique technique

En téléchargement



Programmes d'exemple



 **epsilon**
Collection

Philippe BANQUET



Les éléments à télécharger sont disponibles à l'adresse suivante :
<http://www.editions-eni.fr>
Saisissez la référence de l'ouvrage **EPPRSYL** dans la zone de recherche
et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

Chapitre 1

Introduction aux appels système Linux

- 1. Notion d'appel système 17
 - 1.1 Rôle du noyau 17
 - 1.2 Appel système 18
 - 1.2.1 Exécution d'un appel système 19
 - 1.2.2 Mode utilisateur et mode noyau 19
 - 1.3 Utilisation des appels système en langage C 20
 - 1.3.1 Fonctions enveloppes (wrappers) 20
 - 1.3.2 La fonction syscall() 21
 - 1.3.3 Fonctions de haut niveau 22
- 2. Normes et standards 23
 - 2.1 POSIX (Portable Operating System Interface X) 23
 - 2.2 SUS (Single Unix Specification) 24
 - 2.3 Linux et les normes 25
- 3. Documentation 25
- 4. Portabilité et types 27
- 5. Gestion des erreurs 28
 - 5.1 Code retour et variable errno 28
 - 5.2 Fonctions de gestion des erreurs 28
- 6. Un premier programme système 30
 - 6.1 Documentation de l'appel système 30
 - 6.2 Programme source 32
 - 6.3 Compilation et exécution 33

2 _____ Programmation système

Maîtrisez les appels système Linux avec le langage C

Chapitre 2 Les fichiers

1. Principes généraux	35
1.1 L'interface fichier universelle	35
1.2 Système de fichiers	36
1.3 Chemin d'accès d'un fichier	37
1.3.1 Chemin d'accès absolu ou relatif	37
1.3.2 Répertoire courant, répertoire parent.	37
1.3.3 Analyse d'un chemin d'accès.	38
1.4 Types de fichiers	40
1.4.1 Fichiers ordinaires (regular files).	40
1.4.2 Répertoires	40
1.4.3 Fichiers spéciaux	41
1.4.4 Liens symboliques	41
1.4.5 Tubes nommés et sockets.	41
1.5 Gestion des fichiers ouverts	41
1.5.1 Descripteur de fichier	42
1.5.2 Table des fichiers ouverts par le processus.	42
1.5.3 Table des fichiers ouverts sur le système.	43
1.5.4 Table des inodes ouverts.	43
1.6 Entrées/sorties standards	45
2. Ouverture/fermeture d'un fichier	46
2.1 Appel système open().	46
2.2 Modes d'ouverture	48
2.3 Options d'ouverture	48
2.4 Exemples d'ouvertures de fichiers existants	50
2.4.1 Lecture	50
2.4.2 Écriture.	50
2.4.3 Écriture avec remise à zéro	51
2.4.4 Lecture, sauf lien symbolique	53
2.4.5 Lecture avec remise à zéro.	53

2.5	Création d'un fichier	55
2.5.1	Argument Permissions	55
2.5.2	Gestion de l'umask	57
2.5.3	Exemples de création d'un fichier	57
2.6	Erreurs	60
2.7	Fermeture d'un fichier	61
3.	Lecture d'un fichier	62
4.	Écriture d'un fichier	65
5.	Déplacement dans un fichier	71
6.	Gestion des fichiers ouverts	75
6.1	Obtenir les options d'ouverture	76
6.1.1	Exemple	76
6.2	Modifier les options d'ouverture	79
7.	Duplication de descripteurs	82
7.1	Appel système dup()	82
7.2	Appel système dup2()	85
8.	Gestion des répertoires et des liens	87
8.1	Notion de lien physique	87
8.2	Répertoires et liens physiques	88
8.3	Processus et répertoires	89
8.4	Liens symboliques	89
8.5	Gestion des liens physiques	90
8.5.1	Appel système link()	90
8.5.2	Appel système unlink()	93
8.6	Changement de nom ou déplacement	95
8.7	Gestion des liens symboliques	99
8.7.1	Création d'un lien symbolique : symlink()	99
8.7.2	Lecture du chemin d'accès cible : readlink()	102
8.8	Créer un répertoire : mkdir()	104
8.9	Supprimer un répertoire : rmdir()	106

4 _____ Programmation système

Maîtrisez les appels système Linux avec le langage C

8.10	Lister le contenu d'un répertoire	107
8.10.1	Ouvrir un répertoire : <code>opendir()</code> , <code>fdopendir()</code>	108
8.10.2	Parcourir la liste d'un répertoire : <code>readdir()</code>	109
8.10.3	Repartir du début de la liste : <code>rewinddir()</code>	110
8.10.4	Fermer la liste : <code>closedir()</code>	110
8.10.5	Exemple de parcours d'un répertoire	110
8.11	Gestion du répertoire courant	112
8.11.1	Déterminer le répertoire courant : <code>getcwd()</code>	112
8.11.2	Changer de répertoire courant : <code>chdir()</code> , <code>fchdir()</code>	113
8.11.3	Exemple de changement de répertoire	113
8.12	Changer le répertoire racine du processus : <code>chroot()</code>	115
9.	Lire les attributs d'un fichier	119
9.1	Appels système <code>stat()</code> , <code>lstat()</code> , <code>fstat()</code>	119
9.2	Appel système <code>fstatat()</code>	120
9.3	Structure <code>stat</code>	121
9.4	Exemple.	125
10.	Gestion du contrôle d'accès	127
10.1	Propriétaire et groupe d'un fichier	128
10.2	Changement de propriétaire et de groupe	129
10.3	Droits spéciaux d'un fichier	133
10.3.1	<code>setuserid</code> bit	133
10.3.2	<code>setgroupid</code> bit.	134
10.3.3	<code>sticky</code> bit	134
10.4	Permissions d'accès d'un fichier	135
10.4.1	Types d'accès pour un répertoire	137
10.4.2	Détermination des droits d'accès	137
10.5	Modifier les droits et permissions d'accès.	138
10.6	Exemple de contrôle d'accès.	139
11.	Verrouillage d'un fichier	142
11.1	Types de verrous.	142
11.1.1	Verrouillage consultatif ou impératif.	142
11.1.2	Verrouillage en lecture ou en écriture	144

- 11.2 Gestion des verrous par `fcntl()` 144
 - 11.2.1 Structure `flock` 145
 - 11.2.2 Commandes de gestion de verrou 146
 - 11.2.3 Libération de verrou 147
 - 11.2.4 Étreinte fatale 147
 - 11.2.5 Limites 148
 - 11.2.6 Exemple de gestion de verrou 148
- 12. Fichiers gérés en mémoire (mapping) 152
 - 12.1 Projection en mémoire : `mmap()` 153
 - 12.2 Forcer la mise à jour du fichier projeté : `msync()` 157
 - 12.3 Terminer une projection mémoire : `munmap()` 158

Chapitre 3
Les processus

- 1. Processus et programme 159
 - 1.1 Programme exécutable 159
 - 1.2 Script 160
 - 1.3 Processus 161
 - 1.4 Zones mémoire d'un processus 161
- 2. Attributs d'un processus 162
 - 2.1 Identifiant du processus (PID) 162
 - 2.2 Identifiant du parent du processus (PPID - Parent Process ID) 163
 - 2.3 Comptes utilisateur du processus (rUID, eUID) 163
 - 2.3.1 Identifiant utilisateur réel 164
 - 2.3.2 Identifiant utilisateur effectif 164
 - 2.3.3 Identifiant utilisateur `setuid` sauvegardé 164
 - 2.3.4 Identifiant utilisateur système de fichiers 165
 - 2.4 Processus privilégié 165
 - 2.5 Groupes utilisateurs du processus (GID, eGID) 166
 - 2.5.1 Identifiant groupe utilisateur réel 166
 - 2.5.2 Identifiant groupe utilisateur effectif 166
 - 2.5.3 Identifiant groupe utilisateur `setgid` sauvegardé 167

6 Programmation système

Maîtrisez les appels système Linux avec le langage C

2.5.4	Identifiant groupe utilisateur système de fichiers	167
2.5.5	Groupes utilisateurs supplémentaires	167
2.6	Répertoire courant du processus	167
2.7	Répertoire racine du processus	168
2.8	Valeur d'umask	168
2.9	Sessions et groupes de processus	168
2.9.1	Session de processus	168
2.9.2	Groupe de processus	169
2.9.3	Utilisation	170
2.10	Le pseudosystème de fichiers proc.	170
3.	Environnement d'un processus	172
3.1	Lire l'environnement : <code>getenv()</code>	172
3.2	Modifier l'environnement : <code>setenv()</code>	174
4.	Créer un processus : <code>fork()</code>	176
5.	Gestion des attributs d'un processus.	180
5.1	Identifiants de processus : <code>getpid()</code> , <code>getppid()</code>	180
5.2	Identifiants de groupe et de session : <code>getpgid()</code> , <code>getsid()</code>	181
5.3	Identifiants d'utilisateurs : <code>getuid()</code>	181
5.4	Identifiants de groupes d'utilisateurs : <code>getgid()</code>	182
5.5	Modification des identifiants : <code>setuid()</code> , <code>setgid()</code>	183
5.5.1	Description de <code>setuid()</code>	184
5.5.2	Description de <code>seteuid()</code>	184
5.5.3	Description de <code>setreuid()</code>	184
5.5.4	Description de <code>setresuid()</code>	185
5.5.5	Gestion des identifiants de groupes utilisateurs	185
5.5.6	Gestion des identifiants de systèmes de fichiers	185
5.6	Exemples	185
6.	Terminaison d'un processus.	192
6.1	Appel système <code>_exit()</code>	192
6.2	Appels indirects à <code>_exit()</code>	193
6.2.1	La fonction <code>exit()</code>	193
6.2.2	Terminaison de la fonction <code>main()</code>	194

- 6.3 Exemple..... 194
- 7. Relations entre processus parent et enfant..... 196
 - 7.1 Appel système wait()..... 196
 - 7.2 Le signal SIGCHLD..... 202
 - 7.3 Processus parent/enfant et fichiers..... 203
- 8. Processus zombie..... 204
 - 8.1 Adoption d'un processus orphelin..... 204
 - 8.2 Exemple..... 205
- 9. Chargement et exécution d'un programme externe..... 207
 - 9.1 Appel système execve()..... 207
 - 9.2 Exécution d'un script..... 211
 - 9.3 Gestion des fichiers ouverts..... 212
 - 9.4 La famille de fonctions exec()..... 213
 - 9.5 La fonction system()..... 217

Chapitre 4
Les signaux

- 1. Les principes..... 221
 - 1.1 Qu'est-ce qu'un signal ?..... 221
 - 1.2 Gestion par défaut des signaux..... 222
 - 1.2.1 Signal SIGKILL..... 223
 - 1.2.2 Fichier de vidage mémoire (core dump)..... 223
 - 1.3 Modification de l'effet des signaux..... 224
 - 1.4 Émission d'un signal..... 225
 - 1.5 Réception d'un signal..... 225
- 2. Les types de signaux..... 226
 - 2.1 Signaux d'origine utilisateur..... 226
 - 2.1.1 Signaux liés au clavier..... 226
 - 2.1.2 La commande kill..... 226

8 _____ Programmation système

Maîtrisez les appels système Linux avec le langage C

2.2	Signaux émis à l'initiative du noyau	227
2.2.1	Événements matériels	227
2.2.2	Erreur du processus	227
2.2.3	Événements liés aux processus	228
2.3	Liste des signaux traditionnels Linux	228
3.	Envoi d'un signal.	230
3.1	Appel système kill()	230
3.2	Exemples	232
4.	Traitement des signaux	236
4.1	Appel système signal()	237
4.1.1	Description d'un signal : psignal()	238
4.2	Appel système sigaction()	242
4.3	Bloquer un signal : sigprocmask()	248
4.3.1	Liste des signaux pendants : sigpending()	249
4.3.2	Exemple	250
4.4	Principes d'un gestionnaire de signal	253
4.4.1	Fonctions réentrantes	253
4.4.2	Rôle d'un gestionnaire de signal	253
4.4.3	Protéger les sections critiques	254
4.4.4	Signaux applicatifs	254
4.5	Attendre un signal quelconque : pause()	254
4.6	Traitement du signal SIGCHLD	259
4.6.1	Gestionnaire d'attente de terminaison processus enfant	259
4.6.2	Exemple de gestionnaire SIGCHLD	259
4.6.3	Autres méthodes de gestion de la terminaison des processus enfants	262
4.6.4	Exemples	263
5.	Signaux et démons	268
5.1	Principes d'initialisation d'un daemon System V	269
5.2	Daemon géré par systemd	270
5.3	Daemons et signaux	271

Chapitre 5
Les tubes et les tubes nommés

- 1. Les principes 273
 - 1.1 Flot d'octets (bytestream) 273
 - 1.2 Lecture dans un tube 274
 - 1.3 Écriture dans un tube 274
 - 1.4 Gestion des tubes par le noyau 275
 - 1.4.1 Inode d'un tube 275
 - 1.4.2 Durée de vie d'un tube 276
- 2. Les tubes anonymes (pipes) 276
 - 2.1 Création d'un tube 277
 - 2.2 Attributs d'un tube 279
 - 2.2.1 Lecture des informations de l'inode 279
 - 2.2.2 Mode d'ouverture d'un tube 282
 - 2.2.3 L'appel système pipe2() 282
 - 2.2.4 Taille maximale d'un tube 283
 - 2.2.5 Exemples 284
 - 2.3 Fermeture d'un tube 289
 - 2.3.1 Exemple 289
 - 2.4 Utilisation d'un tube entre plusieurs processus 290
 - 2.4.1 Tube sur la ligne de commande 291
 - 2.4.2 Tube entre processus parent et enfant 291
 - 2.4.3 Exemple 292
 - 2.5 Redirection et tube 295
 - 2.5.1 Principe 295
 - 2.5.2 Exemple 296
 - 2.6 Tube avec une ligne de commande shell 300
 - 2.7 Signaux et tubes 303
 - 2.7.1 Signal SIGPIPE 303
 - 2.7.2 Écriture interrompue 303
 - 2.7.3 Exemple 304

10 _____ Programmation système

Maîtrisez les appels système Linux avec le langage C

3. Les tubes nommés	307
3.1 Création d'un tube nommé	308
3.2 Ouverture d'un tube nommé	310
3.2.1 Ouverture non bloquante	310
3.2.2 Exemple	311
3.3 Utilisation d'un tube nommé	312
3.3.1 Lecture	312
3.3.2 Écriture	313
3.3.3 Fermeture	313
3.3.4 Attributs et options d'ouverture d'un tube nommé . . .	313
3.3.5 Suppression	313
3.4 Exemple	314

Chapitre 6

Communication interprocessus (IPC)

1. Principes de communication interprocessus	321
1.1 IPC System V et IPC POSIX	322
1.2 Fonctions des différents mécanismes d'IPC	322
1.2.1 Segment de mémoire partagée	323
1.2.2 Files d'attente de messages	323
1.2.3 Sémaphores	323
2. Les segments de mémoire partagée	324
2.1 Les segments de mémoire partagée System V	325
2.1.1 Création d'un segment de mémoire partagée : shmget()	326
2.1.2 Identifiants d'un segment de mémoire partagée	327
2.1.3 Générer une clef d'identification : ftok()	328
2.1.4 Exemple de création d'un segment de mémoire partagée	328
2.1.5 Informations sur un segment de mémoire partagée System V : ipc	329
2.1.6 Accès à un segment de mémoire partagée : shmat() . . .	330

- 2.1.7 Contrôle d'un segment
de mémoire partagée : shmctl() 334
- 2.1.8 Détachement d'un segment
de mémoire partagée : shmdt() 339
- 2.2 Les segments de mémoire partagée POSIX 340
 - 2.2.1 Création / ouverture d'un objet
de mémoire partagée : shm_open() 340
 - 2.2.2 Utilisation de l'objet mémoire partagée : mmap() 344
 - 2.2.3 Contrôle d'un objet de mémoire partagée 347
 - 2.2.4 Suppression d'un objet
de mémoire partagée : shm_unlink() 350
- 3. Les files d'attente de messages 351
 - 3.1 Les files d'attente de messages System V 351
 - 3.1.1 Création d'une file d'attente
de messages System V : msgget() 352
 - 3.1.2 Identifiants d'une file d'attente de messages System V 354
 - 3.1.3 Générer une clef d'identification : ftok() 354
 - 3.1.4 Exemple de création d'une file d'attente
de messages System V 355
 - 3.1.5 Informations sur une file d'attente
de messages System V : msgctl() 356
 - 3.1.6 Accès à une file d'attente de messages System V 357
 - 3.1.7 Envoi de messages : msgsnd() 357
 - 3.1.8 Lecture de messages : msgrcv() 360
 - 3.1.9 Contrôle d'une file d'attente
de messages System V : msgctl() 364
 - 3.2 Les files d'attente de messages POSIX 369
 - 3.2.1 Ouverture ou création d'une file d'attente
de messages POSIX : mq_open() 370
 - 3.2.2 Exemple de création d'une file d'attente
de messages POSIX 373
 - 3.2.3 Informations sur une file d'attente
de messages POSIX 374
 - 3.2.4 Envoi de messages : mq_send() 375

12 _____ Programmation système

Maîtrisez les appels système Linux avec le langage C

3.2.5	Lecture de messages : <code>mq_receive()</code>	380
3.2.6	Contrôle d'une file d'attente de messages POSIX : <code>mq_setattr()</code>	383
3.2.7	Fermeture d'une file d'attente de messages POSIX : <code>mq_close()</code>	386
3.2.8	Suppression d'une file d'attente de messages POSIX : <code>mq_unlink()</code>	387
3.2.9	Lecture/écriture de messages avec time-out : <code>mq_timedsend()</code> , <code>mq_timedreceive()</code>	388
3.2.10	Lecture de messages par notification : <code>mq_notify()</code> . . .	392
4.	Les sémaphores	392
4.1	Les sémaphores System V	393
4.1.1	Créer un jeu de sémaphores System V : <code>semget()</code>	394
4.1.2	Identifiants d'un jeu de sémaphores.	396
4.1.3	Générer une clef d'identification : <code>ftok()</code>	396
4.1.4	Exemple de création d'un jeu de sémaphores.	397
4.1.5	Informations sur un jeu de sémaphores System V	398
4.1.6	Contrôler et initialiser un jeu de sémaphores System V : <code>semctl()</code>	399
4.1.7	Utiliser un jeu de sémaphores System V : <code>semop()</code>	407
4.2	Les sémaphores POSIX	412
4.2.1	Créer ou ouvrir un sémaphore POSIX : <code>sem_open()</code> . . .	413
4.2.2	Exemple de création d'un sémaphore POSIX.	415
4.2.3	Informations sur un sémaphore POSIX : <code>sem_getvalue()</code>	417
4.2.4	Utiliser un sémaphore POSIX : <code>sem_post()</code> , <code>sem_wait()</code>	419
4.2.5	Fermer un sémaphore POSIX : <code>sem_close()</code>	423
4.2.6	Supprimer un sémaphore POSIX : <code>sem_unlink()</code>	424
4.2.7	Les sémaphores POSIX anonymes : <code>sem_init()</code> , <code>sem_destroy()</code>	426

Chapitre 7**Communication réseau par les sockets**

1. Principes des sockets	433
1.1 Rôle des sockets	433
1.2 Types de sockets	434
1.2.1 Stream	434
1.2.2 Datagramme	434
1.3 Domaines de communication	435
1.3.1 Domaine IP version 4	435
1.3.2 Domaine IP version 6	435
1.3.3 Domaine Unix	435
2. Gestion des adresses, numéros de port, noms d'hôtes et de services	436
2.1 Représentation des adresses et des données	436
2.1.1 Représentation des données	438
2.2 Gestion des adresses de socket	438
2.2.1 Structure générique sockaddr	438
2.2.2 Structures de stockage des adresses sockets	439
2.2.3 Conversion d'une adresse chaîne de caractères en numérique : inet_pton()	440
2.2.4 Conversion d'une adresse numérique en chaîne de caractères : inet_ntop()	443
2.2.5 Gestion des erreurs : gai_strerror()	445
2.3 Gestion des relations noms et adresses	445
2.3.1 Anciennes fonctions	446
2.3.2 Recherche d'adresse et/ou de numéro de port : getaddrinfo()	450
2.3.3 Recherche de nom d'hôte ou de service : getnameinfo()	457
3. Sockets en mode stream	459
3.1 Créer une socket : socket()	460
3.2 Lier une socket à une adresse : bind()	462
3.3 Écouter une socket : listen()	469
3.4 Attente de connexion : accept()	470

14 _____ Programmation système

Maîtrisez les appels système Linux avec le langage C

3.5	Demande de connexion : connect()	474
3.6	Informations sur une socket :	
	getsockname(), getpeername()	478
3.7	Fermer une connexion close(), shutdown()	482
3.7.1	close()	482
3.7.2	shutdown()	482
3.8	Utilisation d'une connexion socket : appels système fichiers	483
3.8.1	Utilisation avec read(), write()	483
3.8.2	Serveur monoprocessus	484
3.8.3	Serveur multiprocessus	493
3.9	Utilisation d'une connexion socket :	
	appels système spécifiques	501
3.9.1	Réception : recv()	501
3.9.2	Émission : send()	502
3.9.3	Envoi d'un fichier : sendfile ()	503
3.9.4	Exemple client-serveur send()/recv()/sendfile()	504
4.	Sockets en mode datagramme	513
4.1	Créer une socket datagramme : socket()	514
4.2	Lier une socket datagramme à une adresse : bind()	515
4.3	Utilisation de connect() en mode datagramme	518
4.4	Utiliser une socket datagramme	518
4.4.1	Émission d'un datagramme : sendto()	518
4.4.2	Réception d'un datagramme : recvfrom()	519
4.4.3	Envoi de datagramme en broadcast	521
4.5	Exemple de communication par datagrammes	521

Chapitre 8

Les threads

1. Principes des threads	529
1.1 Threads POSIX	530
1.2 Threads Linux	530
2. Threads et processus	530
2.1 Threads et appels système de niveau processus	531
2.1.1 Appel système <code>execve()</code> en contexte multithreads	531
2.1.2 Appel système <code>fork()</code> en contexte multithreads	531
2.1.3 Appel système <code>exit()</code> en contexte multithreads	531
2.1.4 Threads et signaux	531
2.2 Multiprocessus ou multithreads ?	532
2.2.1 Avantages du multithreads	532
2.2.2 Avantages du multiprocessus	533
3. Gestion des threads	533
3.1 Création d'un thread : <code>pthread_create()</code>	534
3.1.1 <code>pthread_create()</code>	534
3.1.2 Identifiant d'un thread : <code>pthread_self()</code>	535
3.2 Terminaison d'un thread : <code>pthread_exit()</code>	538
3.2.1 <code>pthread_exit()</code>	538
3.2.2 Thread zombie	539
3.3 Attendre la terminaison d'un thread : <code>pthread_join()</code>	539
3.4 Détacher un thread : <code>pthread_detach()</code>	543
3.5 Annulation d'un thread : <code>pthread_cancel()</code>	543
4. Synchronisation des threads	547
4.1 Mutex	547
4.1.1 Initialisation d'un mutex	548
4.1.2 Utilisation d'un mutex : <code>pthread_mutex_lock()</code> , <code>pthread_mutex_unlock()</code>	548

16 _____ Programmation système

Maîtrisez les appels système Linux avec le langage C

4.2	Variables conditionnelles	552
4.2.1	Initialisation d'une variable conditionnelle	553
4.2.2	Attendre une variable conditionnelle : pthread_cond_wait().	553
4.2.3	Signaler un changement d'état d'une variable conditionnelle	554
	Index	563



Chapitre 6

Communication interprocessus (IPC)

1. Principes de communication interprocessus

Nous avons étudié dans les chapitres précédents différents mécanismes permettant à des processus de communiquer entre eux, pour échanger des données et pour se synchroniser. Il s'agit des tubes et des fichiers projetés en mémoire, pour les données, des signaux et des verrous de fichiers, pour la synchronisation.

D'autres mécanismes plus élaborés ont été mis en place au cours de l'évolution des systèmes de type Unix : les segments de mémoire partagée, les files d'attente de messages et les sémaphores.

■ Remarque

Les sockets, mécanisme d'échange de données en local, mais surtout à travers un réseau, font l'objet du chapitre suivant.

1.1 IPC System V et IPC POSIX

D'abord implémentés dans le cadre d'Unix System V, le système Unix développé par ATT, ces mécanismes de communication interprocessus (*Inter Process Communication*) ont ensuite été revus et améliorés pour la normalisation POSIX. C'est la raison pour laquelle, bien que les deux ensembles, IPC System V et IPC POSIX, proposent les mêmes types de fonctionnalités, Linux, comme la plupart des systèmes de type Unix, supporte les deux.

On considère généralement que les mécanismes IPC POSIX sont plus simples à programmer et corrigent certains défauts de conception des IPC System V, apparus à l'usage. Ils ont également l'avantage d'être normalisés et compatibles avec le multithreading. Cependant, les mécanismes IPC System V étant apparus plus tôt sont devenus un standard de fait du monde Unix, repris dans les spécifications SUS, et utilisés par de très nombreuses applications.

Il est donc recommandé d'opter pour les IPC POSIX dans le développement de nouvelles applications, mais il est utile, voire nécessaire, de connaître également les IPC System V pour pouvoir maintenir l'existant. C'est la raison pour laquelle nous présenterons les deux ensembles dans ce chapitre.

1.2 Fonctions des différents mécanismes d'IPC

Les trois mécanismes évoqués, segments de mémoire partagée, files d'attente de messages et sémaphores ont des rôles différents et souvent complémentaires.

Les segments de mémoire partagée servent à échanger des données entre processus de manière simple et optimisée, les sémaphores permettent d'assurer la synchronisation des processus, les files d'attente de messages, elles, permettent d'échanger des données de façon ordonnée entre processus, combinant ainsi des fonctions de partage et de synchronisation.

1.2.1 Segment de mémoire partagée

Nous avons vu que le noyau Linux alloue à chaque processus un espace mémoire réservé, que lui seul peut utiliser. Cette protection indispensable des processus les uns par rapport aux autres présente l'inconvénient de rendre difficile le partage d'informations entre processus. C'est la raison pour laquelle Linux propose des méthodes de partage de certaines zones mémoire entre processus : les segments de mémoire partagée. Une fois créées par un appel système spécifique, ces zones mémoire sont accessibles, sous réserve des permissions d'accès, aux autres processus. Le partage des données qu'elles contiennent est alors immédiat et très performant par nature, puisque tout se passe directement en mémoire utilisateur.

L'inconvénient de cette mémoire partagée, c'est qu'elle est délicate à gérer en écriture. Elle nécessite souvent un mécanisme complémentaire pour assurer la synchronisation des accès aux mêmes données par plusieurs processus.

1.2.2 Files d'attente de messages

Ce mécanisme d'échange de données se distingue des tubes par le traitement des données sous forme de messages indépendants et non en flot d'octets. D'autre part, les messages sont gérés par une file d'attente et peuvent être "typés", ce qui permet de les sélectionner selon différents critères et de ne pas forcément les lire dans l'ordre d'arrivée.

1.2.3 Sémaphores

Les sémaphores permettent une synchronisation efficace des processus. Ils sont généralement utilisés en lien avec une ressource partagée, un segment de mémoire par exemple, pour déterminer qui peut accéder à la ressource, dans notre exemple aux données du segment de mémoire, en lecture ou en écriture.

Les processus qui souhaitent obtenir l'accès à des ressources partagées se mettent en attente d'un ou plusieurs sémaphores, gérés par le noyau. Le processus qui obtient le sémaphore peut utiliser les ressources, puis relâcher le sémaphore pour permettre à un autre processus d'y accéder à son tour.

Contrairement aux verrous sur fichiers qui peuvent être rendus contraignants, les sémaphores mettent en place une synchronisation de type **collaboratif**, supposant que tous les processus concernés respectent la règle pour accéder aux ressources partagées.

2. Les segments de mémoire partagée

Les segments de mémoire partagée sont le moyen le plus performant pour partager des données entre processus. Une fois créés, ils sont accessibles à tout processus, sous réserve du contrôle d'accès mis en place par le processus créateur du segment de mémoire partagée. Ils sont persistants au niveau noyau, ce qui signifie que leur durée de vie maximale est celle du noyau, ils sont automatiquement supprimés à l'arrêt du système.

Une fois qu'un processus s'est attaché au segment, il peut accéder à la zone de données correspondante, en lecture et/ou en écriture, pour y gérer des variables, comme si elles étaient dans son espace mémoire virtuel.

La zone mémoire partagée est projetée dans l'espace virtuel du processus. Par conséquent, toute modification effectuée par un processus est immédiatement visible des autres processus attachés au segment. C'est la raison pour laquelle, sauf cas particuliers, il faut mettre en place un mécanisme de synchronisation entre les processus pour éviter les problèmes d'accès concurrents (*race conditions*).

Il existe deux mécanismes de segments de mémoire partagée utilisables avec Linux, ceux d'origine Unix System V et intégrés dans les standards SUS, ceux plus récents mis en place dans le cadre des normes POSIX. Nous allons décrire successivement les deux.

2.1 Les segments de mémoire partagée System V

Les IPC System V, segments de mémoire partagée, files d'attente de messages et sémaphores, partagent une logique et des outils communs. Leur implémentation s'appuie sur des attributs et des méthodes cohérents, facilitant leur utilisation, en contrepartie de la complexité de certains d'entre eux.

Un segment de mémoire partagée System V est un objet géré par le noyau, créé à la demande d'un processus et soumis à des permissions d'accès définies par le processus créateur.

Un processus qui souhaite accéder à un segment de mémoire partagée doit s'y attacher, en lecture et/ou en écriture, en fournissant l'identifiant du segment de mémoire souhaité. Le noyau contrôle les droits du processus et autorise ou non l'accès. Si le processus est autorisé à s'attacher au segment, le noyau en garde la trace. Une fois attaché, le processus reçoit une adresse, dans son espace mémoire virtuel, lui permettant d'accéder à la zone mémoire partagée. Le processus peut donc gérer cette zone relativement à l'adresse fournie, comme il le ferait avec une zone mémoire de son espace virtuel.

Quand un processus n'a plus l'usage du segment de mémoire partagée, il peut s'en détacher. Quand le processus se termine, le noyau le détache automatiquement du segment de mémoire partagée.

Un processus privilégié ou associé à l'UID du créateur ou du propriétaire du segment de mémoire partagée peut demander sa suppression. Cependant, cette suppression ne sera effective que lorsque plus aucun processus ne sera attaché au segment de mémoire partagée.

■ Remarque

La première implémentation des IPC System V sur Linux a été faite via un module dynamique du noyau, gérant un unique appel système pour l'ensemble des mécanismes IPC, `ipc()`. Différentes fonctions enveloppes spécialisées ont été intégrées dans la bibliothèque du langage C, `libc`, pour utiliser les différents objets. Elles doivent être utilisées plutôt que l'appel système proprement dit, pour des raisons de facilité et surtout de compatibilité. Dans la suite du chapitre, nous décrivons donc ces fonctions, comme s'il s'agissait de fonctions enveloppes d'appels système distincts.

2.1.1 Création d'un segment de mémoire partagée : `shmget()`

L'appel système `shmget()` permet de créer un segment de mémoire partagée, ou d'obtenir son identifiant à partir de sa clef, s'il existe.

Syntaxe

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int flags);
```

Arguments

key Clef d'identification du segment de mémoire partagée
size Taille minimum en octets du segment de mémoire partagée
flags Options

Valeur retournée

-1 Erreur, code erreur positionné dans la variable `errno`
!= -1 Identifiant unique du segment de mémoire partagée

Description

Si la clef d'identification fournie n'est pas déjà associée à un segment de mémoire partagée existant, le noyau le crée, de la taille indiquée (arrondie en un multiple de la taille d'une page mémoire), si l'option `IPC_CREAT` est spécifiée.

Une autre possibilité pour créer un segment de mémoire partagée est de spécifier la valeur `IPC_PRIVATE` comme argument `key`, auquel cas il est inutile de positionner l'option `IPC_CREAT`, le noyau créera un segment de mémoire partagée en générant une clef unique, et retournera l'identifiant du segment.

Si la clef est associée à un segment existant, et que la taille mémoire indiquée est inférieure ou égale à celle du segment, l'appel système retourne l'identifiant unique du segment, sauf si les options `IPC_CREAT` et `IPC_EXCL` sont spécifiées, auquel cas l'appel système retourne `-1` et positionne l'erreur `EEXIST` dans `errno`.