Programmation Système sous Linux/Unix

Dr. Ing. Chiheb Ameur ABID

Contact: chiheb.abid@gmail.com

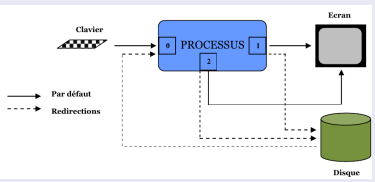
Mars 2023

Plan

Lancement d'une commande

Lancement d'une commande

- Lorsqu'une commande est lancée par le SHELL, trois fichiers ayant chacun un numéro appelé "file discriptor (fd) " sont ouverts par le système.
  - Le fichier stdin ou standard input (fd=0)
  - Le fichier stdout ou standard output (fd=1)
  - Le fichier stderr ou standard error (fd=2)



- Linux permet
  - Rediriger ses résultats vers un fichier
  - Accepter les données à partir d'un fichier

Tubes

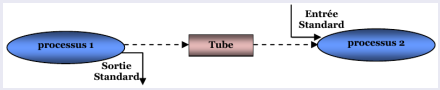
Tubes

- Redirection de la sortie standard
  - `commande 1> fichier`
    `commande > fichier`
  - `commande 1>> fichier`
    `commande >> fichier`
- Redirection de la sortie d'erreur standard
  - `commande 2> fichier`
  - `commande 2>> fichier`
- Redirection de l'entrée standard
  - `commande < fichier`
  - `commande << fichier`

Tubes

Tubes

- Simplifier de l'exécution de plusieurs commandes quand chaque commande doit prendre comme entrée la sortie de la commande qui la précède.



- Les deux lignes de commandes sont équivalentes
  - `commande1 > fichier_temp ; commande2 < fichier_temp ; rm fichier_tmp`
  - `commande1 | commande2`

Tubes

Types

- Il y a deux types de tubes

1. Tube mémoire ou ordinaire ou anonyme
   - Un fichier particulier qui permet de transférer des données entre processus
   - Il est créé dans la mémoire du processus qui les génèrent
   - Les deux processus qui communiqueront via ce système doivent être des descendants d'un même processus créateur (fork()) et que le tube ait été créé avant la duplication pour que les deux processus le connaissent
2. Tube nommé
   - Il correspondent à un fichier (inode) dans lequel les processus pourront lire et écrire
   - Deux processus indépendants peuvent l'utiliser pour transmettre des informations.

Plan

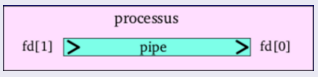
Tubes

Création d'un tube anonyme

- Créer un tube en mémoire et ouvrir le tube créé

```
int pipe(int pipefd[2]);
```

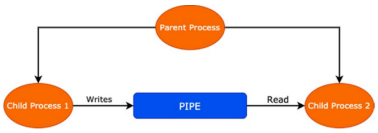
  - La primitive `open()` est inutile dans un pipe mémoire
  - `int fd[2]` : pointeur vers un tableau de deux entiers qui seront remplis par la fonction
  - `fd[0]` contiendra un descripteur permettant l'accès au tube en lecture
  - `fd[1]` contiendra un descripteur permettant l'accès au tube en écriture



Tubes

Étapes d'utilisation d'un tube anonyme

- Appel à `pipe(fd)` pour créer un tube
  - `fd[1]` correspond à l'entrée du tube (accès en écriture)
  - `fd[0]` correspond à la sortie du tube (accès en lecture)
- Création du processus fils avec `fork()`
  - Le nouveau processus a une copie identique de la mémoire et des descripteurs de fichiers
  - Il a donc aussi accès au tube
- Fermer les descripteurs de fichier inutiles
  - Typiquement, un seul processus accède en lecture et un seul accède un écriture
  - Sinon, risque de blocage
- Communiquer en utilisant `read()` et `write()`



Tubes

Écrire dans un tube anonyme

- Par défaut, l'écriture dans un tube est bloquante

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

  - Renvoie le nombre d'octets écrits si OK, sinon -1 et met le code d'erreur dans `errno`
  - `fd` correspond à l'entrée du tube (accès en écriture)
  - `buf` correspond au tampon contenant les données à écrire
  - `nb` spécifie le nombre d'octets à écrire
- Pour rendre l'écriture non bloquante

```
fcntl(fd[1], F_SETFL, O_NONBLOCK);
```


Tubes

Lecture à partir d'un tube anonyme

- La lecture à partir d'un tube est bloquante

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

  - Renvoie le nombre d'octets lus si OK, sinon -1 et met le code d'erreur dans `errno`
  - `fd` correspond à la sortie du tube (accès en lecture)
  - `buf` correspond au tampon pour stocker les données lues
  - `nb` spécifie le nombre d'octets à lire

Tubes

Exemple d'utilisation des tubes anonymes

```
int main(void) {
   pid_t pid_fils;
   int tube[2];
   unsigned char bufferR[256], bufferW[256];
   if (pipe(tube) != 0) { //Création du tube
      fprintf(stderr, "Erreur dans pipe\n");
      exit(1);
   }
   pid_fils = fork(); {/* fork */
   if (pid_fils == -1) {
      fprintf(stderr, "Erreur dans fork\n");
      exit(1);
   }
   if (pid_fils == 0) {
      close(tube[1]);
      read(tube[0], bufferR, BUFFER_SIZE);
      printf("Le fils (%d) a lu : %s\n", getpid(), bufferR);
   }
   else {
      printf("Fermeture sortie dans le pre (pid = %d)\n", getpid());
      close(tube[0]);
      sprintf(bufferW, "Message du pre (%d) au fils", getpid());
      write(tube[1], bufferW, BUFFER_SIZE);
      wait(NULL);
   }
   return 0;
}
```

Tubes

Rediriger les flots d'entrées-sorties vers des tubes

- Lier la sortie d'un tube à `stdin`
  - Tout ce qui sort du tube arrive sur le flot d'entrée standard `stdin`, et peut être lu avec `scanf`, `fgets`, etc.

```
dup2(tube[0], STDIN_FILENO);
```

- Lier l'entrée d'un tube à la sortie standard `stdout`

```
dup2(tube[1], STDOUT_FILENO);
```

  - Tout ce qui sort sur le flot de sortie standard `stdout` entre ans le tube, et on peut écrire dans le tube avec `printf`, `puts`, etc.

Plan

Tubes nommés

Tubes nommés

- Un tube nommé est un tube qui vit dans le système de fichiers
  - Il a donc un nom qui sert de point de rendez-vous aux processus qui désirent communiquer.
- La communication est bidirectionnelle (half-duplex)
- Les tubes nommés synchronisent les lectures et les écritures comme les tubes ordinaires.
  - Une ouverture en écriture bloque jusqu'à ce qu'il y ait un lecteur
  - Une ouverture en lecture bloque jusqu'à ce qu'il y ait un écrivain
- La création d'un tube nommé peut être effectué avec deux appels systèmes : `mknod()` ou `mkfifo()`

Tubes nommés

Création d'un tube nommé avec `mknod()`

- L'appel système `mknod()` crée un fichier spécial, fichier périphérique, ou fichier FIFO
- Cette fonction est déclarée le fichier d'entête `sys/stat.h`

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

  - `pathname` ; le chemin complet du fichier à créer
  - `mode` : les droits d'accès
  - `dev` : le type du fichier

Tubes

Exemple d'un tube nommé créé avec `mknod()`

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
int main() {
   int fd;
   FILE *fp;
   char *nomfich="/tmp/test.txt", chaine[50];
   fd = open(nomfich, O_RDONLY); /* ouverture du tube */
   fp=fdopen(fd, "r"); /* ouverture du flot */
   fscanf(fp, "%s", chaine); /* lecture dans le flot */
   puts(chaine); /* affichage */
   unlink(nomfich); /* fermeture du flot */
   return 0;
}
```


Tubes nommés

Création d'un tube nommé avec `mknod()`

➥ L'appel système `mkfifo()` crée un fichier FIFO

```c
int mkfifo (const char * pathname, mode_t mode);
```

- Renvoie 0 si OK, sinon -1
- `pathname` ; le chemin complet du fichier à créer
- `mode` : les droits d'accès

Tubes

Exemple d'un tube nommé créé avec `mkfifo()`

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
int main() {
   int fd;
   FILE *fp;
   char *nomfich="/tmp/test.txt";
   /* nom du fichier */
   if(mkfifo(nomfich, 0644) != 0) /* création du fichier */
   {
      perror("Problème de création du noeud de tube");
      exit(1);
   }
   fd = open(nomfich, O_WRONLY); /* ouverture en écriture */
   fp=fdopen(fd, "w"); /* ouverture du flot */
   fprintf(fp, "coucou\n"); /* écriture dans le flot */
   unlink(nomfich); /* fermeture du tube */
   return 0;
}
```


Plan

Introduction

Processus coopératifs / indépendants

- Un processus peut être indépendant ou coopératif
  - Un processus indépendant n'est pas affecté par l'exécution d'aucun processus
  - Un processus coopératif peut être affecté par l'exécution d'autres processus
- La communication inter-processus (inter process communication "IPC") est un mécanisme qui permet aux processus de communiquer entre eux et de synchroniser leurs actions
  - Une méthode de coopération.
- Avantages de l'IPC
  - Partage des informations
  - Partage des ressources
  - Augmentation la vitesse de calcul
  - Synchronisation entre les processus

Introduction

Communication entre les processus

- Les IPC couvrent trois mécanismes de communication
  - Les files de messages
  - La mémoire partagée (shared memory)
  - Les sémaphores.
- Il existe deux API
  - IPC système V : compliquée car n'est pas fondée sur la notion des descripteurs de fichiers
  - IPC Posix

Introduction

Construction d'une clé

- `ftok()` permet de construire une clé unique à partir d'un chemin d'accès et d'un caractère indiquant un *projet*
- Elle est déclarée dans le fichier d'entête `<sys/ipc.h>`

```
1 key_t ftok (char *pathname, int project)
```

  - Renvoie une clé unique si OK, sinon -1
  - `pathname` : le chemin absolu du fichier spécial en paramètre jusqu'au point avec une clé spéciale
  - `project` : un caractère qui sera convertit automatiquement au type entier

Exemple illustrant l'utilisation de `ftok()`

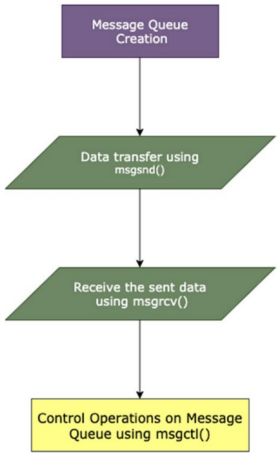
On dispose d'un logiciel commun appelé `/opt/logiciel/OuiOui`. Ce logiciel utilise deux objets partagés. Ainsi, on peut utiliser les clés `ftok("/opt/logiciel/OuiOui",'A')` et `ftok("/opt/logiciel/OuiOui",'B')`.

Plan

Files de messages

Présentation

- Communication effectuée à l'aide de message
  - Les messages sont lus un à un, en intégralité
  - Contrairement aux tubes, il est possible et pratique d'avoir plusieurs écrivains / lecteurs
- Les messages ont des priorités : ils sont livrés par ordre de priorité
- Un processus peut demander à être prévenu de l'arrivée d'un nouveau message



Files de messages

Création/connexion à une file de message

- `msgget()` permet de créer une nouvelle file de messages, ou bien de connecter à une existante

```
1 int msgget(key_t key, int msgflg);
```

  - Renvoie l'identifiant de la file si OK, sinon un code d'erreur
  - `key` : la clé identifiant la file de messages
  - `msgflg` : spécifier les permissions d'accès si la file n'existe pas

Files de messages

Structure d'un tampon de message

- Pour envoyer ou lire un message, on utilise un tampon ayant la structure suivante :

```
1 struct messageBuffer {
2    long messageType;
3    char messageText[SIZE];
4 };
```

  - `messageType` : précise le type de message
  - `messageText` : les données du message
- En cas de lecture, `messageType` précise quel message à lire
  - Si la valeur de `messageType` est 0, le premier message de la file d'attente est lu
  - Si la valeur de `messageType` est supérieure à 0, on lit le premier message dans la file d'attente de type `messageType`
  - Si la valeur de `messageType` est inférieure à 0, le premier message de la file d'attente avec le type le plus bas inférieur ou égal à la valeur absolue du type de message est lu.

Files de messages

Envoie d'un message

- La primitive `msgsnd()` est utilisée pour envoyer un message à partir de la file

```
1 int msgsnd(int msqid,const void *msgp,size_t msgsz,int msgflg);
```

  - Renvoie le nombre d'octets envoyés si OK, sinon -1 et met le code d'erreur dans `errno`
  - `msqid` : l'identifiant de la file
  - `msgp` : un tampon contenant le message à envoyer
  - `msgsz` : taille de message à envoyer
  - `msgflg` : 0 (envoie bloquant) ou une conjonction des options suivantes :
    `IPC_NOWAIT` : pas d'attente
    `MSG_EXCEPT` : utilisée avec un type de message > 0 pour lire un message de différent type

Files de messages

Lecture d'un message

- La primitive `msgrcv()` est utilisée pour lire un message à partir de la file

```
ssize_t msgrcv(int msqid,void *msgp,size_t msgsz,long msgtyp,int msgflg);
```

  - Renvoie le nombre d'octets lus si OK, sinon -1 et met le code d'erreur dans `errno`
  - `msqid` : l'identifiant de la file
  - `msgp` : un tampon pour stocker le message lu
  - `msgsz` : taille de message reçu
  - `msgtyp` : le type de message à lire
  - `msgflg` : 0 (réception bloquante) ou une conjonction des options suivantes :
    `IPC_NOWAIT` : pas d'attente
    `MSG_EXCEPT` : utilisée avec un type de message > 0 pour lire un message de différent type

Files de messages

Contrôler la file de messages

- Contrôler une file de messages

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf)
```

  - Renvoie 0 si OK, sinon -1
  - `msqid` est l'identifiant de la file
  - `cmd` spécifie l'opération à réaliser sur la file de messages
    `IPC_SET` définit l'ID utilisateur et l'ID de groupe du propriétaire et fixe les autorisations
    `IPC_INFO` renvoie des informations sur la file de messages
    `IPC_RMID` supprime immédiatement la file de messages du noyau. `IPC_STAT` fournit des informations sur le tampon `msqid_ds`
  - `buf` est un pointeur sur la structure de la file

Files de messages

Exemple d'envoi d'un message (1/2)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/msg.h>

#define PERMISSIONS 0777

void sendMessage();

int msqid; //Must be global

int main() {
   key_t key;
   system("touch␣messagequeue.txt");
   if ((key = ftok("messagequeue.txt", 'B')) == -1) {
      perror("ftok");
      exit(1);
   }
   if ((msqid = msgget(key, PERMISSIONS | IPC_CREAT)) == -1) {
      perror("msgget");
      exit(1);
   }
   printf("Message␣Queue␣is␣ready␣to␣send␣messages.\n");
   sendMessage(); // Start sending messages
   system("rm␣messagequeue.txt");
   return 0;
}
```


Files de messages

Exemple d'envoi d'un message (2/2)

```c
void sendMessage(){
   int len;
   struct messageBuffer object;
   object.messageType = 1; // Setting the message type value to 1.
   while(fgets(object.data, sizeof object.data, stdin) != NULL) {
      len = strlen(object.data);
      if (object.data[len-1] == '\n') object.data[len-1] = '\0';
      if (msgsnd(msqid, &object, len+1, 0) == -1){
         perror("msgsnd");
         exit(1);
      }
      if(strcmp(object.data, "end") == 0) // end ?
      break;
   }
   if (msgctl(msqid, IPC_RMID, NULL) == -1) {
      perror("msgctl");
      exit(1);
   }
   printf("Message␣Queue␣is␣done␣with␣sending␣messages.\n");
}
```

Files de messages

Exemple de réception d'un message (1/2)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/msg.h>
#define PERMISSIONS 0777

void receiveMessages();
int msqid;

int main() {
   key_t key;
   if ((key = ftok("messagequeue.txt", 'B')) == -1) {
      perror("ftok");
      exit(1);
   }
   if ((msqid = msgget(key, PERMISSIONS)) == -1) { // Connect to MQ
      perror("msgget");
      exit(1);
   }
   printf("Message Queue is ready to receive messages.\n");
   receiveMessages();
   printf("Message Queue is done with receiving messages.\n");
   return 0;
}
```


Files de messages

Exemple de réception d'un message (1/2)

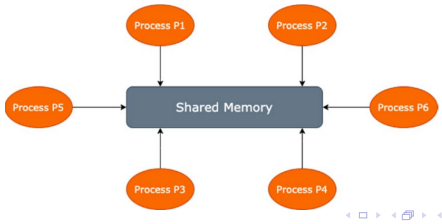
```c
struct messageBuffer {
   long mtype;
   char data[1024];
};
void receiveMessages(){
   struct messageBuffer object;
   while(1) {
      if (msgrcv(msqid, &object, sizeof(object.data), 0, 0) == -1) {
         perror("msgrcv");
         exit(1);
      }
      printf("received: \"%s\"\n", object.data);
      if (strcmp(object.data,"end") == 0) // end received ?
      break;
   }
}
```


Plan

Mémoire partagée

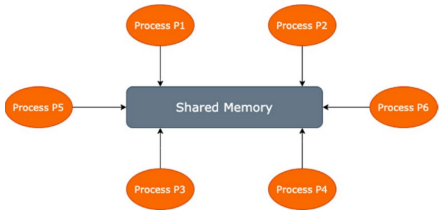
Principe

- Partage d'une page de mémoire entre deux processus
  - Il suffit alors d'écrire des données dans cette zone pour les partager
- Méthode de communication plus efficace que les autres techniques IPC : les tubes, files de messages, etc.
  - Pas de recopie des données d'un espace d'adressage à un autre
  - Allocation de l'espace mémoire pour les données partagées s'effectue une seule fois
- Mais, il faut synchroniser les accès à cette zone mémoire
  - Utiliser les mutex, semaphores, read/write lock



Mémoire partagée

Mémoire partagée vs autres techniques IPC



Mémoire partagée

Différents types de partage mémoire

- Partage anonyme
  - Entre un processus et ses fils uniquement
- Partage lié à un fichier
  - Partage entre plusieurs processus
  - Contenu stocké sur le disque dans un fichier
  - Plus sûr
- Partage nommé (POSIX)
  - Partage entre plusieurs processus
  - Pas de stockage dans un fichier
  - Plus rapide

Mémoire partagée

Partage anonyme d'un segment mémoire

- La création d'un segment s'effectue en trois étapes
  1. Générer une clé à partir d'un nom de fichier avec `ftok()` le segment de mémoire pour les autres processus souhaitant s'y rattacher
  2. Recupérer l'identifiant de la zone mémoire partagée avec `shmget()`
  3. Attacher le segment mémoire partagée a l'espace mémoire du processus `shmat()`
- Modification
  - Le contrôle est réalisé avec `shmctl()`
- Destruction
  - Détacher le segment mémoire de l'espace mémoire du processus avec `shmdt()`
  - Détruire le segment mémoire si plus personne ne l'utilise avec `shmctl()`

Mémoire partagée

Création d'un segment mémoire partagé

- Pour créer un segment de mémoire, on attache à ce segment un certain nombre d'informations
- La création d'un segment mémoire partagé s'effectue avec la primitive `shmget()` déclarée dans le fichier d'entête `<sys/shm.h>`

```
int shmget( key_t cle , size_t taille , int flag);
```

  - `cle` est un entier (une clé conservée par le système) permettant d'identifier le segment de mémoire pour les autres processus souhaitant s'y rattacher
  - `taille` est le nombre d'octets, multiple de la taille d'une page, du segment de mémoire
  - `flag` est un drapeau de permission, utilisé de la même façon que les drapeaux de mode de permission destinés à la création de fichiers sous UNIX

Mémoire partagée

Attacher un segment mémoire partagé

- Attacher un segment mémoire partagé à l'espace d'adressage du processus appelant en utilisant la primitive `shmat()` déclarée dans le fichier d'entête `<sys/shm.h>`

```
1 void * shmat(key_t cle,const void *shmaddr,int flag)
```

  - `cle` est la clé conservée par le système identifiant le segment de mémoire à rattacher
  - `shmaddr` est le nombre d'octets, multiple de la taille d'une page, du segment de mémoire
  - `flag` est un drapeau de permission, utilisé de la même façon que les drapeaux de mode de permission destinés à la création de fichiers sous UNIX

Mémoire partagée

Modifier un segment mémoire partagé

- Contrôle d'un segment mémoire partagé

```
1 int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

  - Renvoie 0 si OK, -1 sinon avec code d'erreur dans `errno`
  - `shmid` est l'identifiant du segment mémoire à contrôler
  - `cmd` spécifie l'opération de contrôle, par exemple :
    `IPC_RMID` désalloue le segment et détruit les données associées
    `IPC_STAT` collecte des informations sur le segment et les place dans `buf`
    `IPC_SET` change les droits d'accès sur le segment

Mémoire partagée

Détacher un segment mémoire partagé

- Attacher un segment mémoire partagé à l'espace d'adressage du processus appelant en utilisant la primitive `shmat()` déclarée dans le fichier d'entête `<sys/shm.h>`

```
1 int shmdt(const void *shmaddr)
```

  - Renvoie 0 si OK, -1 sinon
  - `shmaddr` spécifie l'adresse de l'espace mémoire à détacher

Mémoire partagée

Exemple de partage d'un segment mémoire

```
1  // Create the key
2  key_t key=ftok("/etc/bash.bashrc" ,1);
3  if ( key == -1 ) {
4     perror("ftok");
5     exit(1);
6  }
7  // Get the shmid
8  id=shmget(key,SIZE,IPC_CREATE|0644);
9  if (id==-1) {
10    perror("shmget");
11    exit(1)
12 }
13 // Attac
14 A=shmat(id,NULL,0);
15 if (A==(void *)-1) {
16    perror("shmat");
17    exit(1);
18 }
19 // Detach/destroy
20 shmdt(A) ;
21 shctl(id,IPC_RMID,NULL);
```

Plan

Plan

API Posix pour les files de messages

Opérations de gestion des files de messages

- Les fonctions Posix pour la gestion des files de messages sont déclarées dans `<mqueue.h>`
- Créer ou ouvrir une file de messages

```
mqd_t mq_open(const char *name, int oflag, mode_t mode,struct mq_attr *attr);
```

  - Renvoie un descripteur de la file si OK, sinon -1 et `errno` contient le code de l'erreur
  - `name` est une chaine de caractère choisie arbitrairement qui doit commencer par `/` et ne doit pas comporter d'autres slashs
  - `oflag` est identique au paramètre `flag` pour la fonction `open()`
  - `mode` spécifie les droits d'accès
  - `attr` spécifie certains attribut de la file de message, `NULL` pour utiliser les valeurs par défaut
- Fermeture d'une file de messages sans la détruire. Sa destruction s'effectuera à la fin du processus

```
int mq_close(mqd_t mqd);
```

  - Renvoie 0 si OK, sinon -1 et `errno` indique le code de l'erreur
  - `mqd` est le descripteur de la file à fermer

API Posix pour les files de messages

Opérations de gestion des files de messages

- Même après fermeture, la file n'est pas détruite. Toutefois, la file reste à la disposition d'autres processsus jusqu'au dernier `mq_close()` mais sans pouvoir effectuer un nouveau `open()`

```
int mq_unlink(mqd_t const char *name);
```

  - Renvoie 0 si OK, sinon -1 et `errno` indique le code de l'erreur
  - `name` est la chaine arbitraire choisie lors de la création de la file de messages
- Envoyer un message

```
int mq_send(mqd_t mqd, const char * msg,size_t len, unsigned int msg_prio);
```

  - Renvoie 0 si OK, sinon -1 et `errno` indique le code de l'erreur
  - `mqd` est le descripteur de la file
  - `msg` est le message à envoyer
  - `len` est la taille du message à envoyer
  - `msg_prio` est la priorité du message qui est une valeur comprise entre $[0, sysconf(\_SC\_MQ\_PRIO\_MAX) - 1]$. Les messages sont délivrés par ordre décroissant selon la priorité.

API Posix pour les files de messages

Exemple d'envoi d'un message

```c
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int priorite;
   mqd_t mqd;
   if (argc!=4) {
      printf("Syntaxe : %s fichier priorite message\n",argv[0]);
      return -1;
   }
   mqd = mq_open(argv[1], O_WRONLY|O_CREAT,0644,NULL);
   if (mqd == (mqd_t) -1) {
      perror("mq_open failed!");
      return -1;
   }
   if (sscanf(argv[2],"%d",&priorite)!=1) {
      perror("Priorite invalide!");
      return -1;
   }
   if (mq_send(mqd, argv[3], strlen(argv[3]), priorite)!=0) {
      perror("mq_send!");
   }
   return 0;
}
```


API Posix pour les files de messages

Opérations de gestion des files de messages

➥ Récupérer les attributs d'une file de messages

```c
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

- ☞ Renvoie 0 si OK, sinon -1 et `errno` indique le code de l'erreur
- ☞ `mqd` est le descripteur de la file
- ☞ `attr` est un pointeur sur la structure des attributs

Champ	Type	Signification
`mq_flags`	`long`	Mode d'accès (0 : normal, O_NONBLOCK : non bloquant).
`mq_maxmsg`	`long`	Nombre maximal de messages dans la file.
`mq_msgsize`	`long`	Taille maximale d'un message.
`mq_curmsgs`	`long`	Nombre de messages actuellement présents.

➥ Modifier les attributs d'une file de messages

```c
int mq_setattr(mqd_t mqd, const struct mq_attr *newattr,struct mq_attr *oldattr);
```

- ☞ Renvoie 0 si OK, sinon -1 et `errno` indique le code de l'erreur
- ☞ `mqd` est le descripteur de la file
- ☞ `newattr` est un pointeur sur la structure des nouveaux attributs
- ☞ `oldattr` est un pointeur sur la structure des nouveaux attributs

API Posix pour les files de messages

Opérations de gestion des files de messages

➥ Recevoir un message

```c
ssize_t mq_receive(mqd_t mqdes,char *msg,size_t len, unsigned int *msg_prio);
```

- ☞ Renvoie 0 si OK, sinon -1 et `errno` indique le code de l'erreur
- ☞ `mqd` est le descripteur de la file
- ☞ `msg` est le tampon susceptible de sauvegarder le message reçu. Sa taille doit être suffisante pour contenir le message à recevoir.
- ☞ `len` est la taille maximale du message à recevoir
- ☞ `msg_prio` indique la priorité du message reçu

API Posix pour les files de messages

Exemple de réception d'un message

```c
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
   unsigned int priorite;
   char * buffer = NULL;
   struct mq_attr attr;
   mqd_t mqd;
   if (argc!=2) {
      printf("Syntaxe : %s fichier\n",argv[0]); return -1;
   }
   mqd = mq_open(argv[1], O_RDONLY);
   if (mqd == (mqd_t) -1) {
      perror("mq_open failed!"); return -1;
   }
   if (mq_getattr(mqd,&attr)!=0) {
      perror("mq_getattr"); return -1;
   }
   if ((buffer=malloc(attr.mq_msgsize))==NULL) {
      perror("malloc"); return -1;
   }
   if (mq_receive(mqd,buffer,attr.mq_msgsize,&priorite)<0) {
      perror("mq_receive"); return -1;
   }
   printf("Message %s avec priorite %d\n",buffer,priorite);
   return 0;
}
```

Plan

API Posix pour la mémoire partagée

Opérations de gestion de la mémoire partagée

- La mémoire partagée avec POSIX est créée sans définir un fichier sur le disque de mapping
  - Plus rapide
- Généralement, on utilise un objet (fichier) du système de fichiers `tmpfs`
  - C'est un système de fichiers virtuel créé dans la mémoire
  - `tmpfs` est monté dans `/dev/shm`
- Étapes de création d'un segment mémoire partagée avec l'API Posix
  1. Ouvrir un objet (segment de mémoire partagée) avec `shm_open()` avec un nom donné
  2. Dimensionner le segment mémoire avec `ftruncate()`
  3. Mapper l'objet ouvert dans l'espace mémoire virtuel du process avec la primitive `mmap()`

API Posix pour la mémoire partagée

Opérations de gestion de la mémoire partagée

- La mémoire partagée avec POSIX est créée sans définir un fichier de mapping
- Ouvrir un segment mémoire

```
int shm_open(const char *name, int oflag, mode_t mode);
```

  - Renvoie un descripteur du segment mémoire si OK, sinon -1 et `errno` contient le code de l'erreur
  - `name` est une chaine de caractère choisie arbitrairement qui doit commencer par `/` et ne doit pas comporter d'autres slashs
  - `oflag` est identique au paramètre `flag` pour la fonction `open()`
  - `mode` spécifie les droits d'accès
- Détruire un segment mémoire
  - Le segment ne sera pas accessible par son nom
  - Mais, les processus qui l'ont déjà mappé peuvent continuer à l'utiliser

```
int shm_unlink(const char *name);
```

  - Renvoie 0 si OK, sinon -1 et `errno` indique le code de l'erreur
  - `name` est le nom du segment à détruire

API Posix pour la mémoire partagée

Opérations de gestion de la mémoire partagée

- Un segment mémoire doit être dimensionné avant son utilisation
  - Si (ancienne taille > nouvelle taille), alors les données en excédent sont perdues

```
int ftruncate(int fd, off_t length);
```

  - Renvoie 0 si OK, sinon -1 et `errno` contient le code de l'erreur
  - `fd` est le descripteur du segment mémoire à rédimensionner
  - `length` est la taille à allouer au segment mémoire

API Posix pour la mémoire partagée

Exemple d'utilisation d'un segment de mémoire partagée

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
int main(int argc,char *argv[]) {
   int fd;
   if (argc!=2) {
      perror("Requires memory segment name as argument!"); exit(EXIT_FAILURE);
   }
   if ((fd=shm_open(argv[1],O_RDWR|O_CREAT,0600))==-1) {
      perror(argv[1]); exit(EXIT_FAILURE);
   }
   if (ftruncate(fd,sizeof(int))!=0) {
      perror("ftruncate"); exit(EXIT_FAILURE);
   }
   int *compteur=mmap(NULL,sizeof(int),PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
   if (compteur==MAP_FAILED) {
      perror("mmap"); exit(EXIT_FAILURE);
   }
   while (1) {
      (*compteur)++;sleep(1);
      printf("Compteur = %d\n",*compteur);
   }
   return EXIT_SUCCESS;
}
```


Plan

Gestion de la mémoire

Lien entre mémoire virtuelle et mémoire réelle

- Traduction mémoire virtuelle
  - Mémoire réelle déterminée par le CPU (MMU/TLB)
- Mécanisme de pages
  - La mémoire est gérée par blocs de 2ko de mémoire (4Ko ou 2Mo sur x86), des pages
  - Chaque page virtuelle correspond à une page réelle
- La traduction s'appuie sur un mapping des pages (traduction par dictionnaire)
- L'adresse virtuelle est composée de différents champs permettant de construire l'adresse réelle

Plan

Gestion de la mémoire

Projeter un fichier sur la mémoire (1/2)

- Travailler sur la mémoire est plus facile que sur un fichier
  - Projeter une image d'un fichier sur une zone mémoire

```
void * mmap(void * addr, size_t len,int prot, int flags,int fd,off_t offset);
```

  - Renvoie un pointeur sur la zone de mémoire allouée pour la projection si OK, sinon `MAP_FAILED`
  - `addr` est adresse ou placer le contenu du fichier. `NULL` = l'OS choisit (mieux)
  - `len` est la longueur en octets de la zone mémoire (multiple de taille de page). ´ Le fichier doit faire au moins cette taille ou plus.
  - `prot` spécifie les autorisations d'accès en utilisant des combinaisons avec OU : `PROT_NONE` (pas d'acces), `PROT_READ` (lecture), `PROT_WRITE` (écriture), `PROT_EXEC` (execution)
  - `flags` spécifie les attributs de la projection :
    `MAP_PRIVATE` : pages privées, la projection n'est pas destinée à être réécrite sur le disque
    `MAP_SHARED` : pages partagees avec autres processus
    `MAP_ANONYMOUS` : pages pas associées à un fichier (`fd`, `offset` ignorées)

Gestion de la mémoire

Projeter un fichier sur la mémoire

- Travailler sur la mémoire est plus facile que sur un fichier
  - Projeter une image d'un fichier sur une zone mémoire

```
void * mmap(void * addr, size_t len,int prot, int flags,int fd,off_t offset);
```

  - `fd` est le descripteur de fichier à projeter sur la mémoire
  - `offset` est la position dans le fichier à partir de laquelle le fichier sera projeté sur la mémoire

Gestion de la mémoire

Libérer une projection

- Libérer une zone mémoire projetée d'un fichier

```
int munmap (void * zone, size_t longueur);
```

  - Retourne 0 si OK, sinon -1 et met dans `errno` le code d'erreur
  - `zone` est la zone mémoire allouée avec `mmap()`
  - `longueur` désigne la taille de bloc mémoire à libérer

Gestion de la mémoire

Exemple de projection d'un fichier sur la mémoire

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

#define SIZE 50

void main() {
   int f = open("monfichier.txt", O_RDWR);
   if (f == -1) {
      printf("Erreur_d\'ouverture");
      exit(1);
   }
   // Allocation des pages memoire
   char *A = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, f, 0);
   if (A == MAP_FAILED) {
      perror("mmap");
      exit(1);
   }
   A[10] = 'a';
   // Destruction
   munmap(A, SIZE);
}
```

Plan

Gestion de la mémoire

Limiter l'accès à une zone mémoire

➥ Limiter les possibilités d'accès à certaines pages mémoire

```
1 int mprotect (const void * debut_zone, size_t longueur, int prot);
```

☞ Retourne 0 si OK, sinon -1 et met dans `errno` le code d'erreur
☞ `prot` spécifie les autorisations d'accès en utilisant des combinaisons avec OU :
`PROT_NONE` : aucun accès
`PROT_READ` : accès en lecture
`PROT_WRITE` : accès en écriture
`PROT_EXEC` : accès en exécution

Gestion de la mémoire

Exemple de protection d'une zone mémoire (1/2)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/mman.h>
5  #define TAILLE_CHAINE 128
6  void *mon_malloc(size_t taille) {
7     void *retour = mmap(NULL, taille, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 
8     if (retour == MAP_FAILED) return NULL;
9     return retour;
10 }
11
12 int main(){
13    char *chaine;
14    chaine= mon_malloc(TAILLE_CHAINE);
15    if (chaine==NULL) {
16       perror("mmap");
17       exit(EXIT_FAILURE);
18    }
19    printf("Ecriture...\n");
20    strcpy(chaine,"OK");
21    printf("Ecriture␣OK!\n");
22    printf("Lecture...\n");
23    printf("Chaine␣:␣%s\n",chaine);
24    printf("Lecture␣OK!\n");
```


Gestion de la mémoire

Exemple de protection d'une zone mémoire (2/2)

```
1     printf("Interdiction␣d'ecriture...\n");
2     if (mprotect(chaine,TAILLE_CHAINE,PROT_READ)<0) {
3        perror("mprotect");
4        exit(EXIT_FAILURE);
5     }
6     printf("Lecture...\n");
7     printf("Chaine␣:␣%s\n",chaine);
8     printf("Lecture␣OK!\n");
9     printf("Ecriture...\n");
10    strcpy(chaine,"OK"); // Ooops
11    printf("Ecriture␣OK!\n");
12
13    return EXIT_SUCCESS;
14 }
```

MERCI POUR VOTRE ATTENTION

Questions ?