CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Samedi 16 juin 2007
2 heures
08h00 – 10h00

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103

EXAMEN
Année 2006 – 2007, second semestre
Coefficient : 3/4

Documents autorisés : NON

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Centres Régionaux du PTI Nord Est

Question n°1 : répertoire téléphonique *(7 pts)*

La société « *Line.PhoNet* » souhaite mettre sur le marché une nouvelle gamme de téléphones mobiles « *Line.PhoNux* », basée sur le système d'exploitation Linux. Il est décidé que cette gamme de téléphones sera capable de proposer les outils de communication les plus modernes : téléphone GSM, SMS, MMS, mais aussi e-mail et accès au web.

Tous ces services doivent s'appuyer sur un carnet d'adresses unique (répertoire téléphonique), qui contiendra, pour chaque entrée :

- Nom (63 caractères max.),
- Prénoms (63 caractères max.),
- Numéro de téléphone GSM (15 caractères max.),
- Adresse e-mail (127 caractères max.),
- La page de son site web personnel ou blog (255 caractères max.).

Ce carnet d'adresses pourra contenir un maximum de 256 entrées, et sera stocké dans un fichier sur un disque de mémoire flash (formaté avec un système de fichier classique pour Linux, de type *ext3* ; aussi, les accès aux fichiers se font comme s'il s'agissait d'un disque dur).

Tous les services proposés à l'utilisateur (le logiciel de gestion du carnet d'adresse, le service d'envoi/réception d'appels téléphoniques GSM, le service d'envoi/réception des messages de type SMS/MMS/messagerie électronique, le navigateur web) sont des processus.

Le cahier des charges imposé par « *Line.PhoNet* » est le suivant :

- lorsque c'est possible, à tout moment, dès que le mobile reçoit un flux de données entrant (appel téléphonique, SMS, MMS, e-mail), le mobile doit être capable d'afficher le « Prénoms+Nom » de l'émetteur ;
- pour tout message archivé (appel téléphonique manqué, SMS, MMS, e-mail), l'utilisateur doit posséder une fonction « *Ajouter au carnet d'adresses* » ;
- lorsqu'il utilise un service (appel téléphonique, envoi de SMS/MMS/e-mail, navigateur web), l'utilisateur a la possibilité de chercher le destinataire dans le carnet d'adresse ;
- évidemment, l'utilisateur peut, lorsqu'il est dans le logiciel « gestion du carnet d'adresses », ajouter manuellement un correspondant, lire une entrée, la modifier, ou la supprimer.

Question 1.a) ***[1 point]*** Résumer quelle est la problématique d'accès au(x) fichier(s) qui stockent le carnet d'adresses ?

Question 1.b) ***[2,5 points]*** Quelles solutions techniques pourriez-vous proposer pour répondre à cette problématique ? Argumentez les avantages-inconvénients de chaque solution.

Question 1.c) ***[3,5 points]*** Parmi toutes ces solutions, la société « *Line.PhoNet* » fait le choix d'une gestion non centralisée du problème, en utilisant les sémaphores (fonctions « `Init(semaphore, val0)` », « `P(semaphore)` », et « `V(semaphore)` »). Proposez le squelette (en C ou en pseudo code) des fonctions :

- « `quel_est_le_nom_d_expediteur()` » et « `ajouter_au_carnet_adresses()` » du processus de gestion des SMS, dont le rôle est respectivement de connaître le « Prénoms+Nom » correspondant au numéro d'un SMS reçu, et d'ajouter une entrée (Prénoms+Nom+Numéro) dans le carnet d'adresses ;
- « `quel_est_le_numero()` » et « `effacer_une_entree()` » du processus de gestion du carnet d'adresses, dont le rôle est respectivement de connaître le numéro de GSM correspondant à une personne (connaissant ses Prénoms+ Nom), et effacer une entrée du carnet d'adresses correspondant à une personne (connaissant ses Prénoms+ Nom).

Les squelettes de ces fonctions ne doivent contenir que le code lié aux problématiques de programmation système/résolution des contraintes liées à la multi-programmation décrites dans la réponse à la question *1.a)*. Le candidat n'a pas à programmer la gestion du stockage et la recherche des données dans les fichiers.

Question n°2 : mémoire, segmentation, pagination *(6 pts)*

Question 2.a) ***[1 point]*** Quel est l'intérêt d'utiliser un espace d'adressage virtuel et des mécanismes de traduction d'adresses dans les systèmes d'exploitations multiprogrammés ?

Question 2.b) ***[2 points]*** Quels sont les points communs et les différences fondamentales entre les mécanismes de pagination et de segmentation, utilisés dans la gestion de la mémoire vive ?

Question 2.c) ***[3 points]*** Rappelez les principes de l'algorithme de remplacement des pages « LRU ». Sachant qu'un processus veut accéder aux pages ci-dessous sur un ordinateur équipé de 3 cases (numérotées de 0 à 2) et de 8 pages (numérotées de 0 à 7), faites un schéma qui indique, à chaque accès à une page, l'état de la table des cases, s'il y a un défaut de page, et dans ce cas, quelle est la page victime qui sera choisie pour être stockée sur disque dur :

2, 1, 0, 7, 1, 6, 1, 5, 7, 6, 2, 6, 7, 0, 7, 1, 0, 2, 1, 7.

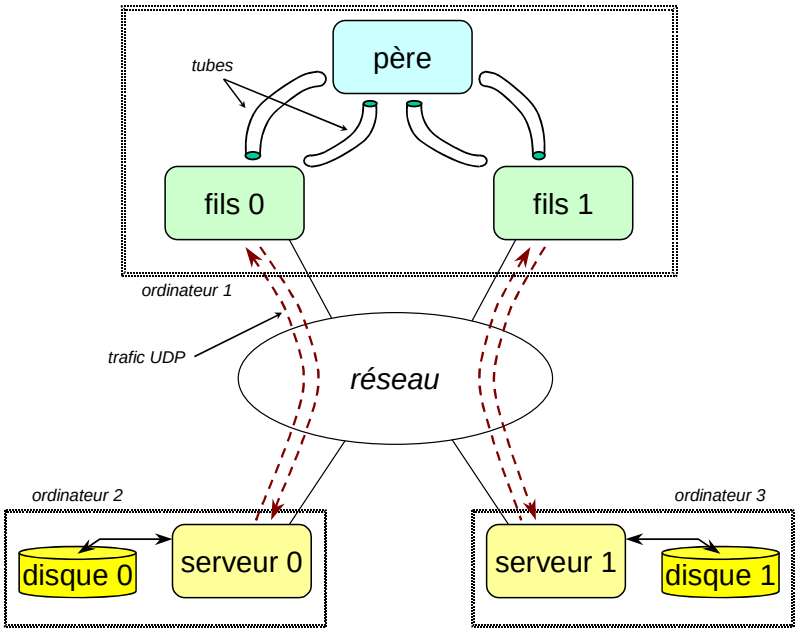
Donner l'état de la table des pages à la fin de ces accès.

Question n°3 : stockage RAID1 réseau *(7 pts)*

Une équipe de chercheurs souhaite effectuer une expérience exothermique. Afin de mesurer la température dégagée, un PC tournant sous Linux, équipé de capteurs de température, est placé à proximité. La température relevée par les capteurs est enregistrée dans un fichier.

Les premières simulations numériques indiquent qu'il est possible que l'ordinateur subisse des dommages irréparables dans certains cas. Aussi, les chercheurs décident-ils de ne pas stocker les données dans le PC, mais sur les disques de machines situées dans d'autres locaux, accessibles via un réseau IP.

Afin de programmer un gestionnaire de système de fichiers, les informaticiens proposent la solution suivante : lorsqu'il a besoin de lire ou d'écrire un bloc sur le disque dur, le processus du PC capteur envoie les ordres de lecture (ou d'écriture) à deux processus fils à travers un tube anonyme. Chacun des deux processus fils envoie une requête UDP (sur le port 1200) de lecture (ou d'écriture) à travers le réseau IP à un serveur. En réponse à cette requête, le serveur envoie le contenu du bloc lu (ou un accusé de réception de bonne écriture). Chacun des deux fils redonne cette information (ou une information « *timeout* » si la réponse à la requête n'est pas obtenue après 3 secondes) au processus père via un tube anonyme. L'ensemble de la solution peut être schématisé ainsi :



Question 3.a) ***[1 point]*** Dans le schéma ci-dessus, pourquoi y a-t-il 4 tubes anonymes connectés au processus « parent » ?

Question 3.b) ***[2 points]*** Les *datagrammes* du trafic client/serveur auront toujours la même taille. Une requête contiendra toujours les données suivantes : un entier de valeur 0 pour une demande de lecture, ou de valeur 1 pour une demande d'écriture. Ce nombre est suivi d'un entier long indiquant le numéro de bloc. Les 512 octets suivants seront le bloc à écrire (en cas de demande d'écriture), ou indéterminés sinon. En réponse, le premier entier contiendra 2 si l'opération s'est bien déroulée, et 4 en cas d'erreur de lecture ou d'écriture. L'entier long suivant rappellera le bloc concerné. Les 512 octets suivants seront soit le blocs lu (en cas d'ordre de lecture), soit indéterminés. Pour implémenter ce protocole, les programmes du projet utiliseront la structure « `struct msg_reseau` » définie dans le fichier « `msg_reseau.h` » dont voici le contenu :

```
#ifndef     __MSG_RESEAU_H_
#define     __MSG_RESEAU_H_

#include    <stdio.h>
#include    <stdlib.h>
#include    <unistd.h>
#include    <string.h>

/* Directives includes pour les fonctions liées au réseau : */
#include    <netdb.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <arpa/inet.h>

/* directives système pour le traitement du multitâches : */
#include    <sys/wait.h>
#include    <signal.h>
#include    <sys/time.h>

/* Numéro de port sur lequel écoute le serveur : */
#define     PORT_SERVEUR      1200

/* Taille d'un bloc sur le disque */
#define     TAILLE_BLOC 512

/* structure permettant la communication client/serveur */
struct msg_reseau
{
      int   operation;
      long  num_bloc;
      char  bloc[TAILLE_BLOC];
};
/* Les codes de message */
#define     OP_LECTURE        0
#define     OP_ECRITURE       1
#define     OP_OK             2
#define     OP_ERREUR         4
#define     OP_TIMEOUT        8

#endif      /* __MSG_RESEAU_H_ */
```

A supposer qu'il existe sur les ordinateurs 2 et 3 deux primitives « `int lit_bloc(long num_bloc, void *bloc);` » et « `int ecrit_bloc(long num_bloc, void *bloc);` » qui permettent respectivement de lire et d'écrire les 512 octets pointés par `bloc` dans le bloc `num_bloc` (fonctions qui retournent 0 si tout va bien, -1 sinon), compléter le programme « `serveur.c` » suivant, qui tourne en tâche de fond sur les ordinateurs 2 et 3 :

```
#include "msg_reseau.h"

extern int  lit_bloc(long num_bloc, void *bloc);
extern int  ecrit_bloc(long num_bloc, void *bloc);
```

```
int   main()
{
      int                     mon_socket;
      struct sockaddr_in      sockadrs_moi;
      struct sockaddr_in      sockadrs_client;
      size_t                  lg_sockadrs_client=sizeof(struct sockaddr_in);
      ssize_t                 taille_msg_recu;
      short int               num_port=PORT_SERVEUR;
      struct msg_reseau       question;
      struct msg_reseau       reponse;
      int                     err;

      mon_socket=socket(AF_INET, SOCK_DGRAM, 0);
      if (mon_socket < 0)
      {
            fprintf(stderr,"Erreur: je ne peux créer le socket\n");
            exit(-1);
      }
      bzero(&sockadrs_moi, sizeof(struct sockaddr_in));
      bzero(&sockadrs_client, sizeof(struct sockaddr_in));
      sockadrs_moi.sin_family = PF_INET;
      sockadrs_moi.sin_addr.s_addr = htonl(INADDR_ANY);
      sockadrs_moi.sin_port = htons(num_port);
      if (bind(mon_socket, (struct sockaddr *) &sockadrs_moi, \
                                        sizeof(struct sockaddr_in)) < 0)
      {
            fprintf(stderr,"Erreur : bind\n");
            exit(-1);
      }
      while (1)
      {
            /*************************************************************/
            /* Ecrire le code qui doit venir dans cette boucle infinie */
            /*************************************************************/
      }
}                                   /* Fin fichier "serveur.c" */
```

Question 3.c) ***[4 points]*** Les dialogues entre le processus père et les processus fils au travers des tubes (ordinateur 1) reprendront le même principe que le protocole décrit à la question *3.b)*, excepté que si un fils n'a pas de réponse à sa requête UDP au bout de 3 secondes, le premier nombre retourné au père sera le chiffre 8 (qui indique un *timeout*).

Considérant le squelette du programme « `client.c` » ci-dessous, qui tourne sur l'ordinateur 1 :

- quand est exécuté le code de la fonction « `fin_demandee()` », et pourquoi la boucle « `while (wait(NULL) > 0) ;` » de la fonction « `termine_fils()` » est elle importante ?
- combien vaut la variable « `nb_fils` » pour le père et pour les deux fils ?
- compléter la fonction « `code_du_fils()` ».

```
#include "msg_reseau.h"

int   tubes_pere_vers_fils[2][2];
int   tubes_fils_vers_pere[2][2];
int   pid_fils[2];
int   nb_fils;
char  adrs_serveur[2]; /* adresses des ordinateurs 2 & 3 passées en paramètre */

void  fin_demandee()
{
            close(tubes_fils_vers_pere[nb_fils][1]);
            close(tubes_pere_vers_fils[nb_fils][0]);
            exit(0);
}
```

```
void  code_du_fils()
{
      int                     mon_socket;
      struct hostent          *serv_hostinfo;
      struct sockaddr_in      sockadrs_serv;
      ssize_t                 taille_msg_recu;
      short int               num_port=PORT_SERVEUR;
      char                    *nom_serveur;
      struct msg_reseau       msg;
      struct timeval          t_out;
      fd_set                  read_fs;
      nom_serveur=adrs_serveur[nb_fils];
      mon_socket=socket(AF_INET, SOCK_DGRAM, 0);
      if (mon_socket < 0)
      {
            fprintf(stderr,"Erreur: je ne peux créer le socket\n");
            exit(-1);
      }
      bzero(&sockadrs_serv, sizeof(struct sockaddr_in));
      serv_hostinfo=gethostbyname(nom_serveur);
      if (serv_hostinfo == NULL)
      {
            fprintf(stderr,"Echec de resolution DNS avec %s\n", nom_serveur);
            exit(-1);
      }
      sockadrs_serv.sin_family=serv_hostinfo->h_addrtype;
      memcpy((char *)&sockadrs_serv.sin_addr.s_addr, \
                        serv_hostinfo->h_addr_list[0], serv_hostinfo->h_length);
      sockadrs_serv.sin_port=htons(num_port);
      t_out.tv_sec=3;  /* le timeout est armé à 3 sec. */
      t_out.tv_usec=0;
      while(1)
      {
            /*************************************************************/
            /* Ecrire le code qui doit venir dans cette boucle infinie */
            /*************************************************************/
      }
}

void  initialise_fils()
{
      pid_t le_pid;
      int         i;
      for (i=0;i<2;i++)
      {
            if ((pipe(tubes_pere_vers_fils[i]) == -1) || \
                        (pipe(tubes_fils_vers_pere[i]) == -1))
            {
                  fprintf(stderr, "Erreur avec pipe()\n");
                  exit(-1);
            }
      }
      for (nb_fils=0;nb_fils<2;nb_fils++)
      {
            le_pid = fork();
            switch (le_pid)
            {
                  case -1     :
                        fprintf(stderr, "Erreur avec fork()\n");
                        exit(-1);
                        break;
                  case 0      :
                        close(tubes_fils_vers_pere[nb_fils][0]);
                        close(tubes_pere_vers_fils[nb_fils][1]);
```

```
                        signal(SIGUSR1, fin_demandee);
                        code_du_fils();
                        break;
                  default :
                        pid_fils[nb_fils]=le_pid;
                        close(tubes_pere_vers_fils[nb_fils][0]);
                        close(tubes_fils_vers_pere[nb_fils][1]);
            }
      }
}

int         ecrit_bloc(long num_bloc, void *donnees)
{
      int               reponse=0;
      int               i;
      struct msg_reseau msg[2];
      msg[0].operation=OP_ECRITURE;
      msg[0].num_bloc=msg[1].num_bloc=num_bloc;
      memcpy(msg[0].bloc,donnees,TAILLE_BLOC);
      for (i = 0 ; i < 2 ; i++)
      {
            if (write(tubes_pere_vers_fils[i][1], &(msg[0]), \
                                    sizeof(struct msg_reseau)) < 0)
            {
                  fprintf(stderr,"Erreur écriture dans le pipe\n");
                  exit(-1);
            }
      }
      for (i = 0 ; i < 2 ; i++)
      {
            if (read(tubes_fils_vers_pere[i][0], &(msg[i]), \
                                      sizeof(struct msg_reseau)) < 0)
            {
                  fprintf(stderr,"Erreur lors de la lecture dans le pipe\n");
                  exit(-1);
            }
      }
      if ((msg[0].operation == OP_TIMEOUT) || (msg[1].operation == OP_TIMEOUT))
      {
            fprintf(stderr,"Pas de réponse répondu après 3 s.\n");
            reponse=-1;
      }
      if ((msg[0].operation == OP_ERREUR) || (msg[1].operation == OP_ERREUR))
      {
            fprintf(stderr,"Erreur : erreur de lecture sur un des serveur.\n");
            reponse=-1;
      }
      return(reponse);
}

int         lit_bloc(long num_bloc, void *donnees)
{
      int               reponse=0;
      int               i;
      struct msg_reseau msg[2];
      msg[0].operation=OP_LECTURE;
      msg[0].num_bloc=msg[1].num_bloc=num_bloc;
      for (i = 0 ; i < 2 ; i++)
      {
            if (write(tubes_pere_vers_fils[i][1], &(msg[0]), \
                                       sizeof(struct msg_reseau)) < 0)
            {
                  fprintf(stderr,"Erreur à l'écriture dans le pipe\n");
                  exit(-1);
```

```
            }
      }
      for (i = 0 ; i < 2 ; i++)
      {
            if (read(tubes_fils_vers_pere[i][0], &(msg[i]), \
                                          sizeof(struct msg_reseau)) < 0)
            {
                  fprintf(stderr,"Erreur lors de la lecture dans le pipe\n");
                  exit(-1);
            }
      }
      if ((msg[0].operation == OP_TIMEOUT) || (msg[1].operation == OP_TIMEOUT))
      {
            fprintf(stderr,"Pas répondu après 3 sec.\n");
            reponse=-1;
      }
      if ((msg[0].operation == OP_ERREUR) || (msg[1].operation == OP_ERREUR))
      {
            fprintf(stderr,"Erreur : erreur de lecture sur un des serveur.\n");
            reponse=-1;
      }
      if ((reponse == 0) && (memcmp(msg[0].bloc,msg[1].bloc,TAILLE_BLOC)))
      {
            fprintf(stderr," !!! 2 serveurs ont répondu OK, mais données <>\n");
            reponse=-1;
      }
      if (reponse == 0)
            memcpy(donnees,msg[0].bloc,TAILLE_BLOC);
      return(reponse);
}

void  termine_fils()
{
      int i;
      for (i = 0 ; i < 2 ; i++)
      {
            close(tubes_pere_vers_fils[i][1]);
            close(tubes_fils_vers_pere[i][0]);
            kill(pid_fils[i], SIGUSR1);
      }
      while (wait(NULL) > 0) ;
}

int   main(int argc, char *argv[])
{
      char  un_bloc[TAILLE_BLOC];
      long  un_num_bloc;
      if (argc!=3)
      {
            fprintf(stderr,"Syntaxe : %s adresse1 adresse2\n",argv[0]);
            exit(-1);
      }
      adrs_serveur[0]=argv[1];
      adrs_serveur[1]=argv[2];
      initialise_fils();
      /* bla bla bla... suite d'appels ecrit_bloc(un_num_bloc, un_bloc) et */
      /* lit_bloc(un_num_bloc, un_bloc) ...bla bla bla */
      termine_fils();
      return(0);
}                                   /* Fin fichier "client.c" */
```

Annexe : les appels systèmes utiles (extrais des « man » Linux)

- `int socket(int domain, int type, int protocol)` : crée un point de communication, et renvoie un descripteur (ou –1 si erreur). Ici, **domain** doit être `PF_INET`, le **type** `SOCK_DGRAM` (protocole UDP), le protocole à 0.
- `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen)` : fournit à la socket **sockfd**, l'adresse **my_addr**, adresse de longueur **addrlen** octets. Retourne 0 si OK, -1 sinon. Avec : struct sockaddr_in {

```
        short sin_family;        /* AF_INET */
        u_short sin_port;        /* Num. de port en endian Internet */
        struct in_addr sin_addr; /* Adresse IP (en endian Internet) */
        char sin_zéro [8];       /* Bourrage */
    };
    struct in_addr {
          u_long s_addr;
    };
```

- `int recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, int *fromlen)` : utilisé pour recevoir des messages depuis un socket. Avec **s** le descripteur de socket, **buf** une zone mémoire de taille minimum **len**, **len** le nombre d'octets demandés, **from** est remplis par l'adresse source, et **fromlen** la taille (en octets) de cette adresse source. Retourne le nombre d'octets reçus, -1 en cas d'erreur.
- `int sendto(int s, const void *buf, int len, int flags, const struct sockaddr *to, int tolen)` : envoie dans le socket **s** les **len** octets pointés par **buf**, à destination de l'adresse **to** de longueur **tolen**. Ici, **flags** vaut 0. Retourne le nombre d'octets écrits, -1 en cas d'erreur.
- `bzero(void *ptr, int n)` : mets à 0 les **n** octets pointés par **ptr**.
- `memcpy(void *dest, void *source, int n)` : copie les **n** octets pointés par **source** vers **dest**.
- `memcmp(void *ad1, void *ad2, int n)` : retourne 0 si les **n** octets pointés par **ad1** sont égaux à ceux pointés par **ad2**, et un nombre non nul sinon.
- `int htons(int n)` : normalise l'entier **n** au format réseau.
- `int htonl(long l)` : normalise l'entier long **l** au format réseau.
- `int ntohs(int n)` : normalise l'entier **n** au format local.
- `int ntohl(long l)` : normalise l'entier long **l** au format local.
- `struct hostent *gethostbyname(const char *name)` : retourne l'adresse IP de la machine de nom **name**, dans une structure `struct hostent {`

```
        char    *h_name;       /* Nom officiel de l'hôte. */
        char   **h_aliases;    /* Liste d'alias. */
        int      h_addrtype;   /* Type d'adresse de l'hôte. */
        int      h_length;     /* Longueur de l'adresse. */
        char   **h_addr_list;  /* Liste d'adresses. */
    }
    #define h_addr  h_addr_list[0] /* pour compatibilité. */
```

- `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)` : permet à un programme de surveiller des descripteurs de fichiers contenus dans les ensembles **readfds** (ensemble de descripteurs à surveiller pour voir si des octets sont disponibles en lecture), **writefds** (ensemble de descripteurs à surveiller pour voir si des octets sont disponibles en écriture), **exceptfds** (recherche des conditions exceptionnelles). **nfds** est égal à 1 + numéro du descripteur le plus grand des 3 ensembles **readfds**, **writefds**, et **exceptfds**. **timeout** permet de gérer un timer, et est de type : `struct timeval {`

```
        long    tv_sec;         /* secondes      */
        long    tv_usec;        /* microsecondes */
    };
```

La fonction retourne le nombre de descripteurs ayant une action possible, 0 en cas de timeout. Quatre macros sont disponibles pour la manipulation ensembles : « `void FD_ZERO(fd_set *`**`set`**`);` » efface un ensemble. « `void FD_SET(int` **`fd`**`, fd_set *`**`set`**`);` » et « `void FD_CLR(int` **`fd`**`, fd_set *`**`set`**`);` » ajoutent et suppriment un descripteur dans un ensemble. « `int FD_ISSET(int` **`fd`**`, fd_set *`**`set`**`);` » vérifie si un descripteur est contenu dans un ensemble.

- `int pipe(int` **`filedes`**`[2])` : crée une paire de descripteurs de tube, et les place dans un tableau **`filedes`** (**`filedes`****[0]** est utilisé pour la lecture, et **`filedes`****[1]** pour l'écriture). Retourne –1 si erreur, 0 sinon.
- `int read(int` **`fd`**`, void *`**`buf`**`, size_t` **`count`**`)` : lit count octets depuis le fichier **`fd`** et place les octets lus dans **`buf`**. Retourne le nombre d'octets lus, -1 sinon.
- `int write(int` **`fd`**`, const void *`**`buf`**`, size_t` **`count`**`)` : écrit dans le fichier **`fd`** **`count`** octets pointés par **`buf`**. Retourne le nombre d'octets écrits, -1 sinon.
- `close(int` **`fd`**`)` : ferme proprement le descripteur **`fd`**.
- `void (*signal(int` **`signum`**`, void (*`**`handler`**`)(int)))(int)` : positionne la fonction **`handler`** comme fonction appelée en cas de réception du signal **`signum`**.
- `pid_t wait(int *`**`status`**`)` : attend la fin d'un enfant. Si **`status`** est non NULL, met dans ***`status`** le code retourné par cet enfant.