Université de Strasbourg
Systèmes d'exploitation

Recueil d'exercices

2021 – 2022

Disponible sur https://gitlab.com/pdagog/ens.

Table des matières

Thème 1 - Langage C : Éléments de base

Exercice 1.1

Écrivez un programme qui affiche à l'écran les valeurs de 2^i ($0 \leq i < 10$) en utilisant la fonction `printf`.

Exercice 1.2

Écrivez un programme qui lit des caractères (avec la fonction `getchar`) sur l'entrée standard et compte le nombre d'occurrences de chaque lettre (on ne distinguera pas les minuscules des majuscules).

Exercice 1.3

Écrivez un programme qui lit des phrases (c'est-à-dire des suites de caractères quelconques) sur l'entrée standard et compte le nombre de mots (c'est-à-dire les suites de caractères composées exclusivement de lettres minuscules ou majuscules) qui s'y trouvent.

Exercice 1.4

Modifiez le programme de l'exercice précédent pour afficher seulement les mots (et pas les chiffres, les signes de ponctuation, etc.), un par ligne.

Exercice 1.5

Avec les fonctions `getchar` et `putchar`, écrivez un programme qui affiche les 10 dernières lignes lues sur l'entrée standard.

Exercice 1.6

Écrivez un programme qui lit une chaîne de caractères sur l'entrée standard (avec `fgets`), la recopie (avec `strncpy`) dans une autre chaîne, puis affiche la nouvelle chaîne (avec `puts`) et sa longueur (avec `strlen` et `printf`).

Exercice 1.7

Écrivez un programme qui lit deux chaînes, puis cherche si la deuxième chaîne fait partie de la première, et affiche un message en conséquence. Vous n'utiliserez pas de fonction de librairie autre que `fgets` et `puts`.

Exercice 1.8

On désire avoir un programme qui lit sur l'entrée standard un nom de mois (entre `"janvier"` et `"decembre"`), et qui affiche son numéro (entre 1 et 12) ainsi que le nombre de jours dans ce mois [1]. Si le mois n'est pas valide, il doit afficher un message d'erreur.

Pour cela, on propose de placer les mois valides ainsi que le nombre de jours correspondant, dans une structure :

```
struct mois
{
    char nom [9 + 1] ;      // nom du mois en clair
    int  jours ;            // nombre de jours dans le mois
} ;
```

1. On ne tient pas compte du cas du mois de février lors des années bissextiles.

Déclarez un tableau constant (initialisé) de 12 structures de cette forme pour mémoriser les 12 mois possibles et écrivez le programme.

Exercice 1.9

Les chaînes de caractères et les constantes de type caractère en langage C peuvent contenir les éléments suivants :

— caractères “normaux”, c’est-à-dire tous les caractères de code compris entre 32 et 126 ;
— caractères de contrôle “classiques” (`\n`, `\r`, `\t`, et `\b`) ;
— caractères non représentables tels quels (`\\`, `\’` et `\"`) ;
— autres caractères, dont la valeur numérique est comprise entre 0 et 31 ou supérieure à 127. Dans ce cas, la représentation est un antislash (\) suivi du code octal du caractère.

Écrivez un programme qui lit une chaîne de caractères contenant éventuellement des caractères spéciaux, puis place dans une deuxième chaîne la représentation C (c’est-à-dire avec le caractère `\t` remplacé par les deux caractères \ et `t` par exemple).

Exercice 1.10

Écrivez le programme inverse de l’exercice 1.9, pour transformer une chaîne de caractère contenant éventuellement des caractères spéciaux en représentation C, en chaîne de caractères contenant les caractères traduits (c’est-à-dire avec les deux caractères \ et `t` remplacés par le caractère `\t` par exemple).

Exercice 1.11

Le programmeur qui a écrit la définition ci-dessous aura certainement quelques problèmes lorsqu’il l’utilisera. Expliquez ces problèmes et aidez ce pauvre programmeur à améliorer sa définition.

```
#define carre(x) x*x
```

Exercice 1.12

Expliquez les différences entre votre version corrigée de l’exercice précédent et la définition ci-dessous :

```
int carre (int x) { return x * x ; }
```

Exercice 1.13

Pourquoi la fonction ci-après peut donner des résultats faux ?

```
int factorielle (int n)
{
    int f ;
    if (n <= 1) f = 1 ; else f = n * factorielle (--n) ;
    return f ;
}
```

Exercice 1.14

Écrivez la fonction :

```
unsigned int lire_16 (void)
```

qui lit avec `fgets` une chaîne contenant des chiffres hexadécimaux, et renvoie la valeur convertie. Vous n'utiliserez pas les opérateurs `*`, `/` et `%`.

Exercice 1.15

Écrivez la fonction :

```
void ecrire_2 (unsigned int)
```

qui prend un nombre en paramètre, et en affiche la représentation binaire sur la sortie standard. Comme pour l'exercice 1.14, vous n'utiliserez pas les opérateurs `*`, `/` et `%`.

Thème 2 - Langage C : Fonctions d'entrées/sorties

Exercice 2.1

Écrivez un programme qui recopie un fichier `toto` vers un fichier `titi` à créer, à l'aide des fonctions `fopen`, `putc`, `getc` et `fclose`.

Exercice 2.2

Écrivez un programme pour compter le nombre de lignes sur l'entrée standard, en utilisant la fonction `getchar`.

Variante : votre programme doit prendre un argument optionnel, le caractère à compter.

Exercice 2.3

On désire construire un programme pour réaliser un carnet d'adresses rudimentaire. Le carnet est placé dans un fichier, composé de fiches représentées par des structures du langage C :

```
#define MAXNOM 50
#define MAXTEL 10

struct fiche
{
    int  occupe ;               // vrai si la fiche est utilisée
    char nom [MAXNOM] ;         // le nom de la personne
    char telephone [MAXTEL] ;   // son numéro de téléphone
} ;

typedef struct fiche fiche ;
```

Lorsqu'une fiche est détruite, le champ `occupe` est initialisé à la valeur 0 (*faux*). Cette fiche pourra être réutilisée par la suite.

L'accès au carnet est réalisé par une suite de petits programmes accomplissant chacun une action bien précise :

- `chercher`
  Ce programme prend un argument, le nom à chercher. Si le nom est trouvé, le numéro de la fiche est affiché sur la sortie standard. Dans le cas contraire, rien n'est affiché.
- `afficher`
  Ce programme prend un argument, le numéro de la fiche à afficher. Si ce numéro n'existe pas (argument vide), aucune fiche n'est affichée.
- `ajouter`
  Ce programme prend deux arguments, un nom et un numéro de téléphone. La fiche est ajoutée dans le fichier, soit en récupérant une fiche inoccupée, soit en l'ajoutant à la fin du fichier.
- `detruire`
  Ce programme prend un argument, le numéro de la fiche à détruire.

Exercice 2.4

La commande Unix `tail` affiche, par défaut, les 10 dernières lignes d'un fichier texte. L'option `-n` permet de changer ce nombre de lignes. Programmez votre propre version de cette commande `tail`

Pour analyser les arguments, il faut utiliser la fonction `getopt`. Pour repérer les n dernières lignes d'un fichier, une méthode suggérée consiste à lire les m derniers octets du fichier et à compter le nombre de caractères `\n` en partant de la fin. Si ce nombre n'est pas suffisant, on recommence. Lorsque le nombre n de `\n` est atteint, on affiche tous les caractères jusqu'à la fin du fichier.

Exercice 2.5

Le programme `tar` est un utilitaire d'archivage : il recopie toute une arborescence et stocke tous les fichiers bout à bout dans un seul fichier (l'*archive*). Ceci peut être utile pour réaliser des sauvegardes (la cartouche ou

la bande est alors vue par le système comme le fichier archive), pour transmettre des fichiers multiples par le réseau, par courrier électronique ou bien avec `ftp`.

On désire programmer deux utilitaires d'archivage comparables à `tar`.

1. la syntaxe de l'archiveur est :

   ```
   archiver archive f1 f2 ... fn
   ```

   Le fichier `archive` est le fichier dans lequel les fichiers `f1` à `fn` sont recopiés et mis bout à bout. Chaque fichier y est précédé de son nom et de sa taille en octets.
   Note : quelles précautions doit-on prendre pour qu'un fichier archivé sur une machine puisse être désarchivé sur une autre ?
2. la syntaxe du désarchiveur est :

   ```
   desarchiver archive
   ```

   Cette commande analyse le fichier `archive` et en extrait tous les fichiers.

Thème 3 - Langage C : Pointeurs

Exercice 3.1

À l'aide d'un papier et d'un crayon, reconstituez le déroulement du programme suivant. Déduisez-en l'affichage sur la sortie standard, ainsi que les endroits auxquels le programme provoque une erreur.

```
#define PRabp(n)    printf ("Ligne␣%d:␣" \
        "a:␣0x%x␣%d,␣b:␣0x%x␣%d,␣p:␣0x%x␣0x%x␣%d\n", \
        n, &a, a, &b,b,&p, p, *p)
#define PRtp(n)     printf( "Ligne␣%d:␣" \
        "t:␣0x%x␣%d␣%d␣%d,␣p:␣0x%x␣0x%x␣%d\n", \
        n, t, t[0], t[1], t[2],&p, p, *p)

int main (int argc, char *argv [])
{
    int a, b ;
    int t [3] = { 4, 5, 6 } ;
    int *p ;

    p=&b ; a=0 ; b=2 ;        PRabp (1) ;
    *p=4 ;                    PRabp (2) ;
    p++ ;                     PRabp (3) ;
    (*p)++ ;                  PRabp (4) ;
    p=0 ;                     PRabp (5) ;
    p=t ;                     PRtp (6) ;
    p[0] = 10 ; p[1] = 11 ;   PRtp (7) ;
    p++ ;                     PRtp (8) ;
    *p = 15 ; *(p+1) = 16 ;   PRtp (9) ;
    p++ ;                     PRtp (10) ;
    p[0] = 20 ; p[1] = 21 ;   PRtp (11) ;
    return 0 ;
}
```

Exercice 3.2

Écrivez un programme qui lit une liste de nombres entiers (avec les fonctions `fgets` et `atoi`) jusqu'à la fin de fichier, les enregistre dans une liste en les triant au fur et à mesure, puis affiche les nombres de la liste (avec la fonction `printf`).

Exercice 3.3

Modifiez le programme de l'exercice précédent pour mettre dans la liste (qui doit toujours être triée) des chaînes de caractères lues au clavier.

Exercice 3.4

Lorsqu'une structure est déclarée, chaque champ est placé à une adresse relative au début de la structure. Soit une structure `s` et une champ `c`, expliquez comment obtenir la valeur numérique de cette adresse relative.

Exercice 3.5

Écrivez un programme qui lit deux chaînes, puis cherche si la deuxième chaîne fait partie de la première, et affiche un message en conséquence (voir exercice 1.7). Cette fois-ci, vous utiliserez les fonctions de librairie

`strlen` et `strncmp`.

Exercice 3.6

Écrivez la fonction `char *mon_strchr (char *chaine, int car)` sans utiliser de fonction de librairie (et surtout pas `strchr`).

Exercice 3.7

Quelles sont les erreurs dans les fonctions ci-après ?

```
int *f1 (int val)
{
    int *p ;
    p = val == 0 ? NULL : &val ;
    return p ;
}

char *f2 (void)
{
    char tab [3] ;
    strcpy (tab, "abc") ;
    return tab ;
}

static char *texte [] = { "une", "suite", "de", "lignes", NULL, } ;

void f3 (void)
{
    char **p ;
    for (p = texte ; *p ; p++)
        printf ("%s\n", p) ;
}

char tab [] = "abcdef\n" ;

void f4 (void)
{
    int i ;
    for (i = 0 ; *(tab + i) ; i++)
        putchar (*(tab + i)) ;
    for ( ; *tab ; tab++)
        putchar (*tab) ;
}

void f5 (char tab [], int pas)
{
    char *chaine = "abcdef" ;
    int i ;
    for (i = 0 ; i < sizeof (chaine) ; i += pas)
        chaine [i] = '-' ;
    strcpy (tab, chaine) ;
}
```

Exercice 3.8

On désire écrire un programme pour lire une série de nombres flottants et les placer dans un tableau (on supposera qu'il n'y a pas plus de 100 valeurs), trier ce tableau et enfin l'afficher sur la sortie standard. Pour cela, on propose la décomposition en fonctions :

1. `int lire_tableau (double *tableau, int max_elem, int *nb_elem)`
   qui reçoit un tableau en paramètre contenant au plus `max_elem` éléments, lit les valeurs et les range dans le tableau, puis renvoie le nombre d'éléments lus dans `nb_elem`. La valeur de retour de cette fonction doit être 0 si la lecture s'est bien passée, -1 sinon.
2. `void trier_tableau (double *tableau, int nb_elem)`
   qui trie les `nb_elem` éléments du tableau `tableau`.
3. `void afficher_tableau (double *tableau, int nb_elem)`
   qui affiche les `nb_elem` éléments du tableau `tableau`.

Programmez ces fonctions ainsi que `main`.

Exercice 3.9

Écrivez un programme qui lit une série d'entiers sur l'entrée standard, les range dans un tableau (on supposera qu'on ne lit pas plus de 100 nombres), puis trie ce tableau et enfin l'affiche sur la sortie standard. Vous n'avez pas le droit d'utiliser l'opérateur `[...]`.

Exercice 3.10

Écrivez la déclaration d'un tableau de 50 pointeurs sur des structures contenant chacune un pointeur sur un caractère, un tableau de 10 entiers et un pointeur sur cette même structure. Donnez une définition de ce type avec `typedef`.

Exercice 3.11

La fonction `fgets` a un défaut : l'espace dans lequel les caractères lus sont rangés doit être alloué par la fonction appelante. Écrivez une nouvelle fonction :

```
char *mon_fgets (void)
```

qui lit des caractères sur l'entrée standard (on supposera qu'on ne lit pas plus de 100 caractères) et renvoie l'adresse d'une zone mémoire où votre fonction les aura rangés.

Exercice 3.12

On désire remplacer la fonction de lecture de tableau de l'exercice 3.8 en éliminant la contrainte d'un nombre d'entrées maximum. Pour cela, on change le prototype en :

```
int lire_tableau (double **tableau, int *nb_elem)
```

Le paramètre `tableau` reçoit en sortie un tableau avec le nombre suffisant d'entrées pour mémoriser tous les éléments lus. Vous commencerez par allouer (avec `calloc`) un tableau de $n = 100$ entrées. Lorsque n éléments sont saisis, allouez un nouveau tableau de taille $2n$ (sans utiliser `realloc`), recopiez-y les n éléments et désallouez le premier tableau.

Exercice 3.13

Comment peut-on déduire que le programmeur qui a écrit la fonction ci-dessous est débutant ?

```
int main (void)
{
    char *ligne ;

    ligne = malloc (80) ;
    while (gets (ligne) != NULL)
        puts (ligne) ;
}
```

Exercice 3.14

On désire manipuler des expressions (composées des quatre opérations classiques manipulant exclusivement des entiers) représentées sous forme d'arbres. Pour cela, on définit une structure :

```
enum code { noeud, feuille } ;

struct noeud
{
    enum code code ;
    union
    {
        int valeur ;
        struct
        {
            char operation ;
            struct noeud *fils_gauche ;
            struct noeud *fils_droit ;
        } s ;
    } u ;
} ;
```

Un nœud de l'arbre est identifié par le code `noeud` et contient l'opération et ses deux opérandes. Une feuille est identifiée par le code `feuille` et contient la valeur entière.

1. Étant donnée l'expression spécifiée en notation préfixée :

   ```
   (* (* 3 (+ 1 2)) (+ 4 5))
   ```

   — dessinez l'arbre correspondant ;
   — donnez l'expression équivalente en notation algébrique.
2. Écrivez une fonction :

   ```
   struct noeud *lire_prefixe (void)
   ```

   qui lit sur l'entrée standard une expression en notation préfixée et la mémorise sous forme d'un arbre dont la racine est la valeur de retour.
3. Écrivez une fonction :

   ```
   void ecrire_prefixe (struct noeud *arbre)
   ```

   qui affiche sur la sortie standard l'arbre en notation préfixée.
4. Écrivez une fonction :

   ```
   int profondeur_arbre (struct noeud *arbre)
   ```

   qui renvoie la profondeur maximum de l'arbre.
5. Écrivez une fonction :

   ```
   int evaluer_arbre (struct noeud *arbre)
   ```

   qui renvoie la valeur de l'expression mémorisée dans l'arbre. Pour choisir et réaliser une opération, vous utiliserez le choix multiple (`switch`) pour sélectionner la fonction `ajouter`, `soustraire`, `multiplier` ou `diviser`. Ces fonctions prennent les deux opérandes comme arguments et renvoient le résultat de l'opération.
6. Écrivez une fonction :

   ```
   void ecrire_algebrique (struct noeud *arbre)
   ```

   qui affiche l'expression mémorisée dans l'arbre comme une expression en notation algébrique (infixée). Par exemple, l'expression ci-dessus peut être écrite : `((3*(1+2))*(4+5))`.

Exercice 3.15

Reprendre la fonction d'évaluation d'une expression mémorisée dans un arbre (exercice 3.14). Cette fois-ci, l'évaluation d'une opération doit être réalisée à l'aide d'un tableau de pointeurs sur des fonctions.

Exercice 3.16

On définit une liste doublement chaînée par :

```
struct element
{
    int clef ;                  // clef de recherche
    char *valeur ;              // valeur associée
    struct element *suivant ;   // suivant dans la liste
    struct element *precedent ; // précédent dans la liste
} ;

struct element tete ;
```

La variable `tete` est un élément ne contenant aucune clef et aucune valeur. Le champ `suivant` pointe sur le premier élément de la liste, le champ `precedent` pointe sur le dernier. La liste doit toujours être triée suivant la clef.

Écrivez les fonctions :

- `void initialiser_liste (void)`
  initialise la structure de liste ;
- `void ajouter_element (int clef, char *valeur)`
  ajoute un élément. La valeur doit être recopiée ;
- `char *lire_valeur (int clef)`
  renvoie un pointeur sur la valeur (sans la recopier) ;
- `void retirer_element (int clef)`
  retire l'élément identifié par sa clef.

Exercice 3.17

La variable Shell `PATH` contient une liste de répertoires séparés par des caractères " : ". Par exemple :

```
/bin:/usr/bin:/usr/local/bin
```

Écrivez une fonction qui prenne en paramètre une chaîne et sépare dans un tableau de chaînes les différents constituants.

Écrivez un programme qui lit le contenu de la variable `PATH` (avec la fonction `getenv`), sépare les différents constituants à l'aide de la fonction que vous venez de programmer, et qui affiche ces différents constituants.

Exercice 3.18

On désire réaliser un module de gestion de pile générique (c'est-à-dire qui soit capable de gérer des éléments de type quelconque). Pour cela, on définit les fonctions suivantes :

- `PILE *initialiser_pile (int taille)`
  initialise les structures de données nécessaires. Le paramètre `taille` est la taille d'un élément de la pile.
- `void empiler (PILE *pile, void *valeur)`
  empile l'élément situé à l'adresse `valeur` ;
- `int pile_vide (PILE *pile)`
  teste si la pile est vide ;
- `void depiler (PILE *pile, void *valeur)`
  dépile l'élément au somment et le recopie à l'adresse spécifiée par `valeur`.

Définissez le type `PILE` (comme une liste doublement chaînée), et programmez ces fonctions.

Exercice 3.19

On désire réaliser une fonction de tri polymorphique, c'est-à-dire qui soit capable de trier des objets de n'importe quel type. Cette fonction a la syntaxe suivante :

```
void quicksort (void *tab [], int debut, int fin,
                int (*comp) (void *, void *))
```

Cette fonction utilise l'algorithme du tri rapide (*quick sort*) pour trier un les éléments d'indice compris entre `debut` et `fin` (bornes incluses) d'un tableau d'éléments référencés par un pointeur générique. Pour comparer deux éléments, `quicksort` utilise une fonction prenant en paramètre deux pointeurs génériques et renvoyant -1 si le premier argument est inférieur au deuxième, 0 s'ils sont égaux, et 1 si le premier est supérieur au deuxième.

1. écrivez un programme qui lit une suite de nombres flottants, utilise `quicksort` pour les trier, et enfin affiche le tableau trié (on suppose qu'il ne peut pas y avoir plus de 10000 nombres) ;
2. écrivez un programme analogue pour trier des chaînes de caractères (on suppose qu'il ne peut pas y avoir plus de 10000 chaînes, et chaque chaîne est limitée à 1000 caractères) ;
3. écrivez enfin la fonction `quicksort`.

Exercice 3.20

Les sciences physiques sont consommatrices de puissance de calcul. Les problèmes traités font souvent apparaître des *matrices creuses*, c'est-à-dire des matrices peuplées essentiellement d'éléments nuls.

Par exemple, une matrice $10\,000 \times 10\,000$ représentée sous la forme d'un tableau bidimensionnel occupe une place correspondant à 100 000 000 fois la taille d'une valeur flottante, soit environ 400 Mo si 32 bits sont utilisés (ce qui correspond à la simple précision sur la plupart des processeurs actuels). Si cette matrice contient seulement deux nombres non nuls par ligne, il n'y a que 80 000 octets d'information "utile", d'où une perte énorme.

Pour résoudre ce problème, on propose d'utiliser un codage différent pour les matrices creuses : on représente un élément non nul par un triplet (i, j, v) où i et j sont les numéros de ligne et de colonne et v est la valeur de cet élément. Ces éléments sont placés dans une liste chaînée. Pour simplifier l'implémentation, on ne cherchera pas à minimiser les temps de calculs.

Programmez les fonctions :

```
typedef struct element *matrice ;

matrice ajouter_matrice (matrice M1, matrice M2, int n)
matrice multiplier_matrice (matrice M1, matrice M2, int n)
```

Note : on suppose que les matrices sont de dimension $n \times n$.

Thème 4 - Langage C : Programmation avancée

Exercice 4.1

Écrivez en C une fonction `afficher_couples` :

```
void afficher_couples (int nbcouples, ...)
```

qui prend un nombre de couples, puis une suite de couples (chaîne, entier) à afficher sur la sortie standard. Votre fonction doit prendre un nombre variable d'arguments (voir `stdarg.h`)

Exercice 4.2

On désire écrire un programme pour calculer la circonférence et la surface d'un cercle étant donné son rayon. Le programme doit être composé de plusieurs fichiers :

`principal.c`	contient la fonction `main`
`circonference.c`	contient la fonction `circonference`
`surface.c`	contient la fonction `surface`
`pi.h`	contient la définition de la valeur approchée de π

1. Écrivez le contenu des différents fichiers, et donnez la commande minimum de compilation du programme.
2. Décomposez cette commande de manière à avoir une commande pour la compilation de chaque fichier, puis une commande pour l'édition de liens.
   Si vous modifiez le fichier `surface.c`, donnez les commandes minimum pour recompiler le programme. Même question pour la modification du fichier `pi.h`.
3. Représentez les dépendances de l'exercice précédent sous forme de graphe étiqueté. Écrivez le fichier `Makefile` correspondant.
   Ajoutez une cible fictive `clean` dans le `Makefile` pour supprimer tous les fichiers intermédiaires ainsi que le fichier résultat.

Exercice 4.3

On désire connaître le nombre d'occurrences de chaque mot lu sur l'entrée standard. Par exemple, le texte "*il fait beau, n'est-il pas ?*" est constitué de :

```
il    : 2 fois
fait  : 1 fois
beau  : 1 fois
n     : 1 fois
est   : 1 fois
pas   : 1 fois
```

Pour cela, on définit les fonctions :

— `void initialiser_mots (void)`
qui initialise la structure de données associée aux mots ;
— `struct mot *trouver_mot (char texte [])`
qui trouve un mot, et le crée s'il n'est pas trouvé ;
— `void mettre_a_jour_mot (struct mot *mot)`
qui met à jour (incrémente) le compteur associé au mot ;
— `void afficher_mots (void)`
qui affiche le résultat final.

1. Écrivez le programme en supposant que les fonctions ci-dessus existent déjà et sont dans un autre module (fichier). Vous placerez dans un fichier `mot.h` les déclarations des prototypes de ces fonctions. Y a-t-il besoin de placer la définition de la structure `mot` dans ce fichier ?

2. Écrivez les fonctions ci-dessus en prenant comme structure de données une liste simplement chaînée :

```
struct mot
{
    char *texte ;           // texte associé au mot
    int   nb_occurrences ;  // nb d'occurrences du mot
    struct mot *suivant ;   // liste chaînée
} ;
struct mot *tete ;
```

3. Écrivez une nouvelle version des fonctions ci-dessus en utilisant une table de hachage.

Exercice 4.4

À l'aide des mécanismes de compilation conditionnelle offerts par le préprocesseur, reprenez l'exercice précédent et intégrez les deux structures de données pour représenter les mots (liste chaînée et table de hachage) en un seul fichier. La personne qui compile votre programme doit pouvoir sélectionner l'une ou l'autre structure en définissant un seul symbole ou en ne le définissant pas.

Exercice 4.5

Sur Linux, la page de manuel *p* du chapitre *n* est localisée dans le fichier de nom `/usr/share/man/man`*n*`/`*p*`.`*n*`.gz`. Cette page est compressée, et la commande pour la formater est : `gunzip | nroff -man` en supposant que le fichier est présenté sur l'entrée standard.

Sur HP-UX, la page de manuel *p* du chapitre *n* est localisée dans le fichier de nom `/usr/share/man/man`*n*`.Z/`*p*`.`*n*. Cette page est compressée, et la commande pour la formater est : `zcat | nroff -man` en supposant que le fichier est présenté sur l'entrée standard.

Sur SunOS, la page de manuel *p* du chapitre *n* est localisée dans le fichier de nom `/usr/man/man`*n*`/`*p*`.`*n*. Cette page n'est pas compressée, et la commande pour la formater est : `nroff -man` toujours en supposant que le fichier est présenté sur l'entrée standard.

Écrivez une commande `man`. Votre programme doit être portable sur Linux, HP-UX ou sur SunOS, en utilisant le symbole `linux`, `hpux` ou `sun` du préprocesseur. Vous utiliserez la fonction `popen` pour formater la page de manuel.

Rappel : la vraie commande `man` admet 1 ou 2 paramètres :

`man` [*n*] *p*

Si le chapitre *n* n'est pas fourni, la page correspondant à *p* est cherchée dans l'ensemble des chapitres. On supposera que le chapitre est un entier entre 1 et 8.

Thème 5 - Système : Gestion de fichiers

Exercice 5.1

Nommez quelques primitives système. Est-ce que `fopen` est une primitive système ?

Quelles sont les différences et ressemblances entre primitives système et fonctions de bibliothèque ? Comment justifier ces différences ?

Illustrez ces différences et ressemblances sur les fonctions et primitives d'entrée / sortie.

Exercice 5.2

Pourquoi la fonction ci-dessous ne peut pas fonctionner ? Détaillez toutes les fautes. On ne demande pas de corriger cette fonction.

```
int faux (char *nom)
{
    FILE *fp ;
    int c ;

    fp = open (nom, "r") ;
    read (fp, &c, 1) ;
    fclose (fp) ;
    return c ;
}
```

Exercice 5.3

Écrivez un programme qui recopie un fichier `toto` vers un fichier `titi` à créer, à l'aide des primitives système. Vous ne chercherez pas à créer le nouveau fichier avec les permissions du fichier original.

Exercice 5.4

Écrivez un programme pour réaliser une copie de fichier. Votre programme doit prendre exactement deux arguments : le chemin du fichier existant, et le chemin du fichier (qui peut déjà exister ou non) qui contiendra la copie à la fin de l'exécution.

Vérifiez (à l'aide d'un grand fichier) que la copie s'est bien déroulée : la commande `cmp` appelée avec les deux chemins ne doit rien afficher.

Exercice 5.5

Écrivez la fonction `getchar` qui renvoie un caractère lu sur l'entrée standard, ou la constante `EOF` en fin de fichier.

Quelle peut être la valeur numérique de la constante EOF ?

Pour tester votre implémentation, rédigez une fonction `main` qui recopie tous les caractères lus sur l'entrée standard avec votre fonction et les recopie sur la sortie standard avec la fonction `putchar` (qu'on ne vous demande pas d'écrire). Utilisez votre programme en redirigeant l'entrée standard depuis un grand fichier, binaire de préférence (un programme dans `/bin` par exemple), et la sortie standard vers un nouveau fichier. Utilisez la commande `cmp` pour comparer l'original et la sortie de votre programme : si `cmp` n'affiche rien, c'est que les deux fichiers sont identiques.

Exercice 5.6

Écrivez une version bufferisée de `getchar`. Pour ce faire, votre version doit conserver un tampon en mémoire.

Si le tampon est vide lorsque votre `getchar` est appelée, celle-ci doit remplir le tampon et en retourner le premier octet. Si le tampon contient déjà des octets, votre `getchar` doit retourner l'octet suivant.

Exercice 5.7

On croit souvent que les primitives système étant de plus bas niveau, elles sont plus efficaces que les fonctions de bibliothèque équivalentes. On désire confirmer ou infirmer cette proposition par l'expérimentation.

Pour cela, on demande de rédiger deux programmes pour copier l'entrée standard sur la sortie standard. Le premier utilisera les fonctions de bibliothèque `getchar` et `putchar`. Le deuxième utilisera les primitives système `read` et `write` et prendra en argument la taille du buffer utilisé pour la copie. Si cette taille égale 1, la copie sera effectuée caractère par caractère.

Vous utiliserez la commande Unix `time` pour comparer les temps d'exécution, en considérant la somme des temps CPU en mode utilisateur et en mode système [2]. Vous attacherez la plus grande importance à la qualité de vos mesures : valeurs significatives (utilisez de grands fichiers en entrée), élimination des variations (moyenne de plusieurs valeurs), élimination des perturbations dues à la sortie sur votre écran (redirigez la sortie standard), etc.

En prenant comme tailles de buffer les puissances successives de 2 (2^0, 2^1, 2^2, 2^3, etc.), à partir de quelle taille de buffer est-il plus intéressant d'utiliser les primitives système que les fonctions de bibliothèque ?

Exercice 5.8

Écrivez un programme qui affiche en clair le type du fichier demandé (répertoire, fichier ordinaire, etc.), ainsi que ses permissions (lecture, écriture et exécution) en octal (base 8, avec le format `%o` de `printf`).

En option : affichez les permissions sous la même forme que la commande `ls` avec l'option `-l`).

Exercice 5.9

On désire implémenter une nouvelle version de la librairie standard d'entrées/sorties à l'aide des primitives système.

1. Donnez une définition du type `FICHIER`. N'oubliez de prévoir la bufferisation des entrées/sorties.
2. Programmez la fonction `my_open`, analogue à `fopen`. Pour simplifier, on ne considérera que les modes d'ouverture `"r"` et `"w"`.
3. Reprenez l'exercice 5.6 pour programmer `my_getc`, analogue à `getc`.
4. Programmez la fonction `my_putc`, analogue à `putc` (qui bufferise en sortie de la même manière que `getc` bufferise en entrée).
5. Programmez la fonction `my_close`, analogue à `fclose`.
6. Pour tester, écrivez une fonction `main` qui ouvre deux fichiers `toto` et `tata` en lecture, puis qui lit en boucle 1 caractère dans chacun des deux fichiers et les affiche sur la sortie standard. Votre boucle s'arrêtera dès que vous rencontrez la première fin de fichier.

Exercice 5.10

Écrivez une commande qui prend en paramètre un nom de répertoire, et qui affiche tous les objets contenus dans ce répertoire. On prendra les mêmes conventions de restriction d'affichage que la commande `ls` (pas d'affichage des noms commençant par un point).

Exercice 5.11

Reprenez le programme de l'exercice 5.4 pour recopier toute une arborescence. Votre programme prendra en paramètre deux noms de répertoire : un nom de répertoire existant à copier et un nom de répertoire destination

2. Le temps « réel » correspond au temps réellement écoulé : on n'utilisera pas cette valeur car elle dépend de la charge du système.

(que l'on pourra supposer inexistant pour simplifier).

Exercice 5.12

Reprenez le programme de l'exercice précédent pour restaurer dans les copies les dates d'accès et modification ainsi que les permissions des fichiers originaux.

Exercice 5.13

Rédigez un programme pour afficher tous les noms des fichiers d'une arborescence dont le contenu comprend une chaîne de caractères. Votre programme doit admettre la syntaxe suivante :

`chercher` *chaine répertoire*

Par exemple « `chercher "struct utimbuf" /usr/include` » doit afficher :

```
/usr/include/utime.h
/usr/include/linux/utime.h
....
```

Vous utiliserez les primitives système pour rechercher la chaîne dans chaque fichier, en utilisant une méthode efficace (i.e. pas une lecture octet par octet). Pour afficher le nom de chaque fichier trouvé, vous pouvez utiliser la fonction de bibliothèque printf.

Exercice 5.14

Reprenez la fonction développée lors de l'exercice 3.17 pour écrire un programme `which` afin de chercher où une commande est trouvée. Par exemple, "`which ls`" doit donner : `/bin/ls`.

Exercice 5.15

Certains shells (`ksh`, `zsh`, `bash`, etc.) disposent d'une variable `CDPATH`. Celle-ci spécifie un certain nombre de répertoires de recherche. Lorsqu'on utilise `cd` avec un argument (nom de chemin relatif), celui est cherché dans les différents répertoires indiqués par `CDPATH` et le changement de répertoire est effectué s'il est trouvé.

Programmez une commande `chdir`.

Pourquoi cette commande ne peut pas fonctionner ?

Thème 6 - Système : Gestion des processus

Exercice 6.1

Écrivez un programme générant un processus fils avec la primitive système `fork`.

— le processus fils doit afficher son numéro (*pid*) ainsi que le numéro du père à l'aide des primitives système `getpid` et `getppid`, puis sort (primitive `exit`) avec un code de retour égal au dernier chiffre du *pid*.
— le processus père, quant à lui, affiche le *pid* du fils, puis attend sa terminaison (primitive `wait`) et affiche son code de retour.

Exercice 6.2

Écrivez un programme admettant un argument entier n. Ce programme doit lancer n processus fils dans une première étape puis, dans une deuxième étape, attendre leur terminaison à l'aide de la primitive `wait`. À chaque fois qu'un processus se termine, le père affiche son numéro (*pid*) et son code de retour.

Exercice 6.3

Écrivez un programme ayant la syntaxe suivante :

`matproc` *n m*

L'action de ce programme doit être de générer n processus, chacun d'entre eux devant générer n processus à son tour, et ainsi de suite jusqu'à m niveaux.

Combien de processus sont générés au total ?

Exercice 6.4

Écrivez un programme admettant un argument, un nom de fichier, et qui :

1. ouvre le fichier en lecture ;
2. créé un processus fils ;
3. puis les deux processus (père et fils) lisent les caractères l'un après l'autre dans le fichier en utilisant la primitive système `read`, et les affichent à l'écran.

Expérimentez ce programme avec un fichier texte de taille assez grande. Vérifiez que le bon nombre de caractères a été lu, ou non.

Que se passe-t-il lorsque le fichier est ouvert *après* l'appel à `fork` ?

Exercice 6.5

Écrivez un programme admettant un argument, un nom de répertoire, et qui :

1. affiche d'abord l'heure, sous forme de secondes et de microsecondes (primitive `gettimeofday`) ;
2. appelle ensuite (avec une des primitives `exec`) la commande `ls` avec l'option `-l` sur un répertoire passé en paramètre ;
3. affiche enfin l'heure (comme précédemment) ainsi que le temps écoulé pendant l'appel de `ls`.

Exercice 6.6

Écrivez un programme admettant un argument, un nom de répertoire, et qui :

1. lance la commande `ls` avec l'option `-R` sur le répertoire ;

2. redirige la sortie standard de `ls` sur `/dev/null`;
3. affiche la somme des temps « processeur » consommés par la commande `ls` (primitive système `times`) en secondes.

Exercice 6.7

Écrivez un programme admettant les arguments suivants :

`parexec` *n cmd arg ...*

Votre programme doit appeler n fois (avec `fork` et `execvp`) la commande spécifiée en paramètre en lui passant les arguments (en nombre non connu) fournis, en ajoutant le rang d'appel i ($0 \leq i < n$) comme argument supplémentaire à la fin.

Ainsi, l'appel :

```
parexec 5 sh tester toto
```

doit provoquer l'exécution de :

— `sh tester toto 0`
— `sh tester toto 1`
— ...
— `sh tester toto 4`

Les n commandes doivent être lancées en parallèle, et le code de retour de `parexec` doit être égal à 0 si toutes les commandes réussissent (i.e. renvoient un code de retour nul), ou 1 si au moins l'une des commandes échoue (i.e. renvoie un code de retour non nul).

Exercice 6.8

Rédigez un programme qui prend en argument deux nombres n et m.

— Dans une première étape, votre programme doit générer n nombres aléatoires a_i (avec la fonction de bibliothèque `rand`) compris entre 1 et m, bornes incluses et les placer dans un tableau.
— Dans une deuxième étape, votre programme doit générer n processus fils. Chaque processus fils i (avec $1 \leq i \leq n$) doit attendre (avec la fonction de bibliothèque `sleep`) a_i secondes, puis se terminer avec un code de retour égal à a_i.
— Dans une troisième étape, votre programme doit attendre la terminaison des n processus fils et récupérer les différentes valeurs a_i retournées. Ces valeurs doivent être affichées au fur et à mesure.

Exercice 6.9

Écrivez un programme C équivalent au script shell suivant, qui prend en argument un nom d'utilisateur :

```
ps eaux > toto ; grep "^$1 " < toto > /dev/null && echo "$1 est connecté"
```

Votre programme devra :

— réaliser effectivement les exécutions de `ps` et `grep` en utilisant la primitive `execlp`;
— mettre en place les redirections des entrées/sorties nécessaires grâce à la primitive `dup` (ou `dup2`);
— réaliser l'affichage final avec la primitive `write`.

Exercice 6.10

Écrivez un programme qui :

1. attende une ligne (une commande) sur l'entrée standard (vous pouvez utiliser la fonction de librairie `fgets`);
2. la décompose en mots séparés par des espaces;
3. recherche le premier mot dans le `PATH`;

4. exécute cette commande par le biais de la primitive système `execv;`
5. revienne au point 1.

Félicitations, vous avez écrit un *shell* !

Exercice 6.11

Reprenez le programme de l'exercice précédent et ajoutez les redirections dans des fichiers (à l'aide des mots $<$, $>$ ou les deux).

Exercice 6.12

Reprenez le programme de l'exercice précédent et ajoutez un traitement spécial dans le cas où la commande est terminée par le mot `&`.

Thème 7 - Système : Environnement d'un processus

Exercice 7.1

La variable globale `environ` est déclarée comme :

```
extern char **environ
```

Cette variable est automatiquement définie. Elle référence le premier élément d'un tableau de chaînes de la forme *var=valeur*. Ce tableau est terminé par le pointeur `NULL`.

Dessinez cette structure de données.

Écrivez la fonction de librairie `getenv`.

Exercice 7.2

La fonction de librairie `system` prend en argument une commande (contenant éventuellement des redirections, ou même composée de plusieurs commandes reliées par un tube), l'exécute et renvoie le code de retour de la commande si elle a été lancée, ou -1 sinon.

Rédigez une version de la fonction `system`. Vous utiliserez le Shell de Bourne (`/bin/sh`) avec l'option `-c` pour exécuter la commande.

Exercice 7.3

Il peut arriver qu'un programme désire savoir si l'entrée ou la sortie standard a été redirigée.

Donnez des exemples de programmes dont le comportement dépend de la redirection de l'entrée ou de la sortie standard.

Écrivez la fonction `isatty` qui prend en argument un descripteur de fichier, et renvoie 1 si c'est un terminal (ou plus généralement un périphérique en mode caractère) ou 0 sinon.

Exercice 7.4

Les champs `st_dev` et `st_ino` de la structure `stat` identifient de manière unique un fichier dans le système. Écrivez une fonction `my_ttyname` analogue à la fonction de bibliothèque `ttyname` qui prend en paramètre un descripteur de fichier, et renvoie un pointeur sur une chaîne (statique) contenant le nom complet du fichier correspondant (cherché dans `/dev`[3]), ou `NULL` si le fichier n'est pas trouvé.

Exercice 7.5

Écrivez une nouvelle version de la fonction `getcwd`. Pour cela, on cherchera le numéro d'inode du répertoire courant. Puis, on cherchera dans le répertoire parent le nom correspondant à l'inode du répertoire courant. En répétant cette opération jusqu'au répertoire d'inode numéro 2 (la racine du système de fichiers), on peut reconstituer le nom du répertoire courant.

On notera que cette méthode ne tient compte que du système de fichiers courant. On n'essayera pas de s'affranchir de cette limitation.

3. Sur Linux, on cherchera dans `/dev/pts`.

Thème 8 - Système : Tubes

Exercice 8.1

Écrivez un programme composé de deux processus : le père lit des données sur l'entrée standard et les passe par un *tube* au fils qui les affiche sur sa sortie standard. Le père attend ensuite la terminaison du fils.

Pour débuter, il est suggéré de rédiger une fonction `void copier(int fdsrc, int fddst)` qui recopie le contenu du fichier dont le descripteur est `fdsrc` vers le fichier dont le descripteur est `fddst`, puis d'utiliser cette fonction dans un programme simple (monoprocessus) pour recopier l'entrée standard sur la sortie standard. Vous pouvez tester avec :

```
$ ./a.out < /bin/ls > toto
$ cmp /bin/ls toto
```

Si `cmp` n'affiche aucune erreur, la copie s'est bien passée.

Exercice 8.2

Généralisez l'exercice précédent à n processus : le premier passe les données depuis l'entrée standard au second, qui les passe au troisième, et ainsi de suite jusqu'au $n-1$-ème qui les passe au n-ème (le père), qui les écrit sur sa sortie standard. Il doit donc y avoir $n-1$ tubes, et les $n-1$ fils doivent être les fils directs du processus père qui exécute la fonction `main`.

Exercice 8.3

On désire connaître la capacité d'un tube. Pour cela, on propose d'envoyer des données, en les comptant, dans un tube qu'aucun lecteur ne consulte (ouvert en lecture, mais jamais lu par un processus). Au bout d'un certain nombre d'octets, l'écrivain se bloque en attendant que le tube se vide. Si un signal survient dans cet état, on peut afficher le nombre d'octets placés dans le tube, c'est-à-dire sa capacité.

Faites un programme pour afficher la taille d'un tube avec la méthode ci-dessus.

Exercice 8.4

Écrivez un programme qui crée deux processus fils, redirige les entrées/sorties et appelle la fonction `execlp` pour faire l'équivalent de ce que fait le shell lorsqu'on tape la commande :

```
ps eaux | grep "^<nom>" | wc -l
```

Le nom sera donné par l'utilisateur en argument du programme, ou prendra la valeur de la variable d'environnement `USER` par défaut. Les deux processus fils doivent être les fils directs du processus père qui exécute la fonction `main`.

Exercice 8.5

Reprenez l'exercice 6.12 en y ajoutant les pipes.

Exercice 8.6

Reprenez l'exercice 8.1 en le séparant en deux programmes distincts et en utilisant un tube nommé. Le premier programme crée le tube avec `mkfifo`, et y place les données lues sur l'entrée standard. Le deuxième programme ouvre le tube, lit les données qui s'y trouvent et les affiche sur sa sortie standard.

Exercice 8.7

On désire mettre en place un système de communication interprocessus sous le contrôle d'un gérant de communication. Le gérant de communication reçoit des requêtes des processus par un (unique) tube nommé (appelé T). Le nom de ce tube est connu par tous les processus voulant communiquer.

Lorsqu'un processus P_i veut communiquer avec un autre, il crée un tube nommé (appelé T_i) qui lui est propre, puis envoie au gérant (par l'intermédiaire du tube T) un message spécifiant le nom de T_i. C'est l'*abonnement*.

Le gérant ouvre alors T_i, lui associe un numéro interne (≥ 0), et renvoie ce numéro interne au processus P_i par l'intermédiaire de T_i.

Par la suite, lorsque le processus P_i veut envoyer un message au processus P_j, le processus P_i écrit un message dans le tube T, le gérant lit ce message, l'analyse pour voir si c'est possible (i.e. si P_j est déjà abonné) et l'envoie à P_j par l'intermédiaire du tube T_j. Le gérant envoie un message P_i pour lui signaler si l'opération a réussi ou non.

Le processus P_i doit pouvoir obtenir la liste de tous les processus abonnés. Dans ce cas, il envoie un message dans T, le gérant répond alors par T_i en lui envoyant la liste.

Le processus P_i doit aussi pouvoir se désabonner.

1. Spécifiez le format des messages (c'est-à-dire le *protocole*) circulant dans T et dans les tubes T_i.
2. Programmez le gérant.

Exercice 8.8

On s'intéresse maintenant aux processus clients (les P_i). Ils doivent avoir une interface simplifiée (interface de programmation) pour dialoguer avec le gérant.

Cette interface suppose que toutes les données relatives à une connexion sont décrites par un type `comm` dont vous devez définir le contenu.

Les fonctions disponibles sont :

— `comm initialiser (char tube [])`
Cette fonction amorce une communication avec le gérant spécifié par le tube de nom `tube` (elle crée et ouvre le tube propre au processus, et elle le tube du gérant).
— `int liste (comm gerant, int abonnes [], int maxab)`
Cette fonction retourne dans le tableau `abonne` la liste des abonnes au gérant de communication spécifié par `gerant`. Le nombre maximum [4] de numéros pouvant être placés dans ce tableau est spécifié par `maxab`. Cette fonction retourne le nombre d'abonnés placés dans le tableau, ou -1 pour signifier une erreur.
— `int envoyer (comm gerant, int abonne, char *message, int lg)`
Cette fonction envoie à l'abonné `abonne`, par l'intermédiaire du gérant, un message spécifié par les `lg` octets inclus dans `message`. Le résultat est un code signifiant si l'opération s'est bien déroulée.
— `int recevoir (struct comm *comm, char *message, int *lgmax)`
Cette fonction attend un message. Le message (au plus `lgmax` octets) est placé dans la zone spécifiée par `message`. La longueur du message reçu est placée en retour dans `lgmax`. Le résultat de cette fonction est l'identificateur de l'émetteur, ou -1 si une erreur est intervenue.

On vous demande de :

1. définir le type `comm` ;
2. programmer une application simple à l'aide des fonctions décrites ci-dessus : à chaque fois qu'un processus s'abonne, il demande à tous les autres abonnés leur numéro de processus, et affiche le numéro minimum ;
3. programmer les fonctions ci-dessus.

4. S'il y a plus d'abonnés, leurs numéros sont perdus.

Thème 9 - Système : Signaux

Exercice 9.1

À l'aide des primitives V7, écrivez un programme qui incrémente et affiche un compteur à chaque fois qu'il reçoit le signal `SIGINT`. Au bout de 5 fois, il doit s'arrêter. L'incrémentation et l'affichage du compteur ne doivent pas être réalisés dans la fonction `main`.

Exercice 9.2

Activez la génération des fichiers « `core` »[5]. Faites un programme pour générer une erreur de segmentation. Par exemple, rédigez un programme avec un tableau global de 10 entiers et une fonction `main` qui appelle une fonction `a` qui appelle elle-même une fonction `b`. Cette dernière fonction écrit dans le tableau global à partir de l'indice 0, sans borne supérieure. N'oubliez pas de compiler avec l'option `-g`, et exécutez le programme.

Vérifiez qu'un fichier `core` a été créé. Utilisez `gdb` pour afficher l'endroit où le programme s'est arrêté (commandes `win`, `where`, `up`/`down` et `list`) et l'indice (command `print`).

Variation : interrompez un programme en utilisant le signal `SIGQUIT` (obtenu par défaut avec `^\` dans un shell) pour interrompre le processus et générer un fichier « `core` ». Utilisez `gdb` pour localiser l'endroit où vous avez interrompu le programme et la valeur des variables à ce moment.

Exercice 9.3

À l'aide des primitives V7, écrivez un programme qui attend l'arrivée d'un signal (n'importe lequel), affiche sa signification (par exemple : `illegal instruction` pour `SIGILL`) puis se termine. Vous pouvez utiliser la fonction de bibliothèque `psignal` pour afficher la signification d'un signal.

Exercice 9.4

Modifiez le programme de l'exercice précédent pour traiter plusieurs signaux consécutifs : votre programme ne doit pas se terminer après l'arrivée d'un signal.

Exercice 9.5

Écrivez un programme composé de deux processus. Le processus père génère un processus fils qui doit exécuter une fonction `traite` toutes les secondes, qui ne fait qu'afficher un message. Au bout d'une minute, le processus père affiche un message et prévient le fils qu'il doit s'arrêter en lui envoyant le signal `SIGUSR1`. Lorsque le processus fils reçoit l'ordre du père, il affiche un message et s'arrête effectivement, provoquant alors la terminaison du père. Vous utiliserez les primitives V7, avec le signal `SIGALRM` pour tout ce qui est temporisation.

Exercice 9.6

Écrivez un programme composé d'une boucle sans fin qui incrémente un compteur. Lorsque l'utilisateur appuie sur la touche d'interruption (signal `SIGINT`), le programme sauve la date en clair et la valeur courante du compteur dans un fichier, à la suite de ce qui s'y trouve déjà. Lorsque le signal `SIGTERM` est reçu, le programme écrit le mot `fin` à la fin du fichier, puis se termine. En dépit des règles de bon usage des signaux, vous grouperez les accès au fichier (ouverture, écriture, fermeture) dans les fonctions associées aux signaux. Vous utiliserez les primitives POSIX, en veillant à ce que les traitements des deux signaux n'interfèrent pas.

Exercice 9.7

Reprenez l'exercice précédent en appliquant cette fois-ci les règles de bon usage des signaux, c'est-à-dire en

5. Avec `bash`, utilisez la commande « `ulimit -c unlimited` ».

réalisant le minimum d'opérations dans les fonctions associées aux signaux.

Exercice 9.8

Écrivez un programme qui lance une commande chaque seconde en vous aidant du signal `SIGALRM`. Vous ferez en sorte d'éviter les processus « *zombies* », en vous aidant du signal `SIGCHLD`. Vous utiliserez les signaux POSIX.

Exercice 9.9

Écrivez un programme composé d'une boucle sans fin qui affiche le message « traitement normal » toutes les secondes. Lorsqu'il reçoit un signal `SIGINT`, il exécute une fonction qui affiche le message « traitement du signal de niveau 1 », puis attend une seconde, cinq fois de suite. Lorsqu'il reçoit une nouvelle fois le signal `SIGINT`, il augmente la valeur du niveau affiché, puis revient au niveau précédent au bout de ses cinq affichages. Vous pouvez afficher des tabulations correspondant au niveau de traitement courant pour plus de lisibilité. Vous utiliserez les primitives POSIX. On notera que le flag `SA_NODEFER` (primitive `sigaction`) permet de ne pas masquer le signal en cours de traitement.

Exercice 9.10

On désire simuler un mécanisme matériel comparable à un coupleur série à l'aide des signaux POSIX. Un 0 (zéro) est matérialisé par le signal `SIGUSR1`, un 1 (un) est matérialisé par le signal `SIGUSR2`. Un octet est transmis par une succession de 8 bits (0 ou 1). À chaque fois qu'il reçoit un bit, le récepteur doit envoyer en retour un acquittement (signal `SIGUSR1`) pour prévenir l'émetteur qu'il peut passer au bit suivant.

Écrivez les fonctions suivantes :

1. `void envoyer (pid_t recepteur, int octet)`
   Cette fonction envoie les 8 bits constituant un octet au processus désigné.
2. `void preparer_reception (void)`
   Cette fonction prépare le récepteur à recevoir un octet.
3. `int recevoir (pid_t emetteur)`
   Cette fonction attend que suffisamment de bits soient reçus pour constituer un octet, et renvoie alors la valeur reçue.

Rédigez un programme de test dans lequel le processus père envoie une valeur (passée en argument du programme) et le fils affiche la valeur reçue.

Exercice 9.11

En utilisant un protocole similaire à celui de l'exercice précédent, on désire simuler le fonctionnement d'un tube, c'est-à-dire réaliser les fonctions suivantes :

1. `void preparer_tube (void)`
   Cette fonction initialise les structures de données associées au tube.
2. `void processus_tube (pid_t autre, int sens)`
   Connaissant le *pid* de l'autre processus, cette fonction initialise l'accès au tube pour une lecture (sens = 1) ou pour une écriture (sens = 2).
3. `void fermer_tube (void)`
   Cette fonction ferme le tube.
4. `void ecrire_tube (void *buffer, int longueur)`
   Cette fonction écrit dans le tube les données spécifiées.
5. `int lire_tube (void *buffer, int longueur)`
   Cette fonction extrait du tube les données et les renvoie dans le buffer. Le nombre d'octets lus est renvoyé (0 en fin de tube).

Thème 10 - Système : Gestion mémoire

Exercice 10.1

Quelles sont, sur `turing`, les tailles des types suivants :

- `int`
- `char`
- `char *`
- `struct { char c ; char *pc ; int i ; }`
- `struct { char c ; char *pc ; int i ; } *`

Expliquez ce que vous constatez pour la taille de la structure.

Exercice 10.2

Soit une variable `v` de type `struct s` contenant, parmi d'autres, un champ `c`. Affichez l'adresse relative (en nombre d'octets) du champ c par rapport au début de la variable `v`.

Donnez à présent la définition d'une macro `ADRESSE_RELATIVE_DE(s,c)` permettant d'obtenir l'adresse du champ `c` d'une structure `s` relativement au début de la structure. L'adresse relative renvoyée par votre macro doit être exprimée en octets.

Utilisez cette macro pour afficher les adresses relatives des différents champs de la structure décrite dans l'exercice précédent.

Exercice 10.3

Pour détecter les corruptions mémoire, on propose d'utiliser la technique dite du « canari »[6] : à chaque fois qu'une zone mémoire est allouée, `malloc` initialise un espace avant et un espace après avec des valeurs convenues[7] (fixes). Lorsqu'on libère la zone, `free` vérifie que ces valeurs convenues sont toujours présentes. Si elles ne le sont pas, `free` signale l'erreur.

Écrivez les fonctions `mon_malloc` et `mon_free` qui fonctionnent comme décrit ci-dessus. Vos fonctions appelleront les vraies `malloc` et `free` pour allouer et libérer effectivement la mémoire.

Exercice 10.4

À l'aide de la primitive système `mmap`, écrivez un programme qui prend en argument un nom de fichier et affiche la première et la dernière ligne qui s'y trouvent.

Exercice 10.5

À l'aide de la primitive système `sbrk`, écrivez une version des fonctions de librairie `malloc` et `free`.

Votre version doit tenir à jour une liste de blocs mémoire, triée selon les adresses. Chaque bloc est soit alloué, soit libre. Cette liste est gérée selon l'algorithme du « *First Fit* », c'est-à-dire que lors d'une demande d'allocation, le premier emplacement libre suffisamment grand est choisi. S'il n'y a pas assez de mémoire, un nombre entier de pages de 4096 octets est demandé au système avec `sbrk`. Lorsqu'un emplacement est libéré avec `free`, on fusionne cet emplacement avec les emplacements adjacents s'ils étaient eux-mêmes libres.

Pour simplifier, on ne tiendra pas compte des contraintes d'alignement.

6. Autrefois, pour détecter les gaz toxiques dans les mines de charbon, les mineurs emmenaient avec eux un canari. Lorsque celui-ci, plus sensible, donnait des signes d'intoxication, les mineurs devaient quitter la mine au plus vite.
7. On prendra par exemple les 8 octets `0x01`, `0x23` ... `0xef`.

Thème 11 - Système : IPC System V

Exercice 11.1

Écrivez un programme composé de deux processus. Le père crée une file de messages anonyme, puis y envoie des messages composés de lignes lues sur l'entrée standard. Lorsque la fin de fichier est détectée, il envoie une chaîne vide, puis attend la terminaison du processus fils. Le processus fils reçoit les messages, puis comptabilise le nombre d'octets reçus.

Exercice 11.2

On désire traiter des requêtes fournies sous forme de messages, chaque requête étant composée d'une priorité (de 1 à 3, 1 étant la plus prioritaire) et d'une donnée de type `char []`.

La file de messages est identifiée par une clef formée à l'aide d'un nom de fichier (`toto` par exemple) et de la lettre E.

Écrivez le programme serveur qui crée la file de messages, puis traite les messages selon leur priorité. Chaque message est traité à l'aide d'une fonction `traite` que l'on supposera écrite. Le programme doit se terminer et effacer la file de messages lorsqu'il reçoit un message contenant une donnée de taille nulle.

Écrivez à présent le programme client qui reçoit zéro (pour envoyer un message de taille nulle) ou deux paramètres (priorité et chaîne de caractères représentant la donnée).

Exercice 11.3

On suppose qu'il existe une file de messages de clef c. On désire avoir un système qui permette à n processus d'émettre des messages à destination du processus i ($1 \leq i \leq n$).

Programmez les fonctions :

```
int ouvrir (key_t clef)
int envoyer (int msqid, int moi, int lui, void *message, size_t taille)
int recevoir (int msqid, int moi, int *lui, void *message, size_t *taille)
```

La fonction `ouvrir` doit permettre l'accès à la file. La fonction `envoyer` envoie, par l'intermédiaire de la file, un message à destination du processus `lui`, composé de `taille` octets situés à l'adresse `message`. La fonction `recevoir` attend un message de l'un quelconque des processus, place à l'adresse `lui` le numéro de l'émetteur, et place à l'adresse `message` le message reçu. Le paramètre `taille` précise la place maximum utilisable pour le message en entrée, et la place réellement utilisée en sortie.

Exercice 11.4

On désire communiquer des grandes quantités de données entre deux processus. Pour cela, on propose d'utiliser un segment de mémoire partagée de taille `MAX` octets. Lorsque l'émetteur désire envoyer n octets, il les place au début du segment, puis il envoie un message contenant n au récepteur. Lorsque le récepteur a fini de lire le message, il renvoie un message à l'émetteur pour signifier que la place est libre. Programmez les deux processus.

Exercice 11.5

Reprenez l'exercice précédent en utilisant un segment de mémoire partagée et un groupe de sémaphores : expliquez la méthode de synchronisation et programmez les deux processus.

Exercice 11.6

Écrivez un programme composé de deux processus : le premier initialise un segment de mémoire partagée et un groupe de deux sémaphores, crée le fils, puis lit des données entières dans le segment et les affiche. Le fils

calcule des données avec une fonction `calcul` (qui renvoie 0 lorsque toutes les données ont été calculées) et les transmet au fur et à mesure au père par l'intermédiaire du segment de mémoire partagée. Pour écrire une donnée, le fils fait un P sur le premier sémaphore, écrit la donnée, puis fait un V sur le deuxième sémaphore. De manière symétrique, le père fait un P sur le deuxième sémaphore, lit la donnée, puis fait un V sur le premier sémaphore.

Exercice 11.7

En vous basant sur l'exercice précédent, écrivez deux programmes. Le premier s'utilise de la manière suivante :

`producteur` *n*

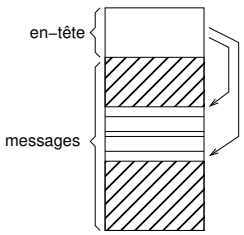
où *n* est une valeur entière. La valeur est placée dans le segment de mémoire partagée après la dernière valeur déjà produite. S'il n'y a plus d'emplacement libre, le programme attend qu'un emplacement se libère par le deuxième programme.

Le deuxième programme, `consommateur`, ne prend aucun argument et renvoie la première valeur stockée dans le segment de mémoire partagée. Si aucune valeur n'est disponible, le programme attend qu'un producteur y stocke une valeur.

Vous noterez que l'indice de la première valeur stockée doit être partagé par tous les consommateurs. De même, l'indice du premier emplacement libre doit être partagé par tous les producteurs. Ces deux indices doivent donc être placés dans le segment de mémoire partagée, et constituent des variables critiques qui doivent être protégées par des sémaphores d'exclusion mutuelle.

Exercice 11.8

On souhaite implémenter un mécanisme comparable aux files de messages à l'aide de la mémoire partagée et un groupe de trois sémaphores. Le segment de mémoire partagée contient (voir figure) un en-tête, puis les messages eux-mêmes. La zone des messages est gérée comme un buffer circulaire : lorsqu'un message ne peut être mis en entier à la fin du segment, sa suite est placée au début de la zone des messages. L'en-tête contient la taille de la zone des messages, les références au premier message et au dernier message.



Par convention, le groupe de sémaphores a une clef égale à la clef du segment de mémoire partagée + 1. Il contient :

- un sémaphore représentant la place disponible dans le segment : il est initialisé à la taille de la zone des messages en nombre d'octets ;
- un deuxième sémaphore pour réaliser l'exclusion mutuelle lors de la manipulation des structures de données : il est initialisé à 1 ;
- un troisième sémaphore pour représenter le nombre de messages dans la file : il est initialisé à 0 ;

Écrivez les fonctions :

- `int initialiser_file (key_t clef)`
  cette fonction, appelée une seule fois, crée et initialise un segment de mémoire partagée pour 10000 octets de messages, puis crée le groupe de sémaphores et l'initialise et retourne -1 en cas d'erreur ;
- `int envoyer (key_t clef, const char *message)`
  cette fonction envoie le message spécifié, et retourne -1 en cas d'erreur ;
- `int recevoir (key_t clef, char *message)`
  cette fonction attend un message, le recopie dans la zone spécifiée par l'adresse fournie en paramètre (que l'on suppose déjà allouée), et retourne -1 en cas d'erreur.

Thème 12 - Threads POSIX

Exercice 12.1

Que fait le programme ci-après ?

```
#define MAX     10

void *fonction (void *arg)
{
    printf ("%d\n", * (int *) arg) ;
    return NULL ;
}

void erreur (char *msg)
{
    perror (msg) ; exit (1) ;
}

int main (int argc, char *argv [])
{
    pthread_t tid [MAX] ; int i ;
    for (i = 0 ; i < MAX ; i++)
        if (pthread_create (&tid [i], NULL, &fonction, (void *) &i) == -1)
            erreur ("pthread_create") ;
    for (i = 0 ; i < MAX ; i++)
        if (pthread_join (tid [i], NULL) == -1)
            erreur ("pthread_join") ;
}
```

Exercice 12.2

Soit un programme qui prend en argument 2 valeurs n et p et démarre n threads. Chaque thread reçoit les valeurs n et p ainsi que le numéro $t \in [1, n]$ du thread et calcule $u_t = \sum_{i=1}^{p}((t-1)p + i)$. Le thread principal affiche la somme des valeurs u_t calculées.

Quelle est la valeur calculée par ce programme ? Rédigez-le.

Exercice 12.3

Écrivez un programme composé de n nouveaux threads. Chacun de ces threads incrémente un compteur qui lui est propre et s'arrête à la valeur 10 en affichant un message. À chaque incrément, le thread doit attendre un temps aléatoire (vous utiliserez `nanosleep` et vous limiterez la durée à une seconde) et afficherez la valeur du compteur préfixée par le numéro du thread ($\in [1, n]$). Le programme ne doit se terminer que lorsque les n threads ont terminé, et doit afficher le numéro du thread qui s'est terminé en dernier.

Exercice 12.4

Écrivez un programme composé de 4 nouveaux threads. Chaque thread incrémente un million de fois une variable globale `compteur` de type `long int`. Une fois les threads terminés, affichez la valeur du compteur.

Que constatez-vous ? Que proposez-vous pour résoudre le problème ?

Exercice 12.5

Expliquez, pour chacune des deux portions de code suivantes, pourquoi elle ne peut pas être exécutée de manière fiable par deux threads.

```
struct listelem *head ;

void insert_list (struct listelem *e)
{
    e->next = head ;
    head = e ;
}
```

```
double compte_en_banque ;

void recevoir (double montant)
{
    compte_en_banque += montant ;
}
```

Exercice 12.6

On désire réaliser des sémaphores à utiliser avec les threads POSIX. Pour cela, on définit les fonctions suivantes :

```
semaphore_t semaphore_create (int val) ;
void semaphore_P (semaphore_t sem) ;
void semaphore_V (semaphore_t sem) ;
```

Donnez la définition du type `semaphore_t`, et rédigez ces fonctions en utilisant les mutex et les variables de condition.

Thème 13 - Réseau : Sockets

Exercice 13.1

Écrivez un programme qui prend en paramètre un nom d'hôte (tel que `www.unistra.fr` par exemple), une adresse IPv4 ou une adresse IPv6. Pour chaque adresse trouvée par la fonction `getaddrinfo`, votre programme doit afficher la famille d'adresses (IPv4 ou IPv6) et l'adresse avec la fonction `inet_ntop`.

Exercice 13.2

Modifiez votre programme précédent pour ne sélectionner qu'une famille d'adresses (IPv4 ou IPv6).

Exercice 13.3

Écrivez un client UDP pour le protocole Echo (service « `echo/udp` »).

Écrivez un serveur UDP pour le protocole Echo (service « `echo/udp` »).

Exercice 13.4

Même exercice que précédemment en mode TCP.

Exercice 13.5

Écrivez à l'aide du protocole TCP un client et le serveur qui échangent des valeurs. On adoptera le protocole décrit par la structure suivante :

```
struct proto {
    uint16_t nint ;           // nombre de valeurs émises
    int32_t valeurs [MAX] ;   // les valeurs elles-mêmes
} ;
```

Le client accepte comme premier argument un nom d'hôte ou une adresse IP (v4 ou v6), et comme arguments suivants la liste des valeurs à transmettre. Les arguments suivants seront les valeurs. Pour chaque connexion de client, le serveur doit lire la liste de valeurs, et renvoyer au client les valeurs élevées au carré.

Les données doivent être échangées, même si les deux ordinateurs ont des formats de stockage des entiers différents.

Exercice 13.6

Modifiez le programme de l'exercice précédent : si la connexion vient d'une adresse IPv4, le serveur renvoie les nombres élevés à la puissance 4. Si la connexion vient d'une adresse IPv6, le serveur renvoie les nombres élevés à la puissance 6.

Exercice 13.7

Programmez un système de discussion multiutilisateur (*chat*). Chaque client se connecte sur le serveur. Toute ligne saisie par l'utilisateur est envoyée par le client vers le serveur. Toute ligne reçue depuis le serveur est affichée à l'écran. Lorsque le serveur reçoit une ligne d'un client, il la retransmet à tous les autres clients.

Exercice 13.8

L'objet de cet exercice est de rédiger un programme pour distribuer des jetons à des clients IP. On représente un jeton par un entier de 1 à n. Le serveur accepte des connexions des clients.

Si le client demande un jeton, le serveur lui attribue le premier jeton disponible et mémorise l'adresse IP du client et la date d'attribution. Ces informations (jetons, adresse IP associée et date d'obtention) doivent être mémorisées dans un segment de mémoire partagée.

Si un client demande à qui est attribué un jeton, le serveur répond avec l'adresse IP du client, ou un code en cas de jeton non associé.

Enfin, un jeton est considéré comme abandonné au bout d'un certain délai. En conséquence, un client peut demander à rafraîchir l'association d'un jeton avant l'expiration du délai. Pour ce faire, il envoie un message au serveur avec le numéro du jeton à rafraîchir. Si l'adresse IP concorde, le serveur doit réinitialiser la date d'obtention du client.

1. spécifiez le protocole que le serveur et les clients doivent utiliser, de telle sorte que vous puissiez utiliser le programme `telnet` comme client primitif ;
2. rédigez le serveur, qui doit prendre en paramètre le nombre de jetons initialement disponibles ;
3. rédigez le client d'obtention de jeton ;
4. rédigez le client d'interrogation de jeton ;
5. rédigez le client de rafraîchissement de jeton.