

# Programmation système / Unix

Franck Pommereau

Licence 2, Université Paris-Saclay / Évry—Val d’Essonne

## ➤ Consignes

Les exercices signalés par sont à préparer individuellement à l'avance et à remettre en ligne sur <https://badass.ibisc.univ-evry.fr> avant la séance (sauf bien sûr pour la première). Ces exercices seront pris en compte pour la note du cours. Votre chargé·e de TD vous donnera l'échéance exacte pour remettre vos travaux chaque semaine. *Un outil de détection de plagiats sera utilisé pour vérifier que vous fournissez des solutions personnelles.* Enfin, les exercices signalés par sont à réaliser pendant la séance, ils peuvent avoir été préparés mais n'ont pas à être remis en ligne. L'énoncé contient plus d'exercices qu'on ne peut en faire ou corriger pendant les séances, mais cela vous donne de quoi approfondir et vous entraîner. La difficulté des exercices est indiquée par des étoiles :

- ★☆☆ Exercice facile qui devrait être résolu directement.
- ★☆☆ Exercice qui nécessite une réflexion préalable.
- ★★★ Exercice long ou qui demande une réflexion approfondie.

Enfin, du texte signalé par donne les objectifs d'un exercice, du texte signalé par donne des indications pour le résoudre, et du texte signalé par propose du travail supplémentaire pour celles et ceux qui désirent approfondir.

## ➤ Séance 1 : prise en main des outils, révisions de C

Cette séance permet de se familiariser avec les outils CoW et **badass**, les exercices sont notés car ils sont réalisés en séance. Mais vous devez les soumettre en ligne afin de vérifier le fonctionnement de votre compte **badass** et votre compréhension de ses diagnostics.

CoW est un éditeur de C en ligne, qui permet une compilation sous GNU/Linux avec GCC. Il a des fonctionnalités assez minimales mais permet de préparer les TD dans un environnement où ils pourront fonctionner. Quelques informations bonnes à connaître :

- vous devez autoriser les *popups* dans votre navigateur car la fenêtre de compilation/exécution s'ouvre dans un *popup*
- pour ajouter des options de compilation à GCC, insérez dans votre code source des commentaires de la forme `// gcc: --mon-option`
- pour ajouter des options à l'édition de liens de GCC, insérez dans votre code source des commentaires de la forme `// ldd: -lmabibli`
- pour modifier les variables d'environnement avant la compilation et l'exécution de votre code source, insérez des commentaire :
  - `// env: NOM=VALEUR` pour ajouter une variable **NOM**
  - `// env: NOM=` pour supprimer une variable **NOM**
- pour ouvrir un *shell* au lieu de compiler et d'exécuter votre programme, insérez dans votre code source un commentaire `// run: bash` ce qui lancera **bash** au lieu de **make**. Depuis ce shell, vous pouvez lancer **make** pour compiler et lancer votre programme, ou utiliser d'autres commandes.

### Exercice 1 (★☆☆)

- Prendre en main des outils et de l'environnement de travail.
- La plateforme de soumission en ligne est **badass** : <https://badass.ibisc.univ-evry.fr>  
Vous devrez y créer un compte, puis vous y connecter avec le mot de passe fourni lors de l'inscription.
- La plateforme de programmation est CoW : <https://cow.ibisc.univ-evry.fr>  
Prenez soin de télécharger vos travaux car rien n'y est sauvegardé.
- Programmez chacune des questions dans CoW puis soumettez-la sur **badass**.
- Les extraits de code dont vous avez besoin pour certaines exercices sont fournis sur eCampus, avec l'énoncé du TP.

1. Reprenez le programme `hello.c` vu en cours, compilez-le, et exécutez-le.
2. Que se passe-t-il si vous supprimez la ligne commençant par `#include` ? Pourquoi ?

3. Compilez et exécutez le programme constitué des fichiers `hello-read.c`, `hello-print.c`, `hello.h`, et `hello-main.c` vus en cours.

### Exercice 2 (★★★)

- Se (re)mettre à la programmation en C de base : boucles, tests, tableaux statiques, etc.
- On va écrire un puissance 4 en C. Le jeu se joue à deux, chaque joueur dépose un pion dans une colonne, ce pion tombe au bas de la colonne ou sur le dernier pion posé auparavant. Le gagnant est le premier joueur qui parvient à aligner quatre pions dans n'importe quelle direction, y compris les diagonales.
- Respectez bien les prototypes des fonctions, et leur fonctionnement tel qu'il est décrit dans l'énoncé. Cela permettra à badass de les tester, et cela vous facilitera la construction du programme complet.

Pour l'affichage, on utilisera la bibliothèque `ncurses`, classique sous Unix, qui permet d'utiliser le terminal comme une matrice de caractères et d'y faire des dessins à base de texte. Par exemple, notre jeu s'affichera comme représenté sur la figure 1. Sur cette figure, on donne aussi le code source en C permettant d'inclure la bibliothèque `ncurses`, ainsi qu'une fonction `INIT_SCREEN()` pour initialiser le terminal, et une autre `DONE_SCREEN()` pour le rétablir le terminal à son fonctionnement normal et quitter le jeu.

- Pour compiler sous CoW avec la bibliothèque `ncurses`, prenez soin d'ajouter le commentaire `// ldd: -lncurses` comme dans le code source fourni.

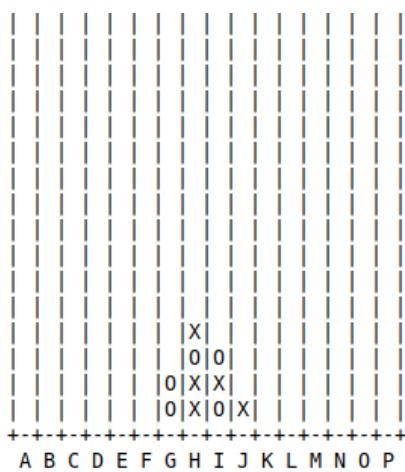
La première fonction se charge aussi d'initialiser le générateur de nombres pseudoaléatoires qu'on utilisera pas la suite pour choisir une case au hasard. On dispose alors des fonctions suivantes :

- `move(0,0)` ⇒ déplace le curseur dans le coin supérieur gauche du terminal ;
- `printw("...", ...)` ⇒ version de `printf` compatible avec `ncurses` ;
- `refresh()` ⇒ affiche sur le terminal les résultats des appels à `printw` ;
- `getch()` ⇒ lit une touche du clavier, y compris les touches de direction, et renvoie soit la valeur du `char` correspondant à la touche, soit un code spécial défini par une constante, comme `KEY_BACKSPACE` que nous utiliserons.

Toutes ces fonctions nous serviront pour gérer le jeu. Dans le code source fourni sur la figure 1, on trouve aussi la déclaration de la structure pour stocker la grille de jeu. Notez que la taille de la grille est paramétrée par `SIZE` et que tout votre programme doit être capable de gérer une valeur de `SIZE` entre 4 et 26. Il s'agit d'un simple tableau de `char` à deux dimensions qui contiendra soit des espaces (caractère ' ') pour une case vide, soit 'X' ou 'O' pour un pion d'un joueur. Notez aussi les déclarations de `NB_PLAYERS` et `CHIP` qui permettent en théorie d'avoir plus de deux joueurs (et donc d'autres caractères pour les jetons). Il serait bien que votre programme soit capable de fonctionner quel que soit le nombre de joueurs. Enfin, même si vous voudrez utiliser vos propres versions de `main` au cours du développement afin de tester vos fonctions intermédiaires, veillez à utiliser celle qui vous est donnée dans la version finale.

1. Écrivez une fonction `void init_board(void)` qui initialise la grille de jeu en la vidant entièrement.
2. Écrivez une fonction `void draw_board (void)` qui affiche la grille de jeu comme présentée sur la figure 1. La ligne du bas doit correspondre à `board[0]`.
3. Écrivez une fonction `int get_col (void)` qui lit des touches au clavier jusqu'à ce qu'on tape soit :
  - `backspace`, et alors la fonction renvoie -1 ;
  - une lettre de colonne valide, et alors la fonction renvoie le numéro de colonne correspondant ;
  - toutes les autres touches sont lues mais ignorées.
4. Écrivez une fonction `int add_coin (int col, int player)` qui ajoute un pion pour le joueur `player` (qui vaut 0 ou 1) dans la colonne `col`, et renvoie le numéro du prochain joueur à jouer :
  - `player` si la colonne est pleine (il recommence) ;
  - le numéro de l'autre joueur si le pion a pu être ajouté.
5. Écrivez une fonction `int game_over (void)` qui renvoie :
  - 0 si personne n'a gagné et que la grille n'est pas pleine ;
  - 1 si personne n'a gagné et que la grille est pleine (le jeu est terminé sans gagnant) ;
  - sinon, le code ASCII du caractère correspondant au gagnant.

De façon contre-intuitive, cette fonction sera plus simple si vous la pensez pour qu'elle fonctionne avec n'importe quel nombre de joueurs (et donc n'importe quel caractère pour leurs jetons).
6. Écrivez une fonction `void play (void)` pour dérouler le jeu :
  - tant qu'il n'y a pas de gagnant ou que la grille n'est pas pleine :
    - le joueur dont c'est le tour dépose un pion en tapant la lettre de la colonne,
    - s'il choisit une colonne pleine, rien ne se passe et c'est à nouveau son tour,
    - sinon, on passe au joueur suivant ;
    - on affiche le résultat de la partie.



```

 1 // ldd: -lncurses
 2 #include <ncurses.h>
 3 #include <time.h>
 4 #include <stdlib.h>
 5
 6 void INIT_SCREEN (void) {
 7     initscr();
 8     raw();
 9     keypad(stdscr, TRUE);
10     noecho();
11 }
12
13 void DONE_SCREEN (void) {
14     endwin();
15     exit(0);
16 }
17
18 #define PLAYERS 2
19 char CHIP[PLAYERS] = "XO";
20
21 #define SIZE 16
22
23 char board[SIZE][SIZE];

```

\_\_\_\_\_ p4.c \_\_\_\_\_

```

25 int main (void) {
26     INIT_SCREEN();
27     init_board();
28     play();
29     getch();
30     DONE_SCREEN();
31 }

```

\_\_\_\_\_ p4.c \_\_\_\_\_

➤ **Figure 1.** À gauche : la grille de puissance 4 en  $16 \times 16$ . (Le rectangle blanc à droite du texte “player 1?” est le curseur clignotant du terminal.) À droite : le code C à intégrer à votre programme.

À vous de voir quand il convient d'afficher la grille.

- ⊕ Étendez le jeu comme suggéré en permettant d'avoir plusieurs joueurs. Le plus difficile est peut-être de trouver des caractères assez différents pour représenter les pions de façon lisible.
- ⊕ Permettez à un ou plusieurs joueurs d'être joués par l'ordinateur. Pour faire jouer l'ordinateur au hasard, utilisez `random` [3](#) pour générer des tirages pseudo-aléatoires.
- ⊕ Programmez une stratégie pour que l'ordinateur essaie de gagner, par exemple en cherchant à poser des pions qui créent des alignements, ou en cherchant à bloquer les alignements des autres joueurs.
- ⊕ Pour programmer un ordinateur très fort, regardez de côté de l'algorithme *minmax* qui permet de calculer des stratégies optimales. Afin de moduler la force de l'ordinateur, on a alors deux moyens : (1) limiter la profondeur d'exploration de l'algorithme (ce qui est de toute façon préférable pour une exécution rapide), l'ordinateur ne fait alors pas d'erreur mais il n'a qu'une vision limitée à quelques coups en avant ; (2) introduire quelques mauvais coups, par exemple, avec une certaine probabilité, chaque coup est joué au hasard au lieu d'être calculé par *minmax* (plus cette probabilité est basse, plus l'ordinateur est fort).

## ➤ Séance 2 : révisions de C (suite)

### 🏠 Exercice 3 (★★★)

- Approfondir les révisions de C : tableaux dynamiques sur le tas, pointeurs, gestion de la mémoire.
- Réviser la numération dans les différentes bases (Unix utilise souvent l'octal).
- Utilisez les chiffres suivants : "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz+/" pour représenter les nombres dans toutes les bases demandées.
- Attention pour l'addition, il ne faut pas repasser par un entier C car notre type peut représenter des nombres arbitrairement grands dont les valeurs ne pourront pas être représentées par un entier C.
- Essayez de ne pas stocker de chiffres inutiles (les zéros non significatifs) et d'économiser la mémoire en général. Vous aurez sûrement besoin de `realloc` [3](#).

1. Définissez un type `number` pour représenter des nombres entiers positifs ou nuls en base quelconque parmi 2, 4, 8, 16, 32, et 64, sans limite sur la taille du nombre.
2. Écrivez une fonction `void free_number(number nbr)` qui libère la mémoire occupée par une valeur de type `number`. Si aucune mémoire n'a été allouée, votre fonction ne fera rien.
3. Écrivez une fonction `number to_number (unsigned int nbr, unsigned char base)` qui convertit un entier C `nbr` en nombre écrit en base `base`. Si la base fournie est incorrecte, elle est remplacée par 16.
4. Écrire une fonction `unsigned int to_uint (number nbr)` qui convertit un nombre en base quelconque en entier C.

5. Écrire une fonction `char* to_string (number nbr)` qui renvoie la chaîne de caractères représentant le nombre dans sa base.

### ▀▀ Exercice 4 (★★★☆)

Suite de l'exercice 3.

➊ En général, on ne peut pas directement convertir d'une base à une autre, il faut passer par le binaire. En revanche, comme nos bases sont toutes des puissances de deux, on peut convertir chiffre par chiffre sans passer par la représentation numérique (qui de toute façon ne pourra pas stocker des nombres arbitrairement grands).

1. Écrire une fonction `number from_string (char *str, unsigned char base)` qui convertit une chaîne en nombre dans la base indiquée.
2. Écrire une fonction `number to_base (number nbr, unsigned char base)` qui convertit un nombre d'une base quelconque vers la base `base`.
3. Écrire une fonction `number add_number (number a, number b)` qui additionne deux nombres en base quelconque et renvoie le résultat dans la base de `a`.
4. Écrire une fonction `int cmp_number (number a, number b)` qui renvoie -1 si `a < b`, 0 si `a == b`, et +1 si `a > b`.

⊕ Ajoutez la représentation des entiers signés codés en complément à 2.

⊕ Programmez la soustraction, la multiplication, la division.

### ⌂ Exercice 5 (★★★☆)

Manipuler des structures et des champs de bits.

➋ La fonction `time`  renvoie le nombre de secondes écoulées depuis l'epoch (c'est-à-dire le 1er janvier 1970 à 0h00 temps universel). Appelez `time(NULL)` pour récupérer ce nombre. Cette fonction renvoie une valeur de type `time_t` qui peut être négative, mais dans cet exercice, on considérera qu'on n'a jamais une valeur négative.

➌ Le 1er janvier 1970 était un jeudi.

➍ Une année est bissextile si elle est divisible par 4 et non divisible par 100, ou si elle est divisible par 400.

➎ La fonction `localtime`  permet d'obtenir les éléments de la date demandés dans cet exercice, ainsi que d'autres informations. Vous pouvez l'utiliser pour tester votre solution, mais la réponse finale ne doit pas faire appel à `localtime`.

On reprend le type `date` vu en cours :

---

```
date.c
4  typedef enum {MON = 1, TUE = 2, WED = 3, THU = 4, FRI = 5, SAT = 6, SUN = 7} dayname;
5
6  typedef enum {JAN = 1, FEB = 2, MAR = 3, APR = 4, MAY = 5, JUN = 6, JUL = 7,
7      AUG = 8, SEP = 9, OCT = 10, NOV = 11, DEC = 12} monthname;
8
9  typedef struct {
10     dayname weekday : 3;
11     unsigned int day : 5;
12     monthname month : 4;
13     int year : 20;
14 } date;
```

---

1. Écrivez une fonction `char* dayname_str (dayname day)` renvoyant le nom du jour qu'on lui passe en paramètre.
2. Écrivez une fonction `char* monthname_str (monthname month)` renvoyant le nom du mois qu'on lui passe en paramètre.
3. Écrivez une fonction `dayname weekday (time_t when)` renvoyant le jour de la semaine correspondant à la date exprimée en secondes depuis l'epoch.
4. Écrivez une fonction `int leapyear (unsigned int year)` indiquant si une année passée en paramètre est bissextile.
5. Écrivez une fonction `date from_time (time_t when)` qui calcule la date correspondant à un nombre de secondes depuis l'epoch.
6. Écrivez une fonction `main` qui affiche la date du jour.

⊕ Prenez en compte le décalage par rapport au temps universel dû au fuseau horaire.

⊕ Prenez en compte les valeurs négatives de `time_t`.

### Exercice 6 (★★★)

Manipuler des structures et des types unions. Découvrir le format d'images XPM, traditionnel sous Unix.

Vous aurez besoin de `fmod` [🔗3](#) et `fabs` [🔗3](#) qui sont définies dans `math.h`, et du coup il faudra utiliser l'option `// ldd: -lm` dans CoW, ou `-lm` si vous lancez vous-même GCC.

On se donne les déclarations de la figure 2 pour définir une structure `pixel` permettant de représenter une couleur en mode RGB (*red/green/blue*) ou HSL (*hue/saturation/lightness*) :

Un pixel RGB est représenté par trois valeurs entre 0 et 255 correspondant aux quantités de lumière rouge, verte, ou bleue à mélanger. C'est la représentation la mieux adaptée pour afficher sur un écran qui fait exactement ce mélange de couleurs. Un pixel HSL est représenté par trois flottants  $0 \leq H < 360$  (teinte),  $0 \leq S \leq 1$  (saturation), et  $0 \leq L \leq 1$  (luminosité). C'est une représentation plus adaptée pour manipuler les couleurs sur trois axes indépendants. La figure 3 donne les formules pour passer d'une représentation à l'autre.

- Écrivez une fonction `pixel copy_pixel (pixel pix)` renvoyant la copie d'une structure `pixel` passée en paramètre.
- Écrivez une fonction `pixel to_hsl (pixel pix)` qui renvoie un `pixel` converti en HSL (ou une copie de `pix` s'il est déjà en HSL).
- Écrivez une fonction `pixel to_rgb (pixel pix)` qui renvoie un `pixel` converti en RGB (ou une copie de `pix` s'il est déjà en RGB).
- Écrivez une fonction `void gradient (pixel start, pixel stop, pixel *tab, unsigned int len)` qui remplit le tableau `pixel tab[len]` d'un dégradé de couleurs démarrant à `start` et s'arrêtant à `stop`. Les couleurs dans le tableau doivent avoir le même mode que `start`. Pour obtenir un tel dégradé, on interpole sur les composantes des versions HSL de `start` et `stop`. Si  $(H_0, S_0, L_0)$  et  $(H_k, S_k, L_k)$  sont ces deux couleurs, le principe est de calculer des couleurs intermédiaires, pour

```
colors.c
1 // ldd: -lm
2
3 typedef enum {RGB, HSL} colormode;
4
5 typedef union {
6     colormode mode;
7     struct {
8         colormode mode;
9         unsigned char r, g, b;
10    } rgb;
11    struct {
12        colormode mode;
13        double h, s, l;
14    } hsl;
15 }
```

---

```
xpm.xpm
1 /* XPM */
2 static char *image[] = {
3     /* columns rows colors chars-per-pixel */
4     "8 4 2 1",
5     "X c #FF0000",
6     "O c #0000FF",
7     /* pixels */
8     "XOXOXOXO",
9     "OXOXOXOX",
10    "XOXOXOXO",
11    "OXOXOXOX"
12};
```

➤ Figure 2. À gauche : le code source de la structure `pixel`. À droite : la structure d'un fichier XPM.

$$(R, G, B) \rightarrow (H, S, L)$$

$$r = R/255$$

$$g = G/255$$

$$b = B/255$$

$$c_{\max} = \max(r, g, b)$$

$$c_{\min} = \min(r, g, b)$$

$$\Delta = c_{\max} - c_{\min}$$

$$H = \begin{cases} 0 & \text{si } \Delta = 0 \\ 60 \times (((g - b)/\Delta) \bmod 6) & \text{si } c_{\max} = r \\ 60 \times (2 + (b - r)/\Delta) & \text{si } c_{\max} = g \\ 60 \times (4 + (r - g)/\Delta) & \text{si } c_{\max} = b \end{cases}$$

$$S = \begin{cases} 0 & \text{si } \Delta = 0 \\ \Delta/(1 - |2 \times L - 1|) & \text{si } \Delta \neq 0 \end{cases}$$

$$L = (c_{\max} + c_{\min})/2$$

$$(H, S, L) \rightarrow (R, G, B)$$

$$c = (1 - |2 \times L - 1|) \times S$$

$$x = c \times (1 - |((H/60) \bmod 2) - 1|)$$

$$m = L - c/2$$

$$(r, g, b) = \begin{cases} (c, x, 0) & \text{si } 0 \leq H < 60 \\ (x, c, 0) & \text{si } 60 \leq H < 120 \\ (0, c, x) & \text{si } 120 \leq H < 180 \\ (0, x, c) & \text{si } 180 \leq H < 240 \\ (x, 0, c) & \text{si } 240 \leq H < 300 \\ (c, 0, x) & \text{si } 300 \leq H < 360 \end{cases}$$

$$R = (r + m) \times 255$$

$$G = (g + m) \times 255$$

$$B = (b + m) \times 255$$

➤ Figure 3. Conversions entre RGB et HSL.

$0 \leq i \leq k$  (notez que pour  $i = 0$  et  $i = k$  on obtient bien `start` et `stop`) :

$$\begin{aligned} H_i &= (H_0 \times (k - i) + H_k \times i)/k \\ S_i &= (S_0 \times (k - i) + S_k \times i)/k \\ L_i &= (L_0 \times (k - i) + L_k \times i)/k \end{aligned}$$

les pixels HSL de composantes ( $H_i, S_i, L_i$ ) formant alors un dégradé progressif de  $k + 1$  couleurs.

❸ Attention, on veut  $k + 1 = \text{len}$ .

### ▀▀ Exercice 7 (★★★☆)

☒ Fin de l'exercice précédent.

Écrivez une fonction `main` qui :

- calcule un dégradé de 64 couleurs en tout, entre deux couleurs de votre choix ;
- affiche le texte d'une image de  $64 \times 64$  pixels au format XPM correspondant à ce dégradé, présenté horizontalement.

Le format XPM est décrit sur la figure 2, il s'agit de code source C dont correspondant à un tableau de chaînes :

- la première chaîne contient 4 entiers séparés par des espaces indiquant respectivement :
  - le nombre  $w$  de colonnes de l'image,
  - le nombre  $h$  de lignes,
  - le nombre  $c$  de couleurs,
  - le nombre  $p$  de caractères utilisés pour représenter un pixel ;
- viennent ensuite  $c$  lignes décrivant les couleurs, chacune sous la forme de trois parties séparées par des espaces :
  - $p$  caractères représentant le code d'un pixel de cette couleur,
  - le caractère '**c**',
  - la couleur codée sous la forme **#RRGGBB** en hexadécimal, comme en HTML ;
- enfin, on a les  $h$  chaînes codant les lignes de pixels, chacune étant composée de  $w$  codes de pixels de  $p$  caractères chacun tels que définis plus tôt ;
- notez aussi qu'on a coutume d'ajouter des commentaires comme sur la figure.

Vous pourrez copier-coller la sortie de votre programme et l'enregistrer dans un fichier `gradient.xpm` pour l'ouvrir dans un éditeur d'image.

❸ Si votre système n'a pas de quoi afficher les fichiers XPM, essayez le site <https://www.aconvert.com/image/>.

⊕ Générez une image avec dégradé horizontal.

⊕ Ajoutez un `#define SIZE 64` à votre programme et générez un dégradé de `SIZE` couleurs et une image de `SIZE` pixels de côté pour toutes les valeurs de `SIZE > 0`.

## ► Séance 3 : ligne de commande, environnement, entrées/sorties

### ⌂ Exercice 8 (★★★☆)

☒ Lire la ligne de commande et les variables d'environnement.

❸ Si vous n'avez pas une version qui fonctionne de l'exercice 2, écrivez juste une fonction `main` qui affiche les paramètres à utiliser, en les récupérant comme décrit dans l'énoncé.

Reprenez le programme de l'exercice 2 et modifiez-le pour qu'il accepte des paramètres sur la ligne de commande permettant de définir :

- la largeur de la grille entre 4 et 26 ;
- la hauteur de la grille entre 4 et 16 ;
- le nombre de joueurs entre 2 et 8 (si votre programme autorise plus de deux joueurs).

S'il manque des paramètres, ou s'ils sont hors limites, utilisez des valeurs prises dans des variables d'environnement :

- `P4WIDTH` pour la largeur de la grille ;
- `P4HEIGHT` pour la hauteur de la grille ;
- `P4PLAYERS` pour le nombre de joueurs.

Si des variables ne sont pas définies ou contiennent des valeurs hors limite, utilisez des valeurs par défaut.

### ▀▀ Exercice 9 (★★★)

- Lire la ligne de commande de façon flexible, accéder au variables d'environnement.
  - ② Attention, la ligne de commande ne contient que des chaînes, vous devrez les convertir en nombres en utilisant `strtol` [🔗3](#). Écrivez une fonction intermédiaire pour gérer la conversion des entiers.
  - ③ Utilisez une pile dans un tableau de taille fixée à 128. Écrivez des fonctions intermédiaires pour gérer la pile.
  - ④ Le principe de calcul utilisé ici s'appelle la *notation polonaise inverse* : on spécifie d'abord les opérandes, puis les opérateurs, et on n'a jamais besoin de parenthèses. Par exemple, "9 3 -" vaut 6, et "9 3 - 2 +" vaut 8.
- Écrivez un programme qui lit des entiers ou des opérateurs (un caractère de "+-\*/\*") sur la ligne de commande et les traite ainsi :

- si l'argument est un nombre, il est mis dans une pile ;
- si l'argument est un opérateur •, on dépile  $b$ , on dépile  $a$ , et on empile  $a \bullet b$ .

Si une variable d'environnement BASE existe et que c'est un entier entre 2 et 16, les nombres lus sur la ligne de commande doivent être interprétés dans la base ainsi indiquée (elle-même toujours codée en base 10). Le programme traite la ligne de commande puis dépile le résultat final et l'affiche (en base 10).

Le programme doit gérer les erreurs en terminant avec des codes de retour spécifiques :

- 255 : s'il reste des valeurs sur la pile après l'affichage du résultat.
- 254 : si la pile s'avère trop petite en cours de calcul.
- 253 : s'il manque un argument sur la pile.
- 252 : si l'utilisateur demande une division par zéro.
- 251 : si un entier n'est pas correct.
- 250 : si la base n'est pas entre 2 et 16.

En l'absence d'erreur, le code de retour est EXIT\_SUCCESS.

- + Utilisez une pile de taille dynamique, soit avec `realloc` [🔗3](#) pour gérer son agrandissement par paliers, soit avec une structure de type liste chaînée.

### ▀▀ Exercice 10 (★★★)

- Manipuler la ligne de commande et les variables d'environnement, et travailler avec les pointeurs.
- ② La fonction `system` [🔗3](#) permet de lancer un autre programme depuis son programme C. Elle a beaucoup d'inconvénients mais pour les besoins de l'exercice elle sera bien utile.
- ③ Vous aurez sûrement l'usage des fonctions `index` [🔗3](#) et `memcpy` [🔗3](#) (ou `strncpy` [🔗3](#)).

Écrivez un programme qui lit sa ligne de commande et traite ses paramètres de la façon suivante :

- si c'est une chaîne de la forme `NOM=VALEUR`, il positionne la variable d'environnement `NOM` à `VALEUR`, en la remplaçant si elle existe ;
- si c'est une chaîne de la forme `NOM=`, il supprime la variable d'environnement `NOM` si elle existe ;
- dès qu'il rencontre une autre chaîne, elle et toutes les suivantes sont concaténées dans une chaîne `cmd` en les séparant par des espaces et le programme appelle `system(cmd)`.

La valeur du retour du programme est celle de la fonction `system`, ou `EXIT_SUCCESS` si toutes les chaînes étaient des deux premiers types. Si une erreur se produit en cours d'exécution, le programme doit terminer avec `EXIT_FAILURE`.

Utilisez votre programme pour lancer un shell et vérifier que les variables d'environnement ont bien été ajoutées ou supprimées comme prévu.

- + Cet exercice propose d'écrire une version restreinte de la commande `env` [🔗2](#), vous pouvez l'étendre pour émuler plus précisément cette commande, en ajoutant les fonctionnalités une à une.

### ⌂ Exercice 11 (★★★)

- Lister un répertoire, récupérer les informations d'un inode.
- ② Lancez `ls -aln --time-style=+` dans un shell et comparez ses sorties à celles de votre programme.

Écrivez un programme listant les fichiers du répertoire courant et affichant :

- ses attributs sous la forme `-rwxrwxrwx` où le premier caractère est
- '`-`' pour un fichier
- '`d`' pour un répertoire
- '`l`' pour un lien symbolique
- '`s`' pour un socket
- '`p`' pour un fifo (*pipe*)
- '`b`' pour un périphérique blocs
- '`c`' pour un périphérique caractères

et les autres caractères sont les bits `rwx` des trois types d'utilisateurs. Pensez à gérer les bits spéciaux (setuid, setgid, et *sticky*).

- le nombre de liens sur le fichier
- l'uid propriétaire
- le gid propriétaire
- la taille en octets
- le nom du fichier et, si c'est un lien symbolique, -> suivi de la cible du lien

⊕ Comme ls, triez les fichiers par ordre alphabétique, en listant toujours . et .. en premiers.

⊕ Lisez stat [2](#) et inode [7](#) pour voir comment est stockée la date de dernière modification sous Linux et ajoutez-la à l'affichage, comme le ferait ls -aln --time-style=long-iso dans le shell.

### ▀▀ Exercice 12 (★★★)

☒ Lister un répertoire de façon récursive. Comprendre l'allocation des `struct dirent`. Afficher proprement une arborescence sur un terminal texte.

⓪ L'entrée `d_name` d'une `struct dirent` n'a pas à être libérée : elle est allouée par `opendir` [3](#) et libérée par `closedir` [3](#). Donc, si on veut la garder, il faut la recopier avec `strdup` [3](#).

⓫ Pour afficher les liens d'une arborescence, on peut utiliser des caractères Unicode : "\u2514" affiche ↴, "\u2500" affiche –, "\u251c" affiche ↵, et "\u2502" affiche ↳.

⓬ En utilisant `chdir` [2](#) au moment de lister récursivement un sous-répertoire, vous éviterez de devoir gérer des chemins de taille grandissante.

1. Écrivez une fonction `int is_dir(char *path)` qui renvoie 1 si `path` est un répertoire et 0 sinon.
2. Définissez une structure `dir_t` pour mémoriser la liste des noms de fichiers (tous types confondus) dans un répertoire.
3. Écrivez une fonction `dir_t ls (char *path)` qui liste le contenu d'un répertoire `path` dans une structure `dir_t` en omettant . et .. (qui sont toujours présents).
4. Écrivez une fonction `void print_name (char* path, char *prefix, int last)` qui affiche une entrée de répertoire `path` en la faisant précéder du préfixe `prefix`, le paramètre `last` indique si c'est la dernière entrée du répertoire.
5. Écrivez une fonction `void tree (char *path, char *prefix, int last)` qui affiche les entrées d'un répertoire de nom `path`, avec les mêmes arguments que `print_name`, et lance récursivement l'affichage de ses sous-répertoires en ajustant leur préfixe.
6. Écrivez une fonction `main` qui affiche l'arborescence du répertoire courant.

⊕ Triez les entrées dans la fonction `ls`.

⊕ Regardez `tree` [1](#) et intégrez certaines de ses options.

### ▀▀ Exercice 13 (★★☆)

☒ Lire et écrire des fichiers. Créer et parcourir des répertoires. Gérer les méta-informations.

1. Écrivez une fonction `void cp_mode (char *src, char *tgt)` prenant deux noms de fichiers en paramètres et répliquant les permissions du premier sur le second.
2. Écrivez une fonction `void cp_file (char *src, char *tgt)` prenant deux noms de fichiers en paramètre et recopiant le premier sur le second (s'il existe, il est écrasé), en répliquant aussi ses permissions.
3. Écrivez une fonction `void cp_dir (char *src, char *tgt)` prenant deux noms de répertoire en paramètre et recopiant récursivement le premier sur le second (s'il existe, le programme s'arrête en erreur), en répliquant ses permissions.
4. Écrivez un programme prenant deux noms de fichiers ou répertoires sur sa ligne de commande et copiant le premier sur le second en préservant les permissions.

⊕ En vous appuyant sur les pages de manuel de Linux (cette partie n'est pas portable), répliquez aussi les dates de dernière modification et de dernier changement de statut, ainsi que l'uid et gid propriétaire.

### ▀▀ Exercice 14 (★★☆)

☒ Gérer des entrées/sorties. Trier des fichiers plus grands que la mémoire.

⓪ Le tri de fichiers par fusion consiste à découper un grand fichier à trier en morceaux assez petits pour tenir en mémoire, chacun étant alors chargé, trié et sauvegardé indépendamment. Ensuite, les morceaux sont fusionnés deux à deux, en lisant un élément dans chacun des deux fichiers et en écrivant le plus petit dans un fichier cible, jusqu'à recopie dans

l'ordre de l'intégralité des données des deux fichiers. Les morceaux ainsi fusionnés sont ensuite fusionnés entre eux de nouveau, et ainsi de suite jusqu'à ce qu'on ait obtenu un seul morceau fusionnant toutes les données.

➊ La première fonction demandée crée un fichier à trier, la seconde vérifie que le fichier résultat est bien trié.

Pour l'exercice, on va trier des fichiers contenant des entiers `int`, et on va utiliser des blocs en mémoire dont la taille est donnée par `#define CHUNKCOUNT 1024` (nombre d'entiers dans un bloc) et `#define CHUNKSIZE CHUNKCOUNT*sizeof(int)` (nombre d'octets dans un bloc).

1. Écrivez une fonction `void make_data (char *path, unsigned int size)` qui recopie `size` entiers `int` depuis le fichier `/dev/urandom` vers le fichier `path` (qui est créé ou écrasé pour).
- ❷ Pour tester votre programme, créez un fichier d'une dizaine de blocs, c'est suffisant et ce sera rapide.
2. Écrivez une fonction `int is_sorted (char *path)` qui renvoie 1 si les entiers `int` contenus dans le fichier `path` sont triés en ordre croissant, et 0 sinon.
3. Écrivez une fonction `char* chunk_name (char *base, unsigned int count, char *prev)` qui fabrique un nom de fichier en concaténant `base`, un point, et `count` sur 6 caractères de large (en ajoutant des zéros). Si `prev==NULL`, la fonction doit allouer la mémoire nécessaire et renvoyer le pointeur, sinon, elle doit réutiliser le pointeur `prev`. Cela permettra de réutiliser la même chaîne pour tous les noms de fichiers à générer.
4. Écrivez une fonction `int split (char *path)` qui lit un fichier `path`, par exemple considérons `path="FICHIER"`, et le découpe en morceaux de taille au plus `CHUNKCOUNT` entiers `int` et nommés en appelant `chunk_name`, par exemple `"FICHIER.000000"`, `"FICHIER.000001"`, etc. Chaque morceau doit être trié en mémoire avant d'être réécrit dans le fichier cible. Cette fonction renvoie le nombre de morceaux ainsi créés.
5. Écrivez une fonction `void fuse (char *src1, char* src2, char *tgt)` qui fusionne les entiers `int` contenus des fichiers `src1` et `src2` de façon à les ordonner dans `tgt`. Cette fonction ne doit pas charger les deux fichiers en mémoire mais lire et écrire les données entier par entier.
6. Écrivez une fonction `void sort_file (char *path)` qui trie les entiers `int` contenus dans le fichier `path` en utilisant la méthode du tri par fusion. Le fichier trié doit avoir le nom du fichier d'origine auquel on a ajouté le suffixe `".sorted"`. Les fichiers intermédiaires doivent être supprimés une fois qu'ils ne sont plus utiles.

➌ Modifiez votre programme pour qu'il puisse trier autre chose que des `int` et soit générique par rapport au type de données triés.

## ⌚ Séance 4 : processus et signaux

### 🏡 Exercice 15 (★★★)

➊ Créer des processus pour distribuer du travail. Gérer un groupe de processus. Découvrir le minage de *bitcoins*.

➋ Pour compiler avec OpenSSL sous CoW, ajoutez `// ldd: -lcrypto` dans votre code source comme sur la figure 4.

➌ Pour tuer tous les enfants d'un coup, vous pouvez utiliser `kill -9` sur un groupe de processus (voir `setpgid`).

Les transactions *bitcoins* sont validées par blocs, chaque bloc est désigné par une entête qui comporte un *nonce*. Ce nonce est un nombre arbitraire qui doit être choisi de manière à ce que le *hachage cryptographique* de l'entête soit une valeur numérique inférieure à une difficulté fixée. Un *mineur de bitcoins* est donc un logiciel qui cherche un tel nonce, le hachage cryptographique étant imprévisible, cela revient à énumérer tous les nonces possibles jusqu'à en trouver un qui valide la condition. Le premier mineur à publier une entête complétée d'un nonce valide remporte un nombre de *bitcoins* correspondant à la difficulté fixée. Pour l'exercice, on utilisera le hachage cryptographique MD5 utilisable depuis C comme sur la figure 4 en faisant appel à la bibliothèque OpenSSL. Par ailleurs, on va grandement simplifier le principe pour n'en garder que le cœur (trouver un nonce de façon à avoir un hachage assez petit).

1. Écrivez une fonction `int zeros (char *s, int n)` qui renvoie 1 si la chaîne `s` commence par `n` caractères '`0`', et 0 sinon.
2. Écrivez une fonction `void bruteforce (int first, int step, int zero)` qui :
  - énumère les entiers en partant de `first` et par incrément de `step`;
  - pour chacun, calcule sa représentation textuelle puis le hachage de celle-ci;
  - si ce hachage commence par `zero` caractères '`0`', la valeur de l'entier est enregistrée dans un fichier `found.PID` où `PID` est remplacé par le pid du processus, puis le processus termine avec le code de retour 0;
  - sinon, la fonction teste la valeur suivante.

3. Écrivez un programme qui lance 10 processus enfants, chacun exécutant la fonction `bruteforce` avec des paramètres permettant de répartir les calculs sur tous les enfants (c'est-à-dire : tous les entiers seront testés, mais chacun ne sera testé que par un seul enfant). Pour l'exercice, utilisez `zero=6` afin de limiter les temps de calculs (c'est une faible difficulté). Le programme attend son premier enfant qui termine avec le code 0 et alors, il tue tous les autres puis lit le fichier `found.PID` correspondant à l'enfant qui a été attendu pour afficher la valeur de l'entier enregistré dedans. Le fichier est alors effacé.

- ⊕ Vous pouvez lire la description de `bitcoins` et adapter votre programme pour qu'il utilise la même fonction de hachage et des entêtes bien formées. Pour fabriquer un vrai mineur de `bitcoins`, il ne restera "plus qu'à" ajouter des communications réseau pour récupérer et envoyer les entêtes sur les serveurs appropriés.

```
md5.c
1 // ldd: -lcrypto
2 #include <openssl/md5.h>
3 #include <string.h>
4 #include <stdio.h>
5
6 // = {0} => all the array is reset to zero (only works for zero!)
7 char hash[1 + 2*MD5_DIGEST_LENGTH] = {0};
8
9 char* md5hash (char *str) {
10    unsigned char md5[MD5_DIGEST_LENGTH] = {0};
11    MD5(str, strlen(str), md5);
12    for (int i=0; i<MD5_DIGEST_LENGTH; i++) {
13        sprintf(hash + 2*i, "%02x", md5[i]);
14    }
15    return hash;
16 }
```

➤ **Figure 4.** Code C pour calculer le hachage MD5 d'une chaîne `str`, la valeur est renvoyée sous forme d'un entier codé en hexadécimal dans une chaîne stockée dans `hash`.

### ▀▀ Exercice 16 (★★☆)

⊕ Créez et attendez des processus.

ⓧ Ce tri s'appelle le *sleepsort*, il fonctionne car les processus enfants vont se réveiller, et être récupérés par leur parent, par ordre croissant des valeurs à trier.

Ce programme va trier un tableau de la façon la plus paresseuse possible. Le programme doit déclarer un tableau `unsigned int tab[10]`; rempli de valeurs désordonnées entre 1 et 10, puis :

- pour chaque case `tab[i]`, le processus crée un enfant avec `fork` [🔗2](#);
- chaque enfant attend `tab[i]` secondes avec `sleep` [🔗3](#) puis termine avec le code de retour `tab[i]`
- pour chaque case `tab[i]`, le processus parent attend un de ses enfants, lit son code de retour, et l'écrit dans `tab[i]`.

Le tableau est alors trié.

1. Écrivez ce programme, en affichant le tableau au début et à la fin pour vérifier qu'il a bien été trié.
2. Une limitation évidente de cette méthode est qu'on ne peut pas trier des valeurs qui ne peuvent pas être des paramètres de `exit` [🔗3](#) qui tronque les valeurs à 127. Modifiez votre programme pour qu'il puisse fonctionner avec des entiers positifs quelconques, pour cela le résultat n'est pas transmis par `exit` [🔗3](#) mais écrit dans un fichier `sleepsort.PID` où `PID` est remplacé par le pid du processus enfant.
3. Reste une limitation : on ne peut pas trier des valeurs négatives car un processus enfant ne peut pas dormir un temps négatif. Trouvez un moyen de contourner cette limite et modifiez votre programme en conséquence.
4. Proposez une évolution permettant d'accélérer le tri tout en gardant son principe.

ⓧ Regardez du côté de `usleep` [🔗2](#).

### ▀▀ Exercice 17 (★★☆)

⊕ Envoyer et recevoir des signaux.

ⓧ Pour tirer des nombres au hasard, utilisez `rand` [🔗3](#), après avoir appelé `srand(getpid())` pour initialiser le générateur pseudo-aléatoire.

④ Attention à laisser le temps aux processus enfants de recevoir et traiter les signaux envoyés par le processus parent.

Écrivez un programme qui démarre 10 processus enfants. Chaque enfant a le comportement suivant :

- il tire au hasard un entier `num` entre 1 et 10 ;
- il affiche son pid et la valeur de sa variable `num` ;
- puis il dort indéfiniment ;
- s'il reçoit `num` fois le signal `SIGTERM`, il termine avec le code de sortie 0.

Le processus parent doit envoyer des signaux `SIGTERM` à ses enfants de façon à deviner la valeur de sa variable `num` selon le nombre de signaux qu'il aura envoyés avant sa terminaison. Pour chaque processus terminé, le parent affiche son pid et la valeur de sa variable `num`.

### Exercice 18 (★★★)

Envoyer et recevoir des signaux. Décoder le statut de sortie d'un processus.

Écrivez un programme qui démarre 10 processus enfant. Chaque enfant choisit aléatoirement et affiche l'un des comportements suivants :

- il dort indéfiniment ;
- il attend le signal `SIGUSR1` et l'ignore (indéfiniment) ;
- il attend le signal `SIGUSR1` et termine avec le code de sortie 0 ;
- il attend le signal `SIGUSR1` et termine avec le code de sortie 1.

Le processus parent doit deviner et afficher le comportement choisi par chacun de ses enfants.

### Exercice 19 (★★★)

Installer un comportement élaboré pour réagir à un signal. Terminer proprement un programme sur une demande de l'utilisateur. Recevoir plusieurs signaux.

④ Si on tape `Ctrl+C` dans le terminal, le processus actif dans le terminal reçoit le signal `SIGINT`.

④ Vous aurez besoin de `alarm`  $\textcircled{2}$ .

Écrivez un programme qui affiche "working..." toutes les secondes et qui termine avec le code de retour 0 si l'utilisateur tape deux fois `Ctrl+C` en moins de deux secondes. La première fois que l'utilisateur tape `Ctrl+C`, le programme affiche un message pour expliquer qu'il faut recommencer dans les deux secondes afin de terminer le programme. La seconde fois (avant deux secondes), le programme affiche un message d'au revoir. Si l'utilisateur ne tape pas une deuxième fois `Ctrl+C` dans les deux secondes, le programme affiche un message pour dire qu'il reprend son travail. Entre les deux messages, le programme ne doit pas afficher "working...".

## ► Séance 5 : communications entre processus

### Exercice 20 (★★★)

Dialoguer avec plusieurs processus.

④ Vous aurez bien sûr besoin de `pipe`  $\textcircled{2}$ , mais aussi de `poll`  $\textcircled{2}$ .

Reprenez l'exercice 15 pour faire en sorte que les processus enfants écrivent les nonces trouvés sur un pipe partagé avec leur parent (un pipe par enfant). Le parent affiche les 5 premiers nonces trouvés puis arrête tous ses enfants.

### Exercice 21 (★★★)

Comprendre comment le shell crée des pipes. Utiliser un membre de la famille `exec`  $\textcircled{3}$ .

④ Vous aurez besoin de `pipe`  $\textcircled{2}$  et `dup2`  $\textcircled{2}$ .

Écrivez un programme qui lance l'équivalent de la commande shell "ls -l|sed 's/\.\.c\$/.COUCOU/'".

⊕ Lorsque le shell crée des pipes, il place tous les processus participants dans un même groupe. Faites de même.

⊕ Le shell assure l'interprétation de certains caractères spéciaux, comme \* dans les noms de fichiers. Modifiez votre programme pour qu'il lance l'équivalent de la commande shell "ls -l \*.c|sed 's/\.\.c\$/.COUCOU/'". Vous aurez besoin de `glob`  $\textcircled{3}$  pour transformer \*.c en une liste de noms de fichiers.

### Exercice 22 (★★★)

Utiliser les files de messages IPC système V.

Écrivez un programme qui prend en paramètre une commande, et exécute une action sur une file de messages correspondant à la commande. La file de messages doit être créée au besoin. Les messages font au plus 256 octets. Toutes les erreurs doivent être signalées à l'utilisateur. Les commandes supportées par le programme seront les suivantes :

- **snd** envoie un message : le message est lu au clavier, sur plusieurs lignes éventuellement, jusqu'à ce que l'utilisateur tape une ligne vide ;
  - **rcv** reçoit et affiche le premier message de la file ;
  - **stat** affiche le nombre de messages sur la file d'attente ainsi que leur taille totale ;
  - **rm** supprime la file d'attente.
- ⊕ Structurez les messages avec des champs envoyeurs et destinataires, codés comme les uid des utilisateurs, ainsi qu'une date d'envoi. La commande **rcv** ne doit prendre en compte que les messages à destination de l'utilisateur qui lance le programme.

### Exercice 23 (★★★☆)

- Comparer les files de messages aux pipes nommés.
  - ⊖ Un pipe nommé est vidé lorsque tous les processus le ferment, et il est bloquant en lecture (resp. écriture) si aucun processus ne l'a ouvert en écriture (resp. lecture). Pour régler ça, écrivez un programme serveur qui crée le pipe nommé, et le garde ouvert aux deux extrémités pour éviter qu'il ne se vide et qu'il soit bloquant. Ce serveur sera lancé en tâche de fond, et il effacera le pipe nommé lorsqu'on le tuera avec **Ctrl+C**.
  - ⊖ Pour ouvrir un pipe nommé sans blocage, il faut utiliser l'option **O\_NONBLOCK**, voir [open](#).
- Reproduisez les commandes **snd** et **rcv** du programme de l'exercice 22 en utilisant un pipe nommé au lieu d'une file de messages. Vous devez trouver un moyen de séparer les messages dans le pipe.
- ⊕ Reproduire la commande **stat** en stockant les informations nécessaires dans un fichier.
- ⊕ Gérez les accès concurrents à ce fichier avec des verrous.

## Séance 6 : programmes concurrents en mémoire partagée

### Exercice 24 (★★★☆)

- Utiliser des segments de mémoire partagé. Observer l'effet d'accès concurrents à de la mémoire partagée.

Écrivez un programme réalisant les actions suivantes :

- création d'un segment de mémoire partagée assez grand pour stocker un compteur **int** et un tableau **int[100]**, le compteur représente le nombre d'entiers non nuls dans le tableau ;
- attachement au segment partagé et déclaration de variables dedans pour le compteur et le tableau ;
- initialisation du compteur et du tableau à 0 ;
- création de 10 processus enfants ;
- attente de la terminaison de tous les enfants ;
- affichage du contenu du tableau ;
- vérification que chaque élément du tableau est strictement inférieur au précédent ;
- détachement puis suppression du segment.

Chaque processus enfant doit réaliser les actions suivantes :

- attachement au segment partagé et déclaration de variables dedans pour le compteur et le tableau ;
- génération de 10 entiers aléatoires entre 2 et 100, pour chacun, s'il est premier et n'est pas déjà dans le tableau, ajout de l'entier à la fin du tableau et incrémentation du compteur ;
- tri du tableau en ordre croissant ;
- détachement du tableau.

En faisant plusieurs exécutions successives, vérifiez qu'on en trouve où le tableau final affiché par le processus parent contient plusieurs fois le même entier. Expliquez ce phénomène.

### Exercice 25 (★★★☆)

- Utiliser des sémaphores IPC système V pour régler les problèmes d'accès concurrents à la mémoire.

Corrigez le programme de l'exercice 24 en supprimant les accès concurrents à la mémoire grâce à des sémaphores IPC système V.

### Exercice 26 (★★★☆)

- Utiliser les sémaphores POSIX et les comparer à ceux des IPC système V.

- ⊖ Utilisez des sémaphores nommés pour plus de facilité, mais pensez à les effacer.

Refaites le programme de l'exercice 25 en utilisant des sémaphores POSIX.

- ⊕ Refaites le programme avec des sémaphores anonymes stockés dans le segment partagé.

### Exercice 27 (★★★☆)

- Écrire un programme multithreads.

Refaites l'exercice 20 en utilisant des threads POSIX et un tableau partagés pour stocker les nonces trouvés.

### Exercice 28 (★☆☆)

- Écrire un programme multithreads.

Refaites l'exercice 16 en utilisant des threads POSIX, chaque thread lit et écrit directement dans le tableau résultat en prenant soin de gérer la possibilité de valeurs répétées.

### Exercice 29 (★★☆)

- Écrire un programme multithreads avec des structures de données synchronisées. Déetecter et gérer la terminaison d'un programme multithread complexe.

❶ Utilisez les files de messages POSIX pour gérer la file de tâches ([mq\\_overview](#) ), ainsi vous n'aurez pas à gérer le verrouillage de la structure.

❷ En utilisant un sémaphore pour le compteur partagé, vous n'aurez pas non plus à gérer son verrouillage (et utilisez [sem\\_getvalue](#)  pour consulter sa valeur).

❸ L'algorithme *quicksort* vous est fourni dans le fichier `quicksort.c`.

Écrivez un *quicksort* multithreads sur le schéma suivant :

- le tableau à trier est partagé entre  $n$  threads ;
- une file de tâches est aussi partagée, une tâche correspond aux bornes d'un sous-tableau à trier ;
- chaque thread prend une tâche dans la file, partitionne le sous-tableau correspondant, puis replace les deux sous-tableaux résultants dans la file s'ils contiennent au moins deux éléments.

La principale difficulté est de détecter la fin du tri, et on ne peut pas s'appuyer sur la file de tâches pour ça. En effet, elle peut être vide pendant que des threads sont en train de partitionner des sous-tableaux. À la place :

- le programme déclare un sémaphore initialisé au nombre de valeurs à trier (la taille du tableau) ;
- le programme déclare aussi un condition `pthread_cond_t` qu'il attend une fois les threads créés ;
- pour chaque sous-tableau résultant d'un partitionnement et ne contenant qu'un seul élément (qui ne sera donc pas remis dans la file de tâches), le thread décrémente le sémaphore ;
- si le sémaphore tombe à zéro (voir [sem\\_getvalue](#) ), c'est que le tableau est trié, le thread qui l'a mis à zéro appelle [pthread\\_cond\\_signal](#)  pour débloquer le programme ;
- quand le programme est débloqué, il arrête chaque threads avec [pthread\\_cancel](#)  et l'attend avec [pthread\\_join](#) .

Quelques questions utiles à se poser :

1. Pourquoi n'a-t-on pas besoin de gérer de verrous sur le tableau ?
2. Quelle taille maximale la file de tâches doit-elle faire ?
3. Pourquoi un thread ne peut jamais se bloquer en décrémentant le sémaphore ?
4. Est-il possible que plusieurs threads trouvent le sémaphore à zéro ?
5. Si oui, quelle en sera la conséquence ?