TRAVAUX PRATIQUES

UFAR - IMA-2 - Systèmes 2 - 2024-2025

Emmanuel Rio - Université de Toulouse

Conseils pour compiler les programmes C

Ligne de commande de `gcc`

Utiliser le modèle suivant :

```
gcc -Wall -Werror exercice.c -o exercice
```

Détail des options :

- `-Wall` : affiche les messages d'avertissement (*warnings*).
- `-Werror` : considère les *warnings* comme des erreurs (il n'y a **pas production d'exécutable** dans ce cas).

Utilisation d'un fichier `Makefile`

Il est possible d'utiliser la commande `make` avec le fichier de définition `Makefile` suivant *(sans extension de fichier)* à placer dans le répertoire de travail :

```
CC = gcc
.SUFFIXES: .c .h
.c:
    ${CC} -Wall -Werror -o $@ $<
```

Une fois le fichier `Makefile` correctement écrit et placé, la compilation peut d'effectuer en exécutant dans le terminal :

```
make exercice
```

Cette commande compile le fichier `exercice.c` et produit l'exécutable `exercice`.

Partie 1 - Exécuter, identifier, observer

Exercice 1 - Produit des arguments

1) a) Écrire un programme en C qui :

- accepte un nombre quelconque d'arguments.
- convertit ces arguments en entiers avec la **fonction `atoi`**.
- calcule et affiche le produit de ces arguments.

Indication : si aucun argument n'est donné, le programme affiche le produit `1`.

b) Contrôler que l'on obtient bien les affichages suivants :

```
~/partie1$ ./exercice1 123 -456 789
Produit : -44253432
~/partie1$ ./exercice1
Produit : 1
~/partie1$ ./exercice1 67 0 89
Produit : 0
~/partie1$ ./exercice1 67 foo 89
Produit : 0
```

c) Expliquer le dernier affichage. Pourquoi cela pose-t-il problème ?

2) a) Corriger le défaut observé à la question **1**, en utilisant la fonction `strtol` afin de convertir les arguments en entiers.

Rappel : la fonction `strtol` s'utilise de la façon suivante pour convertir une chaine de caractère `chaine` en sa valeur entière :

```c
char *suite;
errno = 0;
long valeur = strtol(chaine,&suite,10);
if (errno!=0 || *suite!='\0') {
    // traitement de l'erreur
}
else {
    // utilisation de valeur
}
```

Le comportement du programme doit alors être le suivant :

- Si l'un des arguments n'est pas un entier, le programme affiche une erreur et se termine avec le code de retour `EXIT_FAILURE`.
- Sinon, le programme affiche le produit de ses arguments et se termine avec le code de retour `EXIT_SUCCESS`.

b) Contrôler que l'on obtient bien les affichages suivants :

```
~/partie1$ ./exercice1 123 -456 789
Produit : -44253432
~/partie1$ echo $?
0
~/partie1$ ./exercice1
Produit : 1
~/partie1$ echo $?
0
~/partie1$ ./exercice1 67 0 89
Produit : 0
~/partie1$ echo $?
0
~/partie1$ ./exercice1 67 foo 89
Erreur de conversion
~/partie1$ echo $?
1
```

Exercice 2 - Affichage du PID et du PPID

1) Écrire un programme qui affiche son **PID** et son **PPID**.

2) Contrôler le résultat dans le terminal, en comparant le **PPID** du processus avec le **PID** du shell dans lequel on l'exécute :

Exemple d'affichage :

```
~/partie1$ ./exercice2
PID=561082
PPID=559326
~/partie1$ echo $$
559326
```

3) a) Observer les appels sytèmes effectués par le programme avec la commande `strace` :

```
~/partie1$ strace ./exercice2
```

b) Afin de n'observer que les appels systèmes pertinents, utiliser la commande `strace` avec l'option `-e trace=write,getpid,getppid`.

Exercice 3 - Compte à rebours

1) Écrire un programme qui :

- accepte en argument exactement un entier `n`.
- vérifie le nombre d'arguments.
- convertit cet argument avec la **fonction `atoi`** *(pour simplifier)*.
- affiche un décompte **en secondes** de `n` à `0`.

Indication : utiliser le fonction `sleep`.

Exemple d'affichage : pour `n=4`

```
4
3
2
1
0
```

2) Observer les appels sytèmes effectués par le programme avec la commande `strace`.

Partie 2 - Création de processus et recouvrement

Exercice 1 - Attends-moi !

1) Écrire un programme qui :

- affiche son **PID** et son **PPID**.
- crée un processus enfant.
- attend pendant **1 seconde**.
- termine immédiatement après **sans attendre son enfant**.

Le processus **enfant** :

- affiche son **PID** et son **PPID**.
- attend pendant **2 secondes**.
- affiche de nouveau son **PID** et son **PPID** *(avec un **nouvel appel** des fonctions `getpid` et `getppid`)*

Indication : ne pas utiliser la fonction `wait` dans cette question.

c) Vérifier que, lors du **premier affichage**, le **PPID de l'enfant** est bien le **PID du parent**.

d) Pourquoi le processus enfant n'affiche **pas le même PPID** lors du **second affichage** ? Comment appelle-t-on un tel processus enfant ?

2) a) Modifier le programme afin que le processus parent (après avoir attendu 1 seconde) attende aussi la terminaison du processus enfant.

b) Vérifier que dans ce cas le processus enfant affiche deux fois le même **PPID**.

Exercice 2 - Encore un compte à rebours

Écrire un programme qui :

- accepte en argument exactement un entier `n`.
- convertit ce paramètre avec la **fonction `atoi`** *(pour simplifier)*.
- affiche son **PID**.
- crée un processus enfant.
- termine **après que son enfant ait fini**.

Le processus **enfant** :

- affiche un décompte en secondes de `n` à `0`, en indiquant à chaque fois son **PPID**.

Contrôler que le qu'à chaque affichage du décompte le processus enfant affiche un **PPID égal au PID du parent** :

Exemple d'affichage : pour `n=4`

```
Parent [PID=9531]
Enfant [PPID=9531] : 4
Enfant [PPID=9531] : 3
Enfant [PPID=9531] : 2
Enfant [PPID=9531] : 1
Enfant [PPID=9531] : 0
```

Exercice 3 - Calculs d'inverses

On souhaite dans cet exercice mettre en oeuvre le mécanisme de transmission d'un code de retour de l'enfant au parent.

Écrire un programme qui :

- Accepte un nombre quelconque d'arguments.
- Crée autant de processus enfants que d'arguments.

Chaque **enfant** :

- lit et essaie de convertir un des arguments en entier avec `atoi`.
- si `atoi` retourne un entier non nul :
  - affiche son numéro et l'inverse (un `float`) de cet entier.
  - termine avec le code de retour `EXIT_SUCCESS`.
- sinon :
  - affiche son numéro et un message d'erreur.
  - termine avec le code de retour `EXIT_FAILURE`.

Le **parent** :

- Attend la fin de tous les enfants.
- Affiche le nombre de succès et d'échecs

Exemples d'affichage :

```
~/partie2$ ./exercice3
Bilan : 0 succès, 0 échecs
~/partie2$ ./exercice3 1 -8 0 45 foo 2
Enfant 1 : 1/1 ~= 1.0000
Enfant 3 : erreur pour "0"
Enfant 2 : 1/-8 ~= -0.1250
Enfant 4 : 1/45 ~= 0.0222
Enfant 5 : erreur pour "foo"
Enfant 6 : 1/2 ~= 0.5000
Bilan : 4 succès, 2 échecs
```

Remarque : on observe que l'ordre d'affichage n'est pas nécessairement l'ordre de création.

Exercice 4 - Calculs longs en parallèle

*On souhaite dans cet exercice montrer l'intérêt de déléguer une tâche de façon **parallèle** à des processus enfants.*

Écrire un programme qui :

- accepte un nombre quelconque d'arguments entiers sur la ligne de commande.
- crée un processus enfant par entier.

Chaque **enfant** :

- convertit la chaîne en entier `long` avec `strtol`.
- utilise la fonction `estPremier` fournie ci-dessous pour savoir si l'entier est **premier**.
- affiche son numéro et si le nombre est premier ou non.
- termine avec :
  - `0` si le nombre est premier,
  - `1` si le nombre n'est pas premier,
  - `2` si `strtol` a provoqué une erreur.

Le **parent** :

- attend la fin de **tous les enfants**,
- comptabilise :
  - le **nombre de nombres premiers**,
  - le **nombre de nombres non premiers**
  - le **nombre d'erreurs** de conversion.

Fonction `estPremier` :

```
int estPremier(long n) {
    if (n <= 1) return 0;
    for (long d = 2; d*d<=n; d++) {
        if (n%d==0) return 0;
    }
    return 1;
}
```

Exemple d'affichage :

```
~/partie2$ ./exercice4 77777777977777777 foo 17 111 0
Enfant 2 : erreur pour foo.
Enfant 3 : 17 est premier.
Enfant 4 : 111 n'est pas premier.
Enfant 5 : 0 n'est pas premier.
Enfant 1 : 77777777977777777 est premier.
Bilan :
  Nombres premiers     : 2
  Nombres non premiers : 2
  Erreurs de conversion: 1
```

Observation : on remarque que le fait de déléguer les calculs à des processus enfants permet d'obtenir l'affichage des résultats pour les petits entiers, alors que le grand entier donné en premier argument est encore en train d'être traîté.

Indications :

Quelques grands nombres à tester :

- `77777777977777777` (8 `7` puis 1 `9` puis 8 `7`) est premier
- `1111111111111111111` (19 `1` à la suite) est premier.

Exercice 5 - Premiers recouvrements

1) Écrire un programme en C nommé `exercice5_affiche_pids` qui affiche son **PID** et son **PPID**.

2) a) Écrire un second programme qui :

- Commence aussi par afficher son **PID** et son **PPID**.
- Utilise ensuite la fonction `execve` pour exécuter `exercice5_affiche_pids`.

b) Exécuter ce second programme et analyser les **PID** et **PPID** affichés.

Exercice 6 - Création et recouvrement

1) Écrire un programme en C qui accepte un nombre quelconque d'arguments. Chaque paramètre est le nom d'une commande **sans paramètre** *(exemples : `ls`, `pwd`, …)*. Le programme lance **successivement** autant d'enfants qu'il a reçu d'arguments.

Chaque **enfant** :

- Affiche son **PID** et la commande qui lui correspond.
- Utilise la fonction `execvp` pour exécuter cette commande *(consulter `man execvp` pour connaître sa signature)*
- Si la fonction `execvp` échoue, affiche un message d'erreur et termine avec le code de retour `EXIT_FAILURE`.

Le **parent** :

- Attend chaque enfant avant de créer l'enfant suivant.
- Affiche le code de retour de chaque enfant.
- Une fois tous les processus enfants terminés, affiche un bilan du nombre de commandes ayant réussi et échoué.

Exemple d'affichage :

```
~/partie2$ ./exercice6 ls toto pwd
Enfant 1 [PID=564526] lance : ls
exercice6  exercice6.c  Makefile
Parent : enfant PID=564526 termine avec code 0
Enfant 2 [PID=564527] lance : toto
execve: No such file or directory
Parent : enfant PID=564527 termine avec code 1
Enfant 3 [PID=564528] lance : pwd
/home/jdoe/partie2
Parent : enfant PID=564528 termine avec code 0
Bilan : 2 succès, 1 échecs
```

2) On souhaite pouvoir exécuter des commandes avec paramètres *(exemples : `ls -l`, `sleep 5`, …)*, en donnant au programme les arguments sous la forme suivante :

```
~/partie2$ ./exercice6 "ls -l" "sleep 5" pwd
```

Remarque : la délimitation à l'aide des guillemets `"..."` permet de regrouper chaque commande avec ses paramètres.

Adapter de code écrit à la question **1** de façon à utiliser la fonction `decouperCommande` qui accepte en paramètres :

- Un chaine de caractères `commande` à découper en ses arguments.
- Un tableau de chaînes de caractères `arguments`, de taille `NB_ARGUMENTS_MAX+1` et qui contiendra les différents arguments de la commande, suivis d'un pointeur `NULL`. Ce tableau est exactement celui qui devra être fourni à la fonction `execvp`.

Cette fonction retourne le nombre d'arguments *(y compris le nom de la commande)* lus.

Le code de la fonction `decouperCommande` est le suivant :

```
int decouperCommande(char *commande,char *arguments[]) {
    int nbArguments = 0;
    int i = 0;
    while (commande[i]!='\0') {
        if (commande[i]==' ') {
            commande[i]='\0';
        }
        else if (
            (i==0 || commande[i-1]=='\0')
            && commande[i]!='\0'
            && nbArguments<NB_ARGUMENTS_MAX
        ) {
            arguments[nbArguments] = &(commande[i]);
            nbArguments++;
        }
        i++;
    }
    arguments[nbArguments] = NULL;
    return nbArguments;
}
```

Partie 3 - Gestion des fichiers

Exercice 1 - Fonction `stat`

1) Écrire un programme en C qui :

- Accepte **un unique argument** contenant le **nom d'un fichier**.
- Affiche les métadonnées suivantes :
  - Le **numéro d'inode**.
  - Le **type de fichier**.
  - L'**identifiant de l'utilisateur** propriétaire.
  - La **taille du fichier** *(en octets)*.
  - La date de **dernière modification**.

Si une erreur est rencontrée, le programme doit retourner un code d'erreur.

Indication : On pourra s'aider de `man 2 stat` et `man 7 inode`.

Exemple d'affichage :

```
~/partie3$ ./exercice1 Makefile
Inode : 9593150
Type de fichier : Fichier regulier
Id proprietaire : 1000
Taille : 233 octets
Derniere modification : Sat Apr 1 18:24:01 1976
```

2) Comparer avec l'affichage correspondant réalisé avec la commande `stat` :

```
~/partie3$ stat Makefile
  File: Makefile
  Size: 233         Blocks: 8          IO Block: 4096   regular file
Device: 10302h/66306d   Inode: 9593150     Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/    stuart)   Gid: ( 1000/    stuart)
Access: 1976-04-01 21:04:45.556155034 +0400
Modify: 1976-04-01 18:24:01.303112000 +0400
Change: 1976-04-01 21:04:42.604063717 +0400
 Birth: 1976-04-01 00:00:00.604063717 +0400
```

Exercice 2 - Lecture d'un fichier

Écrire un programme qui :

- Accepte exactement 2 arguments *(le vérifier)* : le **nom d'un fichier** et un **caractère**.
- Ouvre avec `open` le fichier en **lecture seule**.
- Lit le fichier par blocs de `1024` (par exemple).
- Compte au fur et à mesure le nombre d'occurrences du caractère donné en argument.
- Ferme le fichier.
- Affiche le caractère cherché et le nombre d'occurrences.

Exemple d'affichage :

```
~/partie3$ ./exercice2 Makefile A
Caractère 'A' : 7 occurrence(s)
~/partie3$ ./exercice2 Makefile B
Caractère 'B' : 3 occurrence(s)
```

Exercice 3 - Accès concurrent à un même fichier

Contrainte : programmer chacune des 3 questions de cet exercice dans un fichier `.c` différent.

1) a) Écrire un programme qui :

- Accepte exactement 2 arguments *(le vérifier)* : le **nom d'un fichier** et une **chaîne de caractères**.
- Ouvre avec `open` le fichier en **lecture seule**.
- Crée autant de processus enfants que la longueur de la chaîne de caractères. À chaque enfant correspond donc un caractère de la chaîne de caractères.

Exemple : si la chaîne est `bar`, le premier enfant a le caractère `a`, le deuxième `b`, etc.

Chaque **enfant** :

- Parcourt le fichier et compte le nombre d'occurences de son caractère dans le texte.
- Affiche :
  - Son numéro
  - Le caractère qu'il a cherché
  - Le nombre d'occurrences
- Termine avec succès.

Le **parent** :

- Attend la fin de tous les enfants.
- Ferme le fichier.

b) Observer que l'on obtient des **résultats incohérents**. Expliquer.

Exemple d'affichage :

```
~/partie3$ ./exercice3_q1 Makefile A
Enfant 1 (caractère 'A') : 7 occurrence(s)
~/partie3$ ./exercice3_q1 Makefile B
Enfant 1 (caractère 'B') : 3 occurrence(s)
~/partie3$ ./exercice3_q1 Makefile AB
Enfant 1 (caractère 'A') : 7 occurrence(s)
Enfant 2 (caractère 'B') : 0 occurrence(s)
```

2) Pour résoudre le problème une **première solution** est que **chaque enfant ouvre le fichier** et obtient donc un descripteur de fichier différent des autres enfants.

Implémenter cette solution et vérifier que les résultats deviennent cohérents.

Exemple d'affichage :

```
~/partie3$ ./exercice3_q2 Makefile A
Enfant 1 (caractère 'A') : 7 occurrence(s)
~/partie3$ ./exercice3_q2 Makefile B
Enfant 1 (caractère 'B') : 3 occurrence(s)
~/partie3$ ./exercice3_q2 Makefile AB
Enfant 1 (caractère 'A') : 7 occurrence(s)
Enfant 2 (caractère 'B') : 3 occurrence(s)
```

3) Une **deuxième solution** est de revenir à la configuration de la question **1** où **seul le parent ouvre le fichier**. Pour garantir la cohérence des résultats, les enfants doivent alors utiliser la **fonction `pread`**.

La **fonction `pread`** est en effet **atomique** du point de vue du descripteur de fichier : elle effectue en **une seule opération la lecture à une position donnée** sans modifier la position courante du descripteur.

Signature :

```
ssize_t pread(int fd, void *buffer, size_t tailleBuffer, off_t position);
```

Implémenter cette solution et vérifier que les résultats deviennent ici aussi cohérents.

Exercice 4 - Modification d'un fichier

Écrire un programme qui :

- Accepte exactement 3 arguments *(le vérifier)* : le **nom d'un fichier** et deux **caractères**.
- Ouvre le fichier avec `open`.
- Lit le fichier par blocs de `1024` (par exemple).
- Remplace au fur et à mesure le premier caractère par le second dans le fichier.
- Ferme le fichier.
- Affiche le nombre de remplacements effectués.

Exemple d'affichage :

```
~/partie3$ ./exercice4 Makefile A Z
7 remplacement(s) 'A' -> 'Z'
~/partie3$ ./exercice4 Makefile Z A
7 remplacement(s) 'Z' -> 'A'
```

Attention ! Il est nécessaire d'utiliser `lseek` pour revenir en arrière après avoir lu un bloc pour écrire par-dessus ce même bloc.

Exercice 5 - Déplacement dans un fichier

L'objectif de cet exercice est de se déplacer à l'intérieur d'un fichier avec la fonction `lseek`. Les déplacements dépendent des caractères lus.

Contrainte : ne pas enregistrer le contenu du fichier dans une structure de données (tableau, ...)

Les fichiers manipulés dans cet exercice auront tous la forme suivante :

- uniquement consitués de caractères `'v'`, `'>'` et `'\n'`.
- toutes les lignes sont de même longueur **non nulle**.

Exemple :

```
v>>>v>v>
>>v>v>v>
vv>>>>vv
>>vvvv>>
```

1) Écrire la fonction `longueurPremiereLigne` qui accepte en argument un descripteur de fichier `fd` et retourne la longueur de la première ligne (ou du fichier complet s'il n'y a pas de retour à la ligne). La position dans le fichier est réinitialisée à `0` avant le retour de la fonction.

2) Écrire la fonction `estFichierValide` qui accepte en paramètre un descripteur de fichier `fd` et retourne `1` si le fichier respecte les contraintes, `0` sinon. La position dans le fichier est réinitialisée à `0` avant le retour de la fonction.

3) Écrire un programme qui :

- Ouvre un fichier au format indiqué.
- Vérifie que le fichier est valide.
- Part du premier caractère *(en haut à gauche)* du fichier.
- Suit la direction de la flèche :
  - Si le caractère lu est `'v'` se déplace d'une ligne vers le bas.
  - Si le caractère lu est `'>'` se déplace d'un caractère vers la droite.
- Répète l'opération en affichant à chaque fois :
  - La position `(ligne,colonne)` dans le fichier.
  - Le caractère lu `'v'` ou `'>'`.
- Le parcours se termine dès que la flèche fait sortir du fichier.

Exemple :

Si le contenu du fichier `carte.txt` est :

```
v>>>v>v>
>>v>v>v>
vv>>>>vv
>>vvvv>>
```

Alors l'affichage est :

```
~/partie3$ ./exercice5 carte.txt
Position (0,0) : v
Position (1,0) : >
Position (1,1) : >
Position (1,2) : v
Position (2,2) : >
Position (2,3) : >
Position (2,4) : >
Position (2,5) : >
Position (2,6) : v
Position (3,6) : >
Position (3,7) : >
Sortie à droite
```

Partie 4 - Mémoire virtuelle

L'objectif de cette partie est légèrement différente des précédentes. On souhaite ici ***programmer des algorithmes*** *liés à la mémoire virtuelle, afin de les comprendre. Le comportement des fonctions écrites est uniquement caractérisé par leur valeur de retour et leur effet sur certains paramètres. On peut donc utiliser ici les fonctions* `assert` *qui permettent d'effectuer des tests unitaires.*

Exercice 1 - Pagination

Un contexte de pagination de pagination est caractérisé par :

- La taille des pages *(en octets)*
- La taille de la mémoire virtuelle *(en octets)*
- La taille de la mémoire physique *(en octets)*
- Une table des pages
- Le contenu de la mémoire physique

À partir de ce contexte, on doit pouvoir déterminer :

- S'il y a un **défaut de page**
- Sinon, **l'octet lu**.

1) a) Dans le fichier `exercice1.c` fourni, compléter la fonction `lireAdresse` qui accepte en paramètres :

- `taillePage` *(un `int`)*
- `tailleMemoireVirtuelle` *(un `int`)*
- `tailleMemoirePhysique` *(un `int`)*
- `tablePages` *(un tableau d'`int` de taille `tailleMemoireVirtuelle`)*
- `memoirePhysique` *(un tableau d'`unsigned char` de taille `tailleMemoirePhysique`)*
- `adresse` *(un `int`)*

La fonction délivre en sortie `pOctet` (pointeur sur `unsigned char`) l'octet lu à l'adresse virtuelle `adresse`. La fonction retourne `1` en cas de défaut de page, `0` sinon.

b) Une fois compilé et exécuté, le programme doit passer tous les `assert`s.

2) Pour l'instant, les paramètres `tailleMemoireVirtuelle` et `tailleMemoirePhysique` n'ont pas été utilisés.

Afin de sécuriser la fonction, utiliser ces deux paramètres pour vérifier que les accès en lecture aux tableaux `tablePages` et `memoirePhysique` sont toujours effectués à des **indices valides**.

Si l'on tente de lire l'un des ces deux tableaux avec un indice invalide, la fonction doit retourner le code d'erreur `2`.

Exercice 2 - Algorithme LRU

Dans cet exercice on souhaite simuler l'algorithme de remplacement LRU.

Le fichier `exercice2.c` fournit le prototype des fonctions suivantes :

```
void afficherMemoire(int nbCadres, const int memoire[], const int horodatage[]);
int rechercherPage(int nbCadres, const int memoire[], int page);
int indexRemplacement(int nbCadres, const int memoire[], const int horodatage[]);
```

L'objectif de cet exercice est tout d'abord de compléter au fur et à mesure ces fonctions. Dans un second temps on utilisera ces fonctions pour implémenter l'algorithme LRU sur des séquences de pages demandées.

Format des variables utilisées :

Dans tout cet exercice :

- `pages` (tableau d'`int` de longueur `nbPages`) contient la séquence des numéros de pages demandées.
- `memoire` (tableau d'`int` de longueur `nbCadres`) contient les numéros de pages contenues dans la mémoire physique. Une valeur de `-1` correspond à un cadre libre.
- `horodatage` (tableau d'`int` de longueur `nbCadres`) contient les dates (indice dans le tableau `pages`) de **dernière utilisation**.

1) Compléter la fonction `afficherMemoire` qui accepte en paramètres l'entier `nbCadres` et les tableaux `memoire` et `horodatage`.

Exemple d'affichage *(la date de dernière utilisation est indiquée entre parenthèses)* :

```
Mémoire : 1(0) 5(1) -1(-1) -1(-1)
```

Une fois la fonction complétée, compiler et exécuter. L'affichage doit être correct, et le programme doit bloquer sur l'`assert` du test suivant.

2) Compléter la fonction `rechercherPage` qui accepte en paramètres :

- l'entier `nbCadres` et le tableau `memoire`.
- un numéro de page `page`.

La fonction retourne l'index de la page en mémoire si elle est présente en mémoire, et `-1` sinon.

Une fois la fonction complétée, compiler et exécuter. Le test de cette fonction doit être validé, et le programme doit bloquer sur l'`assert` du test suivant.

3) Compléter la fonction `indexRemplacement` qui accepte en paramètres l'entier `nbCadres` et le tableau `horodatage`.

La fonction recherche dans la mémoire l'index du cadre qui doit être remplacé, c'est-à-dire celui qui a l'horodatage le plus ancien.

Une fois la fonction complétée, compiler et exécuter. Le test de cette fonction doit être validé, ce qui permet de finir l'ensemble des tests.

4) Utiliser les 4 fonctions écrites précédemment afin d'implémenter dans la fonction `main` l'algorithme LRU sur l'exemple proposé.

Exemple d'affichage :

```
------------------------------
Page demandée : 1
Défaut de page
Mémoire : 1(0) -1(-1) -1(-1) -1(-1)
------------------------------
Page demandée : 5
Défaut de page
Mémoire : 1(0) 5(1) -1(-1) -1(-1)
------------------------------
Page demandée : 3
Défaut de page
Mémoire : 1(0) 5(1) 3(2) -1(-1)
------------------------------
Page demandée : 2
Défaut de page
Mémoire : 1(0) 5(1) 3(2) 2(3)
------------------------------
Page demandée : 1
Page présente
Mémoire : 1(4) 5(1) 3(2) 2(3)
------------------------------
Page demandée : 2
Page présente
Mémoire : 1(4) 5(1) 3(2) 2(5)
------------------------------
Page demandée : 4
Défaut de page
Mémoire : 1(4) 4(6) 3(2) 2(5)
------------------------------
Page demandée : 0
Défaut de page
Mémoire : 1(4) 4(6) 0(7) 2(5)
------------------------------
```

```
Page demandée : 1
Page présente
Mémoire : 1(8) 4(6) 0(7) 2(5)
------------------------------
Page demandée : 2
Page présente
Mémoire : 1(8) 4(6) 0(7) 2(9)
------------------------------
```

5) On peut finaliser l'exercice de façon à accepter en arguments du programme :

- Le nombre de cadres de la mémoire physique.
- La séquence des pages appelées.

Le programme convertit chacun de ses paramètres en entier, remplit un tableau `pages` et les tableaux `memoire` et `horodatage` (initialement remplis de valeurs `-1`) conformément aux paramètres, puis met en oeuvre l'algorithme LRU écrit à la question **4**.

Exemple d'exécution *(qui correspond à l'exemple de la question **4**)* :

```
~/partie4$ ./exercice2 4 1 5 3 2 1 2 4 0 1 2
```

6) Que faudrait-il changer dans les questions précédentes pour implémenter l'algorithme **FIFO** au lieu de l'algorithme **LRU** ?