Licence 2 Informatique

SYSTÈMES 2

Sujet de Travaux Pratiques sur Machine

Alain CROUZIL, Jean-Denis DUROU, Jean-Marc PIERSON

{crouzil,durou,pierson}@irit.fr

Table des matières

1 Sujets des Travaux Pratiques sur Machine **3**
Séance 1 . . . 3
Exercice 1 . . . 3
Exercice 2 . . . 3
Séance 2 . . . 6
Exercice 3 . . . 6
Séance 3 . . . 8
Exercice 4 . . . 8
Exercice 5 . . . 10
Exercice 6 . . . 10
Exercice 7 . . . 11
Séance 4 . . . 12
Exercice 8 . . . 12
Exercice 9 . . . 12
Exercice 10 . . . 13
Séance 5 . . . 14
Exercice 11 . . . 14
Exercice 12 . . . 14
Exercice 13 . . . 14

Chapitre 1
Sujets des Travaux Pratiques sur Machine

SÉANCE 1

Processus Unix : interface de programmation POSIX

Conseils pour compiler les programmes C

Pour limiter le nombre d'erreurs à l'exécution, vous pouvez vous inspirer de l'exemple suivant :
`gcc -ansi -pedantic -Wall -Werror monprog.c -o monprog`
dans lequel "`-ansi -pedantic`" demande à `gcc` de vérifier que le programme respecte rigoureusement la norme ANSI, "`-Wall`" demande à `gcc` d'être plus scrupuleux dans ses vérifications et d'afficher des messages d'avertissement (*warnings*) dès qu'il a un doute et "`-Werror`" demande à `gcc` de considérer les *warnings* comme des erreurs et donc de ne pas produire le fichier exécutable s'il y a des *warnings*.

Pour éviter de devoir taper toutes ces options à chaque compilation, vous pouvez utiliser la commande `make` avec le fichier de définition `Makefile` dont vous trouverez un exemple sur Moodle. Après avoir recopié le fichier `Makefile` dans le répertoire de vos fichiers `C`, il suffit de taper `make monprog` pour compiler le fichier `monprog.c`

Exercices

Exercice 1

1. écrire une fonction qui affiche toutes les informations du processus courant (cf. Cours § 2.11.1, page 14).
2. écrire un programme permettant de tester cette fonction.
3. écrire une nouvelle version de ce programme de telle sorte que :
   - il crée un processus fils ;
   - il affiche les informations (en utilisant la fonction précédente) concernant le père et le fils ;
   - le père et le fils affichent un message juste avant de se terminer en indiquant leur code de retour ;
   - le père attende, avant de se terminer, la terminaison du fils et affiche le code de retour du fils.

Exercice 2

1. écrire un programme permettant de lancer l'exécution des commandes qui lui sont passées en paramètres. Pour chaque commande, le programme doit :
   - créer un processus fils qui doit lancer l'exécution de la commande grâce à la fonction `execvp` ;

- attendre que ce fils soit terminé ;
- passer à la commande suivante.

Des messages doivent être affichés à l'écran afin de suivre l'exécution du programme (voir exemple ci-dessous). Chaque message doit être précédé de l'identificateur du processus produisant cet affichage. Voici un exemple d'exécution dans lequel `lancer` est le nom de la version exécutable du programme demandé :

```
> lancer "sleep 5" pwd date
[1207] J'ai délégué sleep 5 à 1208. J'attends sa fin...
[1208] Je lance sleep 5 :
[1207] 1208 terminé.
[1207] J'ai délégué pwd à 1209. J'attends sa fin...
[1209] Je lance pwd :
/users/linfg/linfg0
[1207] 1209 terminé.
[1207] J'ai délégué date à 1210. J'attends sa fin...
[1210] Je lance date :
Thu Jan  7 19:43:35 MEST 2010
[1207] 1210 terminé.
[1207] J'ai fini.
>
```

La syntaxe de la fonction `execvp` (famille `exec`) est la suivante :

```
#include <unistd.h>
int execvp(const char *nom_fichier, char *const argv[])
```

où `argv` doit être un tableau de pointeurs de caractères semblable à celui qui est utilisé en paramètre de la fonction `main`, mais avec `NULL` comme dernier élément.

Afin de construire un tel tableau correspondant à une commande stockée sous la forme d'une chaîne de caractères, vous pouvez utiliser la fonction suivante (code source sur Moodle) :

```
#define NBMOTSMAX 20
/* Construction d'un tableau de pointeurs vers le début des mots d'une chaîne
 * de caractères en vue de l'utilisation de la fonction execvp.
 * Retourne le nombre de mots trouvés.
 */
int Decoupe(char Chaine[], char *pMots[])
{
  char *p;
  int NbMots=0;

  p=Chaine; /* On commence par le début */
  /* Tant que la fin de la chaîne n'est pas atteinte et qu'on ne déborde pas */
  while ((*p)!='\0' && NbMots<NBMOTSMAX)
  {
    while ((*p)==' ' && (*p)!='\0') p++; /* Recherche du début du mot */
    if ((*p)=='\0') break; /* Fin de chaine atteinte */
    pMots[NbMots++]=p;     /* Rangement de l'adresse du 1er caractère du mot */
    while ((*p)!=' ' && (*p)!='\0') p++; /* Recherche de la fin du mot */
```

```
    if ((*p)=='\0') break; /* Fin de chaîne atteinte */
    *p='\0';               /* Marquage de la fin du mot */
    p++;                   /* Passage au caractère suivant */
  }
  pMots[NbMots]=NULL;      /* Dernière adresse */
  return NbMots;
}
```

Par exemple, si la commande est stockée dans le tableau de caractères `Chaine` sous la forme d'une chaîne de caractère (donc terminée par le caractère `'\0'`), alors il suffit de déclarer un tableau de pointeurs de caractères :
`char *pMots[NBMOTSMAX+1];`
d'appeler la fonction :
`Decoupe(Chaine,pMots);`
et de lancer l'exécution avec :
`execvp(pMots[0],pMots);`

2. écrire une nouvelle version du programme précédent dans laquelle les processus fils sont tous créés (le père n'attend pas la terminaison d'un fils pour créer le suivant) et, ensuite, le père attend la terminaison de chacun de ses fils.

SÉANCE 2

Processus Unix : programmation multi-processus

Exercice 3

Étant données une matrice de nbL lignes et nbC colonnes et deux valeurs $v1$ et $v2$, on veut des statistiques sur son contenu, en comptant le nombre de lignes contenant uniquement $v1$, de lignes contenant uniquement $v2$ et de lignes contenant à la fois $v1$ et $v2$.

nbL, nbC, $v1$ et $v2$ sont des paramètres d'activation du $main()$.

On se propose de comparer la mise en œuvre de ce traitement en le confiant à un seul processus puis à plusieurs processus s'exécutant en parallèle.

On programmera tout d'abord la fonction suivante :

```
unsigned traiterLigne(int uneLigne[], unsigned nbC, unsigned v1, unsigned v2);
```

qui réalise le traitement d'une ligne *uneLigne* contenant nbC colonnes et retourne une valeur précisant si cette ligne contient $v1$, $v2$ ou $v1$ et $v2$.

On supposera que le nombre de lignes est inférieur à `NB_LIGNES_MAX` et que le nombre de colonnes est inférieur à `NB_COLONNES_MAX`.

1. ***Version 1 : Un processus***

   Le processus doit :
   - Initialiser la matrice en utilisant la fonction `initialiserMatrice` fournie :
     ```
     void initialiserMatrice(int matrice[NB_LIGNES_MAX][NB_COLONNES_MAX],
                             unsigned nbLignes, unsigned nbColonnes);
     ```
   - Traiter les *nbLignes* lignes et afficher pour chacune d'elles le résultat de son analyse, selon le format suivant :
     `numéro_de_ligne:code_réponse`
     où `code_réponse` aura pour valeur :
     - 0 pour « non trouvé »,
     - 1 pour « v1 trouvé »,
     - 2 pour « v2 trouvé »,
     - 3 pour « v1 et v2 trouvés ».
   - Afficher, lorsque toutes les lignes ont été traitées, les statistiques demandées, selon le format :
     `total : nbl_aucun_trouvé nbl_V1_trouvé nbl_V2_trouvé nbl_V1_V2_trouvés`
     (où « nbl » est le nombre de lignes où les valeurs ont été trouvées ou non).

   Exécutez votre programme plusieurs fois. Commentez les résultats obtenus.

2. ***Version 2 : plusieurs processus***

   On sous-traite à nbL processus le traitement de chacune des lignes. Le processus `sousTraitant` i traite la ligne i et fournit un compte-rendu d'exécution précisant si cette ligne contient $v1$, $v2$ ou $v1$ et $v2$.

   Le processus père doit quant à lui :
   - Initialiser la matrice en utilisant la fonction `initialiserMatrice` fournie.
   - Sous-traiter l'analyse des nbL lignes à un processus.
   - Afficher les résultats de l'analyse des lignes, au fur et à mesure qu'ils sont obtenus, selon le même format que pour la version 1.
   - Afficher, lorsque toutes les analyses sont terminées, les statistiques demandées, selon le même format que pour la version 1.

Exécutez votre programme plusieurs fois. Commentez les résultats obtenus.

3. ***Comparaison des deux versions***

   La commande `time` permet d'exécuter une commande et d'afficher ses temps d'exécution *real* (temps total d'exécution), *user* (code utilisateur) et *sys* (appels système) : `time commande`

   ```
   $ time ls
   ...
   real 0m0.005s
   user 0m0.002s
   sys 0m0.002s
   ```

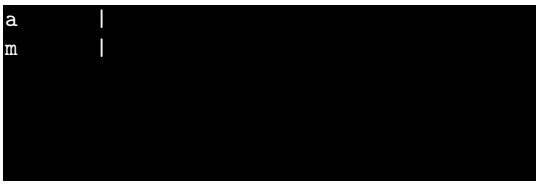
   En utilisant les deux versions du programme et la commande `time`, commentez les résultats obtenus par plusieurs exécutions mono-processus et multi-processus. Justifiez en particulier le temps consacré aux appels système.

SÉANCE 3

Processus Unix : programmation parallèle et communication avec le terminal

Introduction

On désire concevoir un jeu de course entre plusieurs joueurs sur le même terminal. Chaque joueur dispose d'une lettre (un caractère alphabétique) le représentant et peut faire avancer sa lettre sur le terminal en tapant sur la touche correspondant sur le clavier. À chaque frappe de la touche, la lettre avance d'une case sur sa ligne vers la droite. Par exemple, s'il y a deux joueurs `a` et `m`, le terminal affiche au départ :



puis lorsque le joueur `a` aura frappé la touche `a` au clavier, le terminal affichera – *le joueur* `a` *ayant avancé sa lettre d'une case* :

Le caractère `|` (pipe) représente la ligne d'arrivée. Tous les joueurs jouent en même temps sur le même terminal et donc frappent leur touche sur le même clavier.

Pour réaliser ce programme, il est nécessaire d'une part de lire une touche frappée au clavier sans que celle-ci s'affiche et sans attendre la fin d'une ligne (frappe de la touche `Entrée`), et d'autre part d'afficher des caractères à la position désirée sur le terminal. La lecture au clavier se fait en modifiant les paramètres de communication du terminal, notamment en supprimant l'écho et en rendant cette communication non-canonique. L'affichage est fait en utilisant des séquences d'échappement (*suite de caractères particuliers*, cf. page 10) envoyées au terminal lors des affichages.

Exercices

Exercice 4

Écrivez un programme `Course_au_clavier` qui utilise deux fonctions (cf. page suivante et Moodle) pour passer du mode de communication canonique au mode non-canonique avec le terminal :

- mode canonique : communication du terminal en mode interactif, c-à-d : écho, les touches frappées au clavier sont affichées sur le terminal ; et entrée par ligne : la lecture se fait lorsqu'une ligne complète est saisie (terminée par le touche `Entrée`).
- mode non-canonique : l'écho est supprimé et la lecture se fait caractère par caractère sans attendre la fin de la ligne.

```
#define _POSIX_C_SOURCE 200809L /* Respecte la norme POSIX 200809 : IEEE Std 1003.1-2008 */
#include <sys/termios.h>        /* le terminal non canonique */

/* Term_non_canonique ================================
//  Permet de lire le clavier touche par touche, sans
// écho.
//===================================================*/

int Term_non_canonique ()
{
  struct termios      term;

  tcgetattr( fileno(stdin), &term);/* lit les flags du terminal dans term */
  term.c_lflag &= ~ICANON;         /* mode non-canonique */
  term.c_lflag &= ~ECHO;           /* supprime l'écho */
  term.c_cc[VMIN]  = 1;            /* nombre min de caractères */
  term.c_cc[VTIME] = 1;            /* latence (timeout) 1/10e de seconde (0: pas de latence) *
  if (tcsetattr( fileno(stdin), TCSANOW, &term) < 0) /* écrit les flags depuis term */
  {
    perror("Term_non_canonique: problème d'initialisation ");
    return 0;
  }
  return 1;
}

/* Term_canonique ====================================
//  Mode normal du clavier: lecture par ligne et écho.
//===================================================*/

int Term_canonique ()
{
  struct termios      term;
                             /* retour en mode ligne */
  tcgetattr( fileno(stdin), &term);/* lit les flags du terminal dans term */
  term.c_lflag |= ICANON;          /* mode canonique */
  term.c_lflag |= ECHO;            /* rétablit l'écho */
  if (tcsetattr( fileno(stdin), TCSANOW, &term) < 0) /* écrit les flags depuis term */
  {
    perror("Term_canonique: problème d'initialisation ");
    return 0;
  }
  return 1;
}
```

Le programme doit configurer le terminal en mode non-canonique au début et remettre le terminal en mode canonique en fin d'exécution. Ensuite, il doit créer deux processus fils qui appellent la fonction `void touche()`
Cette fonction lit le clavier touche par touche en lisant un `char` sur son entrée standard avec la primitive Unix `read` et affiche son `pid` suivi de ce caractère. Par exemple, dans le processus fils dont

le `pid` est `100` , si la touche '`a`' est frappée, la fonction affiche `100 a`
La fonction lit en boucle le clavier jusqu'à ce que la touche '`f`' soit frappée, et se termine (fin du processus fils).
Le processus père doit attendre la fin des processus fils pour se terminer à son tour.

Exercice 5

Remplacez le fonction `touche` de la question précédente par la fonction :

```
void joueur(char my_char, int ligne, int distance)
```

avec pour paramètres : `my_char` la lettre du joueur (différente pour chacun), la `ligne` du terminal où afficher la lettre et la `distance` de la course. Cette fonction s'occupe d'afficher la lettre du joueur sur sa ligne et de la faire avancer à chaque fois qu'elle lit cette lettre au clavier. Le processus se termine dès que le joueur a atteint l'arrivée.
Le processus père doit attendre la fin des processus fils et afficher la position de chacun à l'arrivée.

Contrôle de l'affichage : séquences d'échappement

Pour positionner le curseur d'affichage du terminal sur la ligne et la colonne voulues, il faut utiliser la séquence d'échappement suivante : `"\033[` *ligne* `;` *colonne* `H"`
par exemple pour afficher un `x` ligne 1, colonne 3, utilisez la suite d'instructions :

```
printf ("\033[%d;%dHx", 1, 3);
fflush (stdout);
```

Cette dernière instruction est nécessaire pour la prise en compte de la séquence d'échappement par le terminal.

Autres séquences d'échappement :

Éffacer tout le terminal :	`"\033[H\033[J"`
Éffacer la fin de la ligne :	`"\033[K"`
Cacher le curseur :	`"\033[?25l"`
Afficher le curseur :	`"\033[?25h"`

Temps de pause

Lors de l'exécution du programme, il est possible qu'un seul processus monopolise la lecture de tous les caractères au clavier. Afin de donner une chance aux autres joueurs, vous pouvez faire dormir votre processus un court laps de temps à chaque fois qu'il a fait une lecture du clavier, avec la fonction `nanosleep` qui prend en paramètre la structure `timespec` pour définir le temps de pause :

```
#include <time.h>              /* Pour nanosleep */

struct timespec micro_pause;
micro_pause.tv_sec = 0;        /* secondes */
micro_pause.tv_nsec = 1000;    /* nanosecondes */
nanosleep( &micro_pause ,NULL);
```

Exercice 6

Modifier votre programme pour qu'on puisse lui passer en paramètres le nombre de joueurs et la distance à parcourir. Par exemple, la commande `Course_au_clavier 3 6` permet de jouer à 3 joueurs sur une distance de 6 colonnes. On limitera le nombre de joueurs à 4, étant donné que tout le monde joue sur le même clavier !

Exercice 7

Pour les plus rapides ...

Ajouter un moniteur à votre programme. C'est un autre processus créé par votre programme qui s'occupe seul de l'affichage de tous les joueurs. Pour cela, les processus des joueurs n'affichent plus rien, mais à la place ils écrivent dans un fichier binaire leur position sur la ligne à chaque fois qu'ils avancent. Le premier joueur écrit sa position dans le premier entier du fichier, le second joueur dans le second... Attention lors de l'écriture, à se positionner au bon endroit dans le fichier pour ne pas écraser la position d'un autre joueur. Le processus moniteur lit dans ce fichier régulièrement pour mettre à jour l'affichage. Pour que tous les processus accèdent au même fichier, celui-ci doit être créé par le programme avant la création des fils. Au départ, ce fichier contient les positions des joueurs sur la première colonne, i.e. autant de 1 que de joueurs.

Attention : quand les processus père et fils accèdent au même fichier (même descripteur hérité par fork()), la position courante est modifiée pour tous (ce qui est un peu contre intuitif !). Donc si vous utilisez les primitives `lseek(...)` suivie de `write(...)` pour écrire la position du joueur dans le fichier, l'ordonnanceur peut avoir l'idée de préempter le processus entre les deux appels, ce qui rend la position de l'écriture imprévisible (idem pour la lecture). La solution consiste à utiliser des primitives qui réalisent dans le même appel à la fois le déplacement dans le fichier et l'écriture (ou la lecture) :

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

fonctionnent comme `read` et `write`, en déplaçant au préalable la position courante de `offset` octets par rapport au début du fichier.

SÉANCE 4

Gestion de fichiers : bibliothèque Unix

Introduction

On désire manipuler des fichiers binaires dans lesquels on stocke des fiches contenant des informations sur des personnes. Le type associé à une fiche sera le suivant :

```
#define LONG_MAX_NOM      20
typedef struct
{
  char Nom[LONG_MAX_NOM+1];
  int Age;
  int NbEnfants;
} Infos;
```

Exercices

Exercice 8

Création séquentielle du fichier
En utilisant les fonctions de la bibliothèque Unix, écrire une fonction d'en-tête :
`int Creation(char NomFichier[])`
qui effectue la création du fichier dont le nom est passé en paramètre à partir de données tapées au clavier. Les grandes étapes à suivre sont les suivantes :

- ouverture du fichier,
- saisie et écriture dans le fichier des différentes fiches,
- fermeture du fichier.

Cette fonction doit retourner un entier décrivant la manière dont la création s'est passée : `0` si tout s'est bien passé ou un entier positif décrivant le problème rencontré (utiliser des constantes symboliques).

Écrivez une fonction principale (`main`) qui vous permet de tester la fonction `Creation` et que vous ferez évoluer durant la séance.

Afin de vérifier le fonctionnement de votre fonction `Creation`, il peut être utile d'afficher le contenu d'un fichier binaire avec la commande `od` (n'hésitez pas à taper `man od`) : l'option `-c` permet d'afficher chaque octet du fichier comme un caractère ; l'option `-t d1` permet d'afficher chaque octet du fichier comme un entier en base 10.

Exercice 9

Consultation du fichier par accès direct
En utilisant les fonctions de la bibliothèque Unix, écrire une fonction d'en-tête :
`int Consultation(char NomFichier[])`
qui permet à l'utilisateur de consulter une ou plusieurs fiches contenues dans le fichier dont le nom est passé en paramètre. La fonction doit effectuer une boucle de consultation dans laquelle on demande à l'utilisateur le numéro de la fiche qu'il souhaite consulter (la première fiche portant le numéro 1) afin de se positionner directement au niveau de la fiche à lire dans le fichier et à afficher à l'écran. Les grandes étapes à suivre sont les suivantes :

- ouverture du fichier,
- affichage du nombre de fiches contenues dans le fichier,
- consultation en boucle (saisie du numéro d'une fiche, positionnement direct en début de fiche, lecture de la fiche, affichage du contenu de la fiche),
- fermeture du fichier.

Cette fonction doit retourner un entier ayant le même rôle que celui de l'exercice 8.

Exercice 10

Terminez l'écriture de la fonction principale (`main`) permettant de tester les deux fonctions précédentes en affichant un menu. On suppose que le nom d'un fichier contient au plus 30 caractères. Si un problème est survenu lors de l'exécution d'une fonction, le programme doit retourner au système d'exploitation le code retourné par la fonction appelée.

SÉANCE 5

Gestion de fichiers : bibliothèque Unix

Exercice 11

écrire un programme C qui affiche des informations sur les fichiers ou répertoires dont les noms sont passés en paramètres. Pour un fichier, les informations suivantes doivent être affichées sur une ligne : la désignation du fichier, la chaîne de caractères `fichier`, sa taille et la date et l'heure de dernière modification des données du fichier. Pour un répertoire, les mêmes informations doivent être affichées sur une ligne à l'exception de la chaîne de caractères `fichier` qui doit être remplacée par la chaîne `répertoire`.

Exemple :

```
> tp5ex11 tp5ex11.c TEST TEST/fich.txt
tp5ex11.c            : fichier        1093 octets Thu May 27 16:50:34 2021
TEST                 : répertoire      120 octets Thu May 27 14:36:52 2021
TEST/fich.txt        : fichier          33 octets Thu May 27 14:36:52 2021
```

Remarque : pour aligner les informations, vous pouvez utiliser les formats `%-20s` pour la désignation et `%8ld` pour le nombre d'octets (le caractère `-` permet d'obtenir un affichage cadré à gauche).

Exercice 12

Compléter le programme de l'exercice 11 pour que, dans le cas où aucun paramètre n'est passé au programme, il affiche les informations sur tous les fichiers et répertoires contenus dans le répertoire courant. En revanche, si des paramètres sont passés au programme, il doit avoir le même comportement que le programme de l'exercice 11.

Remarque : le répertoire courant peut être désigné par la chaîne de caractères `"."`.

Exemple :

```
> cd TEST
> ../tp5ex12
.                    : répertoire      120 octets Thu May 27 14:36:52 2021
..                   : répertoire      752 octets Thu May 27 17:18:35 2021
REP1                 : répertoire      112 octets Thu May 27 14:37:53 2021
REP2                 : répertoire       80 octets Thu May 27 14:38:09 2021
fich.txt             : fichier          33 octets Thu May 27 14:36:52 2021
```

Exercice 13

En vous inspirant de la fonction de l'exercice 15, page 31 du support de cours, écrire un programme qui affiche les nombres d'octets de tous les fichiers contenus dans la sous-arborescence dont le répertoire racine est passé en paramètre. Cet affichage devra être de la forme suivante :

```
> cd ..
> tp5ex13 TEST
TEST/REP1/fich11.txt                    :       39 octets
TEST/REP1/fich12.txt                    :       56 octets
    Total TEST/REP1                     :       95 octets
TEST/REP2/fich2.txt                     :       74 octets
    Total TEST/REP2                     :       74 octets
TEST/fich.txt                           :       33 octets

        TOTAL TEST                      :      202 octets
```