TRAVAUX PRATIQUES 1
Langages de commande Linux
Processus Linux

L'objectif de ce TP est de voir des commandes utiles fournit par un système d'exploitation de type UNIX, puis les outils et fonctions proposées par un système d'exploitation pour la compilation et l'exécution de programmes.

Les documents associés à ce TP sont disponibles à l'adresse ci-dessous :
`http://deptinfo.cnam.fr/new/spip.php?rubrique138`

La commande pour extraire l'archive TP1_sources.zip est `unzip`. Cette commande est à taper dans un terminal, dans votre répertoire de travail, là où vous enregistrerez l'archive.

EXERCICE 1

Dans cet exercice nous allons étudier plusieurs commandes utiles pour travailler sous un système de type Unix/Linux. Ces commandes sont entrées dans une fenêtre appelée terminal qui les interprète et les exécute.
Une fois votre session ouverte, ouvrez un terminal pour entrer les commandes décrites dans la suite de cet exercice.

1) Il est essentiel de connaître la commande `man` fournissant un descriptif détaillé sur l'utilisation d'une commande passée en paramètre, appuyez sur la touche **q** pour quittez le manuel. N'hésitez donc pas à aller consulter cette documentation qui est un véritable aide-mémoire.

   Question 1 – Entrez la commande dans un terminal comme donné ci-dessous. Expliquez ce que retourne la commande `pwd`.

   ```
   $> man pwd
   ```

   Question 2 – Tapez la commande `pwd` dans un terminal et indiquez quel est le résultat retourné par la commande.

2) Vous pouvez créer un nouveau répertoire à l'aide de la commande `mkdir` (pour *make directory* en anglais) suivie du nom du répertoire à créer. Si vous indiquez un chemin (valide dans l'arborescence) devant le nom de répertoire alors le répertoire sera créé à l'endroit indiqué, sinon le répertoire sera créé par défaut dans le répertoire courant.

   Question 3 – Créez un répertoire de travail nommé *rep_travail* dans le répertoire courant, pour cela entrez la commande suivante :

   ```
   $> mkdir rep_travail
   ```

3) Vous pouvez afficher le contenu d'un répertoire à l'aide de la commande `ls` (pour *list* en anglais). Cette commande prend plusieurs paramètres, parmi ceux-ci il y a le chemin du répertoire dont il faut lister le contenu.

Question 4 – Entrez la commande suivante pour lister le contenu du répertoire que vous venez de créer :

```
$> ls rep_travail
```

Comme dans la majorité des systèmes d'exploitation, les fichiers et répertoires disposent de plusieurs attributs comme par exemple le nom du fichier, le type de fichier, les droits d'accès, le nom du propriétaire, la taille, la date de dernière modification.
L'exemple ci-dessous montre des informations fournit par l'exécution de la commande `ls –alh` d'un répertoire :

```
$> ls –alh
```

```
total 32K
drwxr-xr-x   2   steph  steph   4,0K  nov. 24 19:18   .
drwxr-xr-x   6   steph  steph   4,0K  oct.  3 15:45   ..
-rwxr-xr-x   1   steph  steph   7,2K  juin 10 11:28   ecr
-rw-r--r--   1   steph  steph   410   juin 10 11:28   ecr.c
-rwxr-xr-x   1   steph  steph   7,2K  juin 10 11:27   lec
-rw-r--r--   1   steph  steph   359   juin 10 11:27   lec.c
```

La première ligne indique la taille totale des fichiers et répertoires contenus dans le répertoire courant listé, ici 32 kilo-octets (multiple de 4 ko car les données sont stockées dans des blocs disque de taille 4 Ko).
Les informations fournies ensuite sont regroupées par colonnes :

- **Première colonne (premier caractère) :** indique le type fichier (symbole `-`) ou répertoire (lettre `d`). Il existe d'autres types de fichiers utilisés par le système.
- **Première colonne (autres caractères) :** indique les droits d'accès au fichier ou répertoire. Il y a trois ensembles de trois lettre (`r` pour le droit en lecture, `w` pour le droit en écriture, `x` pour le droit en exécution). Le symbole `–` indique que le droit correspondant à la position n'est pas autorisé. Le premier triplet correspond aux droits du propriétaire (ayant créé le fichier en général), le deuxième triplet correspond au groupe auquel appartient le propriétaire et enfin le dernier triplet correspond aux autres utilisateurs du système.
- **Deuxième colonne :** indique le nombre de références vers cette entrée.
- **Troisième colonne :** nom du propriétaire du fichier.
- **Quatrième colonne :** nom du groupe auquel appartient le propriétaire.
- **Cinquième colonne :** taille du fichier (un répertoire possède la taille d'un bloc suffisant en général pour stocker la liste et les attributs des fichiers et répertoire qu'il contient).
- **Sixième colonne :** date de création ou de dernière modification du fichier.
- **Septième colonne :** nom du fichier ou du répertoire.

La deuxième et la troisième ligne sont particulières car elles fournissent des informations respectivement sur le répertoire courant et le répertoire parent. Tandis que les lignes suivantes indiquent les informations associées à chaque fichier ou sous-répertoire contenu dans le répertoire courant.

Question 5 – Indiquer les différents types de fichiers contenus dans le répertoire courant listés par la commande `ls` ci-dessus.

Question 6 – Pourquoi le droit en exécution n'est pas disponible pour les fichiers `ecr.c` et `lec.c` ?

4) Il est possible de changer de répertoire courant en utilisant la commande `cd` (pour *change directory* en anglais) suivie du chemin du répertoire à atteindre dans l'arborescence. Pour remonter d'un niveau dans l'arborescence de fichiers il faut utiliser le symbole « `..` » faisant référence au répertoire parent.

   Question 7 – Entrez la commande suivante pour changer de répertoire courant et aller dans le répertoire que vous avez créé précédemment :

   ```
   $> cd rep_travail
   ```

5) La commande `touch` permet de créer fichier dont le nom est indiqué en paramètre.

   Question 8 – Tapez la commande suivante pour créer un fichier de nom *newfich* dans le répertoire courant :

   ```
   $> touch newfich
   ```

   Question 9 – Il est possible d'éditer sur le terminal le contenu d'un fichier. Pour cela vous pouvez utiliser la commande `less`. Il est aussi possible d'utiliser la commande `more`, mais celle-ci est plus lente pour l'édition de gros fichiers (nécessite la lecture de fichier complet avant édition). Pour quitter le mode édition de la commande `less`, vous devez taper sur la touche **q** du clavier (pour *quit* en anglais).
   Tapez la commande suivante pour éditer le fichier de nom *newfich* du répertoire courant et indiquez le contenu du fichier :

   ```
   $> less newfich
   ```

   Question 10 – Tapez la commande suivante pour ajouter dans le fichier *newfich* le message indiqué entre les guillemets :

   ```
   $> echo "test message" > newfich
   ```

   L'opérateur `>` permet de rediriger un flux d'information vers un fichier (en écrasant son contenu), tandis que l'opérateur `>>` permet d'ajouter en fin de fichier un flux d'information sans écraser son contenu.

   Question 11 – Utilisez la commande `echo` ainsi que l'opérateur `>>` pour ajouter le texte de votre choix en fin du fichier *newfich*.

   Question 12 – Utilisez à nouveau la commande `less` pour éditer le fichier *newfich*. Quel est le contenu du fichier *newfich* ?

6) Il peut être utile d'avoir des informations associées au stockage d'un fichier sur le disque. La commande `stat` fournit des informations sur l'état d'un fichier ou d'un système de fichiers.
   Ainsi sur l'exemple donné ci-dessous, la commande `stat` fournit plusieurs informations sur le fichier `log.txt` :
   - il a une taille de `42973316` octets,

- il est constitué de `83936` blocs disque,
- la taille d'un bloc disque est de 4096 octets,
- le numéro d'i-nœud associé au fichier est `133824`.

```
$> stat log.txt

File: `log.txt'
Size: 42973316 Blocks: 83936 IO Block: 4096 regular file
Device: 806h/2054d Inode: 133824 Links: 1
Access: (0644/-rw-r--r--) Uid: (1000/ steph) Gid: (1000/ steph)
Access: 2014-10-22 22:41:22.000000000 -0400
Modify: 2014-10-23 03:22:34.000000000 -0400
Change: 2014-10-23 03:22:34.000000000 -0400
```

Question 13 – Utilisez la commande `stat` et indiquez les quatre informations données ci-dessous associées au fichier pdf de l'énoncé de TP.

Dans l'exemple ci-dessous, la commande `stat` est utilisée afin d'avoir des informations sur le système de fichiers qui stocke le fichier `log.txt`.
Plusieurs informations sont fournit par la commande :

- le système de fichiers est de type Ext3,
- la taille maximale du nom d'un fichier est de 255 caractères,
- la taille d'un bloc est de 4096 octets,
- le système de fichiers est constitué au total de `2388558` blocs dont sont `593795` blocs libres (parmi ces blocs `470506` sont libre peuvent être utilisées par des utilisateurs différent de l'utilisateur *root*),
- le système de fichiers peut stocker au maximum jusqu'à `2466048` i-nœuds (fichiers ou répertoires) et actuellement il est possible de créer encore `2126726` nouveaux i-nœuds.

```
$> stat -f log.txt

File: "log.txt"
ID: 0 Namelen: 255 Type: ext2/ext3
Block size: 4096
Blocks: Total: 2388558 Free: 593795 Available: 470506
Inodes: Total: 2466048 Free: 2126726
```

Question 14 – Utilisez la commande `stat` et indiquez les informations données ci-dessous associées au système de fichiers stockant l'énoncé de TP.

7) La commande `chmod` permet de changer les droits d'accès d'un fichier ou répertoire. Il est possible de modifier les droits associés soit au propriétaire (option **u**), le groupe du propriétaire (option **g**), les autres utilisateurs (option **o**) ou les trois types d'utilisateurs à la fois (option **a**). Pour cela, étant donné le groupe d'utilisateurs cible il faut indiquer l'ajout (symbole +) ou la suppression (symbole -) de chaque type d'accès lecture (**r**), écriture (**w**) ou exécution (**x**).

Question 15 – Exécutez la commande suivante qui permet d'ajouter le droit en écriture aux utilisateurs appartenant au groupe du propriétaire du fichier *newfich* :

```
$> chmod g+w newfich
```

Question 16 – Donnez la commande permettant d'ajouter le droit en écriture au fichier *newfich* à tous les utilisateurs.

Question 17 – Donnez la commande permettant de retirer le droit en lecture au fichier *newfich* aux utilisateurs n'étant ni le propriétaire du fichier et n'appartenant pas au groupe du propriétaire.

EXERCICE 2

Soit le programme `exo2.c` donné ci-dessous. Ce programme créé un processus fils, de type *processus lourd* car les deux processus (processus père et fils) sont des copies parfaites (copie identique des segments de code et de données, mais les segments de données du processus père et fils sont distincts). Nous rappelons qu'un processus est la représentation de la dynamique d'exécution d'un programme (ou partie d'un programme si celui-ci créé durant son exécution plusieurs autres processus fils).

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

int main(void)
{
     int pid, i;
     pid = fork();
     if(pid == 0){
        for(i=0;i<10;i++){
           printf("je suis le fils\n");
           sleep(5);
        }
     }
     else{
        for(i=0;i<10;i++){
           printf("je suis le père\n");
           sleep(5);
        }
     }
     return 1;
}
```

Question 1 – Téléchargez le fichier `exo2.c` et compilez le programme `exo2.c` à l'aide de la commande ci-dessous.

```
$> gcc –c exo2.c
```

Qu'obtenez-vous comme résultat de la compilation ? Quelle est l'étape suivante ? Tapez ensuite la commande ci-dessous :

```
$> gcc –o exo2 exo2.o
```

Question 2 – La commande `ps -ux` permet d'afficher les caractéristiques de l'ensemble des processus de l'utilisateur et donne les éléments suivants:

```
F   S UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY    TIME    CMD
100 S  0   467   1    0  60   0 - 256 read_c tty5   00:00:00 mingetty
100 S  0   468   1    0  60   0 - 256 read_c tty6   00:00:00 mingetty
100 S  0   576   570  0  60   0 - 498 read_c pts/0  00:00:00 cat
100 S  0   580   578  0  70   0 - 576 wait4  pts/1  00:00:00 bash
100 S  0   581   579  0  60   0 - 77  wait4  pts/2   00:00:00 bash
000 S  0   592   581  2  61   0 - 253 down_f pts/2  00:00:01 essai
040 S  0   593   592  2  61   0 - 253 write_ pts/2  00:00:01 essai
100 R  0   599   580  0  73   0 - 652 -      pts/1  00:00:00 ps
```

Les champs `S, PID, PPID` et `CMD` codent respectivement l'état du processus (S pour *Stopped*, R pour *Running*), la valeur du PID et du PPID pour le processus et le nom du programme exécuté.
La commande `kill -9 593` entraîne la terminaison du processus dont le PID 593 est spécifié en argument.

Lancez l'exécution du programme `exo2` dans une fenêtre, à l'aide de la commande suivante :

```
$>./exo2
```

puis tapez la commande `ps -ux` dans une autre fenêtre. Quelle est l'arborescence de processus liée aux processus `exo2` depuis votre shell.

1. Quelle commande tapez-vous pour détruire le fils `exo2` ? Quel est son état ?
2. Quelle commande tapez-vous pour détruire le père `exo2` ?

Question 3 – Le programme `exo2.c` est modifié comme ci-dessous :

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
     int pid, i;
     pid = fork();
     if(pid == 0){
        for(i=0;i<10;i++){
           printf("je suis le fils\n");
         }
     }
     else{
        printf("je suis le père\n");
        wait();
     }
     return 1;
}
```

1. Modifiez le fichier `exo2.c` comme ci-dessus. Pour cela, vous pouvez utiliser l'éditeur de texte `kwrite` en exécutant la commande ci-dessous dans le répertoire contenant le fichier `exo2.c` :

   ```
   $>kwrite exo2.c &
   ```

   On recompile ce programme pour générer un nouvel exécutable appelé `exo2bis` dont on lance l'exécution.

   Note : le symbole `&` placé en fin de commande permet lancer l'exécution de la commande en tâche de fond, ainsi le terminal vous redonne la main pour entrer une nouvelle commande.

2. Quelle commande tapez-vous pour détruire le fils `exo2bis` ? Quel est son état ?

Question 4 – Le programme `exo2bis.c` est modifié comme ci-dessous :

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
     int pid;
     pid = fork();
     if(pid == 0){
        printf("je suis le fils et vais exécuter la commande ls\n");
        execlp("ls", "ls", "-l", NULL);
     }
     else{
        printf("je suis le père\n");
        wait();
     }
     return 1;
}
```

Dans ce programme modifié, le fils utilise la fonction `execlp` afin d'exécuter un programme exécutable, ici la commande `ls -l`. Il existe plusieurs variantes de la fonction `exec` qui diffèrent de la façon dont les arguments sont passés à la fonction. Ici, les arguments sont passés sous forme de liste (**l** pour *list* en anglais) et la variable d'environnement est utilisée pour le chemin d'accès vers le programme exécutable à exécuter (**p** pour *path* en anglais).

Modifiez le code du programme `exo2bis.c` et recompilez ce programme pour générer un nouvel exécutable appelé `exo2tier`. Puis, lancez son exécution afin de vérifier son bon fonctionnement.

EXERCICE 3

Dans cet exercice, nous allons nous intéresser aux *processus légers* (ou *thread* en anglais). Contrairement à un processus lourd, les processus légers créés par le même processus parent partagent le même espace mémoire que celui-ci, même s'ils peuvent exécuter un segment de code différent.
Ce type de processus permet d'avoir une empreinte mémoire plus faible que l'utilisation de processus lourds. De plus, comme les threads partagent le même espace mémoire il est plus facile de les faire communiquer. En revanche, il est primordial de contrôler l'accès à la mémoire sans quoi certaines exécutions peuvent devenir incohérentes, à cause de l'accès concurrent par plusieurs processus à une même partie de la mémoire.

Question 1 – Soit le programme `exo3.c` suivant :

```
#include <stdio.h>
#include <unistd.h>

int i;

int main(void)
{
     int pid;

     i = 0;
     pid = fork();
     if (pid == 0){
        i = i + 10;
        printf ("hello,  fils %d\n", i);
        i = i + 20;
        printf ("hello, fils %d\n", i);
     }
     else{
        i = i + 1000;
        printf ("hello, père %d\n", i);
        i = i + 2000;
        printf ("hello, père %d\n", i);
        wait();
     }
     return 1;
}
```

Téléchargez le fichier `exo3.c` puis compilez et exécutez le programme exécutable obtenu. Quelles traces génère l'exécution de ce programme ?

Peut-on obtenir une trace différente lors d'une nouvelle exécution du programme ? Testez en relançant plusieurs fois ce programme.

Question 2 – Le programme `exo3.c` est modifié comme suit :

```
#include <stdio.h>
#include <pthread.h>

int i;

void addition()
{
     i = i + 10;
     printf ("hello, thread fils %d\n", i);
     i = i + 20;
     printf ("hello, thread fils %d\n", i);
}

int main(void)
{
     pthread_t num_thread;

     i = 0;
     if(pthread_create(&num_thread, NULL, (void *(*)())addition, NULL) ==
     -1)
          perror ("pb pthread_create\n");
     i = i + 1000;
     printf ("hello, thread principal %d\n", i);
     i = i + 2000;
     printf ("hello, thread principal %d\n", i);
     pthread_join(num_thread, NULL);
     return 1;
}
```

Modifiez le fichier `exo3.c` comme ci-dessus, puis compilez en utilisant l'option `-lpthread` et exécutez le programme exécutable `exo3bis` obtenu.
Quelles traces génère l'exécution de ce programme ?

Peut-on obtenir une trace différente lors d'une nouvelle exécution du programme ? Testez en relançant plusieurs fois ce programme.

EXERCICE 4

Dans cet exercice, nous allons nous intéresser à la concurrence d'accès à la mémoire par des processus légers.

Soit le programme suivant :

```
#include <stdio.h>
#include <pthread.h>

int tab_heure[4];

void heure()
{
     while(1) {
           tab_heure[0] = (tab_heure[0] + 1)%60;

           if (tab_heure[0] == 0)
           {
              tab_heure[1] = (tab_heure[1] + 1)%60;
           }
           if (tab_heure[1] == 0)
           {
             tab_heure[2] = (tab_heure[2] + 1)%24;
           }
           if (tab_heure[2] == 0)
           {
             tab_heure[3] = tab_heure[3] + 1;
           }
           sleep(1);
     }
}

void main(void)
{
     pthread_t num_thread;

     tab_heure[0]=1;
     tab_heure[1]=4;
     tab_heure[2]=5;
     tab_heure[3]=2;

     if(pthread_create(&num_thread, NULL, (void *(*)())heure, NULL) == -1)
        perror("pb pthread_create\n");

     while(1)
     {
        printf("L'heure du système est : %d jour,%d heure,%d minutes,%d
     secondes\n", tab_heure[3], tab_heure[2], tab_heure[1], tab_heure[0]);

       sleep(5);
     }

}
```

Question 1 – A votre avis, la synchronisation dans ce code est-elle correcte ?
Supposons par exemple que le tableau `tab_heure` contienne les valeurs suivantes :

```
tab_heure[0] = 59     // seconde
tab_heure[1] = 59     // minute
tab_heure[2] = 2      // heure
tab_heure[3] = 4      // jour
```

et que les deux threads père et fils soient lancés en même temps. L'heure affichée par le thread parent est elle toujours juste ?

Question 2 – Que doit-on ajouter à ce programme pour que le problème identifié ci-dessus ne se produise pas.

Nous vous fournissons un fichier `sem_func.c` à télécharger implémentant les fonctions nécessaires pour la synchronisation des processus, ainsi que le fichier `ex_synchro.c` ci-dessus également à télécharger.
Trois fonctions sont disponibles et implémentées dans le fichier `sem_func.c` :

- `init_verrou()` : cette fonction permet d'initialiser la structure de synchronisation,
- `prendre_verrou()` : cette fonction est à appeler à l'entrée de la section critique,
- `rendre_verrou()` : cette fonction est à appeler en sortie de la section critique.

Question 3 – Identifiez la section critique et utilisez ces trois fonctions pour intégrer une synchronisation correcte dans le programme donné ci-dessus comme demandé à la Question 2.

Question 4 – Afin d'exécuter le programme, réalisez une compilation séparée pour générer le module objet associé aux différents fichiers fournis comme indiqué ci-dessous :

```
$> gcc –lpthread –c ex_synchro.c
$> gcc –c sem_func.c
$> gcc –lpthread –o ex_synchro ex_synchro.o sem_func.o
```

Les deux premières commandes permettent de générer respectivement les codes objets `ex_synchro.o` et `sum_func.o`, tandis que la dernière commande permet de réaliser l'édition des liens du programme et générer l'exécutable `ex_synchro`.

L'outil Make

L'outil `make` exploite les dépendances existantes entre les modules entrant en jeu dans la construction d'un programme exécutable pour ne lancer que les opérations de compilations et éditions de liens nécessaires, lorsque ce programme exécutable doit être reconstruit suite à une modification intervenue dans les modules sources.
Pour cela, `make` utilise deux sources d'informations : un fichier de description appelé le *Makefile* qui contient la description des dépendances entre les modules, et les noms et les dates de dernières modifications des modules.

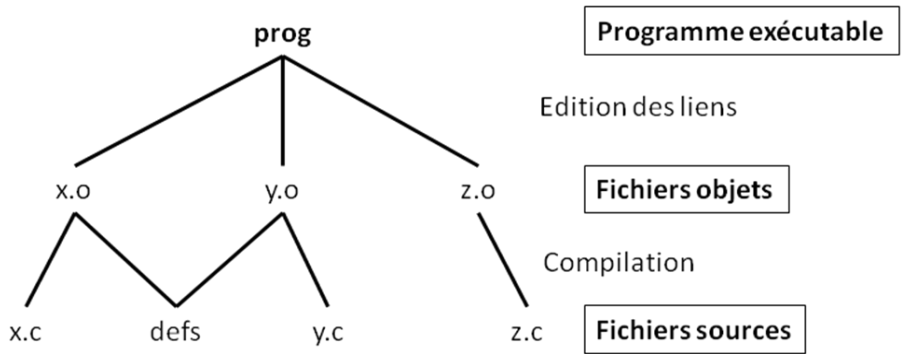
Format du fichier Makefile

Le fichier *Makefile* décrit les dépendances existantes entre les modules intervenant dans la construction d'un exécutable. Il traduit sous forme de règles le graphe de dépendance du programme exécutable à construire et indique pour chacune de ces dépendances, l'action qui lui est associée.

Une règle dans le fichier *Makefile* est de la forme :

```
module cible :  dépendances
     commande pour construire le module cible
```

Prenons comme exemple le cas suivant, le programme exécutable `prog` est construit à partir d'une étape d'édition des liens prenant en compte les trois modules objets `x.o`, `y.o` et `z.o`. Le module objet `z.o` est issu de la compilation d'un programme source `z.c`. Les modules `x.o` et `y.o` sont à leur tour issus de la compilation respective des modules source `x.c` et `y.c`. Ces deux derniers modules utilisent un module `defs` par le biais d'un ordre d'inclusion `#include`.
Le graphe de dépendance du programme `prog` est donné sur la figure ci-dessous.



Le fichier *Makefile* résultant est :

```
prog : x.o y.o z.o
     gcc –o prog x.o y.o z.o
x.o : defs x.c
     gcc –c x.c
y.o : defs y.c
     gcc –c y.c
z.o : z.c
     gcc –c z.c
```

La première règle stipule la dépendance associée au programme exécutable `prog` qui est donc construit à partir des modules `x.o`, `y.o` et `z.o`. La commande permettant la construction du programme exécutable `prog` à partir de ces trois modules objets est la commande d'édition des liens `gcc –o prog x.o y.o z.o`.
La seconde et la troisième règle stipulent que le fichier objet `x.o` (respectivement `y.o`) dépend à la fois du fichier `x.c` (respectivement `y.c`) et du fichier `defs`. Les fichiers `y.o` ou `x.o` sont construits par compilation des fichiers sources `.c` correspondants.
Enfin, la dernière règle spécifie que le fichier `z.o` dépend uniquement de la compilation du fichier `z.c`.

Fonctionnement de l'utilitaire Make

L'outil `make` utilise le fichier *Makefile* et les dates de dernières modifications des modules pour déterminer si un module est à jour.
Un module est à jour si le module existe et si sa date de dernière modification est plus récente ou égale aux dates de dernière modification de tous les modules dont il dépend. Si un module n'est pas à jour, la commande associée à ses dépendances est exécutée pour reconstruire le module.

Considérons par exemple que le module `z.c` soit modifié. L'utilitaire `make` va détecter que ce module est devenu plus récent que le module objet `z.o`. Il va donc lancer la commande associée à la règle de dépendance du module `z.o`, soit la commande de compilation `gcc – c z.c`. L'exécution de cette commande va à son tour générer un module `z.o` plus récent que le programme exécutable `prog`. En conséquence, l'utilitaire `make` va reconstruire le programme exécutable `prog` en lançant la commande d'édition des liens `gcc –o prog x.o y.o z.o`. D'une façon similaire, toute modification au sein du module `defs` entraînera la reconstruction des modules `y.o` et `x.o`, par le biais de deux opérations de compilation, puis la reconstruction du programme exécutable `prog`. Dans ces deux cas, seules les opérations de compilation ou d'édition des liens nécessaires sont exécutées.
L'utilitaire `make` est appelé au moyen de la commande `make prog` qui suppose qu'un fichier *Makefile* est présent dans le répertoire où la commande est lancée.

Question 5 – En suivant les indications données ci-dessus, réalisez un *Makefile* contenant les règles de compilation et d'édition des liens afin d'obtenir le programme exécutable `ex_synchro`. Le programme `ex_synchro` est généré à partir des modules objets `ex_synchro.o` et `sem_func.o`, obtenus respectivement à partir de la compilation des fichiers sources `ex_synchro.c` et `sem_func.c`.
Vous pourrez pour cela vous inspirer des commandes données à la Question 4.