Révision du C

Exercice 1

Le main d'un programme en C est en général de la forme suivante :
*int main (int argc, char * argv[])*
argv[] contient l'ensemble des arguments qui ont été passés sur la ligne de commande.
Ecrire un programme qui liste ses arguments
exemple

```
 ./testMain tat aaaa d
l'arg 0 est ./testMain
l'arg 1 est tat
l'arg 2 est aaaa
l'arg 3 est d
```

Exercice 2

Une variable d'environnement est une variable qui est manipulé à l'intérieur du shell.. Il s'agit par exemple de PATH, ou de PS1. Cette variable peut être locale (interne à ce processus). Le mot clé du shell **export** permet de la rendre accessible aux processus fils
La primitive système **getenv(char *)** vous permet de récupérer son contenu dans un programme en C.
Definissez quelques variables d'environnement, et essayez de les afficher grâce à un programme en C.
exemple

```
/**Ici, trouvez un moye n pour créer les variables ENV1 et ENV2 accessibles aux fils /**
 ./testMain2
ENV1 vaut coucou
ENV2 vaut coucou2
```

Exercice 3

Ecrire un programme qui affiche la table de multiplication d'un nombre qui aura été passé en argument du programme.

Exercice 4

Ecrire une fonction min à laquelle on passe deux entiers et qui renvoie le plus petit des deux.
Utilisez cette fonction dans un programme

Exercice 5 printf et scanf

Ecrire un programme permettant de saisir une chaine (long maxi 255 car), puis qui affiche le contenu de cette chaine. Écrire une deuxième version du programme dans laquelle la saisie est faite dans une fonction.

Exercices sur les primitives systèmes fork exec wait exit.

Exercice 1

Ecrivez un programme qui récupére les programmes dont le nom est passé en arguments sur la ligne de commandes, puis les exécute dans des processus fils.
exemple

```
./testMain3  who uname ps
je suis le fils chargé de faire who
sylvain  tty9         2008-05-16 15:17 (:0)
sylvain  pts/1        2008-05-16 17:20 (:1.0)
16285 se charge du boulot
je suis le fils chargé de faire uname
Linux
16286 se charge du boulot
je suis le fils chargé de faire ps
  PID TTY          TIME CMD
 6304 pts/1    00:00:00 bash
16284 pts/1    00:00:00 testMain3
16285 pts/1    00:00:00 who <defunct>
16286 pts/1    00:00:00 uname <defunct>
16287 pts/1    00:00:00 ps
16287 se charge du boulot
```

Exercice 2

Chaque fils doit maintenant attendre avant d'éxécuter sa tache, selon une valeur défini dans une variable d'environnement ATTENTE. Même résultat qu'au dessus, avec une attente...

Exercice 5
Si vous définissez votre variable d'attente à 0, et que vous lancez un ps en dernier de la liste, que remarquez vous dans la réponse de ce ps ?
Le problème est lié au fait que les fils veut signaler à leur père qu'ils ont terminé, mais celui-ci les ignore. Pour éviter le problème, on peut utiliser la primitive wait().
Le code de la partie père devient le suivant :

```
} else {
                        //on est le père
                        printf("%i se charge du boulot\n",p);
                        wait(); //on attends le fils
                }
```

On peut aussi utiliser waitpid(pid_t pidAEcouter, int *status, int options)
(le pidAECouter peut être un pid fils précis, ou -1 par exemple pour n'importe quel fils)
status est un pointeur sur un entier, que la fonction renseignera (fin normale, anormale, etc)

Ce qui est dommage ici, c'est que finalement on attends pour chaque fils son exécution (on est bloqué, wait est un appel bloquant). Or on peut vouloir que le père continue à faire son travail (lancer d'autres fils par exemple). C'est l'intéret de l'option WNOHANG qui peut être fournie à waitpid(), qui devient alors non bloquant.
Mais alors on a le même problème : comment réagir au bon moment, en temps que père ? Il faut alors se préparer à recevoir et à gérer des signaux

Les signaux

consulter kill -l pour obtenir la liste
On peut programmer notre réaction à un signal / on peut les ignorer, on peut changer leur comportement. Les seuls signaaux qu'on ne peut pas intercepter sont SIGKILL et SIGSTOP.
Pour intercepter et traiter un signal, il faut écrire une fonction attendant un entier (obligatoire, c'est le numéro du signal géré)
Il suffira ensuite d'appeler la primitive signal() en lui indiquant le signal à intercepter, et la fonction à lancer (il s'agit ici d'un pointeur sur la fonction, mais il suffit d'écrire le nom de la fonction, le compilateur retrouvera ses petits).
Par exemple, interceptons le signal SIGHUP (souvent synonyme de relecture des fichiers de configuration)

```
void restart(int s) { //fonction qui traitera le signal
        printf(« Je relis mes fichiers de conf, on me l'a demandé\n »);
}

int main.....{
        ...
        ...
        if (signal(SIGHUP,restart)==SIG_ERR) {
                perror(« problème interception du SIGHUP »);
                exit(-1); //c'est la cata, on donne un retour d'erreur
        }
        ....
}
```

Plus évoluée, et dont l'usage est recommandée, la primitive sigaction() dont le rôle est identique utilise des structures de type sigaction.
Sigaction(int nSignal, struct sigaction *act, struc sigaction *oldact)

Comme d'habitude, le signal est l'int correspondant à un nom de signal.
La structure sigaction à fournir doit avoir été déclarée, (et surement remplie par un memset pour éviter les mauvaises surprises), et son membre sa_sigaction doit pointer sur la bonne fonction.

```
        struct sigaction act;  //notre structure


        memset(&act,0,sizeof(act)); //pour eviter les mauvaises surprises !
        act.sa_sigaction=laFonctionCréePour;  //je remplis les bonnes infos dans la structure
        if(sigaction(SIGINT,&act,NULL)==-1) {  //et j'intercepte
                perror("SIGINT pas pris !");
                exit(5);
        }
```

Exercice 3 :

Ecrire un petit programme qui intercepte deux signaux : le contrôle C de l'utilisateur (qui tente de l'arréter) et le SIGUSR1 (venant d'un autre terminal) qui au contraire lui prolonge la vie. Il faut par exemple disposer d'un entier (déclaré en global, donc hors de toute fonction) valant 5 au début du programme. Chaquel appel à CTR-C décremente ce compteur, et reception d'un signal SIGUSR1 lui ajoute 10. Le main quant à lui boucle tant que le nombre de vie ne vaut pas zéro.

Ecrire ce programme avec signal, puis avec sigaction.