Solutions des exercices du TD/TP 1

Solution 1

1. Un entier, un pointeur, un flottant est codé sur 4 octets, un char est codé sur un octet, une chaîne C est un pointeur. Un tableau a comme taille le produit de sa longueur par la taille d'un élément. Un tableau dynamique n'est qu'un pointeur ! K est une macro traitée par le préprocesseur, pas une variable C.
2. Les variables locales ne sont pas initialisée, tandis que les variables globales et *static* le sont à 0.
3. – les variables locales l, s, td sont dans la pile `0x22...` ;
   – les variables globales KBIS, tab, f sont dans le segment de données statiques `0x40...` ;
   – la variable statique j est dans le segment de données statiques `0x40...` ;
   – `s[0]` pointe sur "un" qui est dans le segment de données statiques ; (littéral chaîne connu à la compilation)
   – `tab[1]` pointe sur 0 qui est dans le segment de données statiques ;
   – `td[0]` pointe sur 0 qui est dans le segment de données dynamiques (tas) en `0x66...` ;
4. La durée de vie des objets locaux est égale à la durée de vie de l'appel de fonction. La durée de vie des objets dynamiques dépend explicitement du programmeur : jusqu'à `free(td)`. La durée de vie des objets statiques est égale à la durée de vie du processus.

Solution 2

1.
```
#define N1 100
int main(){
  int t[N1]; // non initialisées
```
2.
```
  int N2;
  printf("Entrez un entier positif SVP :");
  scanf("%d",&N2);
  int t2[N2]; // non initialisées
```
3.
```
  int N3;
  int *t3;
  printf("Entrez un entier positif SVP :");
  scanf("%d",&N3);
  t3=malloc(N3*sizeof(int)); // non initialisées
```
4.
```
  int** mat;
  mat=malloc(N2*sizeof(int*));
  if(!mat) exit(1); /* plus d'espace */
  for(int i=0;i<N2;i++){
    mat[i]=malloc(N3*sizeof(int));
    if(!mat[i]) exit(1); /* plus d'espace */
  }
  for(int i=0;i<N2;i++){
    for(int j=0;j<N3;j++){
      mat[i][j]=i*N3+j;
      printf("%d ",mat[i][j]);
    }
  }
  for(int i=0;i<N2;i++){ /* désallocation */
    free(mat[i]);
  }
  free(mat);
  mat=NULL;
```
5. Il faut retourner le pointeur sur la zone allouée dans la fonction.
6.
```
  int matpile[N2][N3];
  for(int i=0;i<N2;i++){
    for(int j=0;j<N3;j++){
      matpile[i][j]=i*N3+j;
      printf("%d ",matpile[i][j]);
    }
  }
```
   La durée de vie de l'objet `matpile` est égale à la durée de vie du bloc où il est défini. Il n'est pas visible en dehors.

Solution 3 Dans un système mono-tâche (ou mono-programmation), l'unité centrale est mobilisée à 100% constamment par ce processus qui boucle. Il ne peut gêner que l'utilisateur qui l'a lancé. Il continue l'exécution tant qu'on le laisse faire.

Dans un système multi-tâche (ou multi-programmation), le processus correspondant obtient un quantum de temps et il l'utilise entièrement. C'est gênant pour tous les autres utilisateurs (et processus) car il utilise la ressource `CPU` et

empêche les autres de prendre la main. Mais ça ne les bloque pas. Ils sont ralentis. De façon systématique ce processus reprendra la main et épuisera un quantum de temps à chaque reprise.

Pour l'arrêter, une interruption est nécessaire : par exemple `Ctrl C` ou `Ctrl \` dans un système multi-tâche (cf. Unix) ou `kill -9 1234` ou `1234` est le pid du processus infernal.

Solution 4 1. il est le seul processus utilisateur,

2. tous les autres sont en attente d'entrée-sortie,
3. il est le plus prioritaire.

Pour l'enchaînement des opérations, voir le cours.

Solution 5 *Les réponses ne sont pas détaillées. dans chaque cas, il faut se demander ce qui se passerait si tout utilisateur pouvait exécuter à sa guise les instructions ou codes correspondants. Attention quand même : pour ce qui concerne la date du jour ou l'allocation mémoire, ce ne sont pas des instructions simples, mais des programmes ou des fonctions.*

masquer une interruption	*oui*	lire la date du jour	*non*
modifier la date du jour	*oui*	modifier l'allocation mémoire	*oui*
appeler l'ordonnanceur	*oui*	appel d'une fonction en C	*non*

Solution 6 1.
```
afficher "Nb de paramètres (arguments) : " argc "\n"
Pour i=0 à argc-1
  afficher i ":" argv[i] "\n"
afficher "\nVariables d'environnement :\n"
i=0
Tq arge[i] != NULL
  afficher i ":" arge[i] "\n"
  i++
afficher "Nb de variables d'environnement :" i "\n"
```

2.
```
#include <stdio.h>

int main(int argc, char** argv, char** arge){
  printf("Nb de paramètres (arguments) : %d\n", argc);
  for(int i=0; i<argc; i++){
    printf("%d : %s\n",i, argv[i]);
  }
  printf("\nVariables d'environnement :\n");
  int i=0;
  while(arge[i] != NULL){
    printf("%d : %s\n", i, arge[i]);
    i++;
  }
  printf("Nb de variables d'environnement : %d\n",i);
}
```

Solution 7 Toutes les substitutions sont faites par le `shell` avant de lancer l'exécution. Donc on aura successivement :

- 4 paramètres ;
- nombre des fichiers du répertoire, sauf ceux qui commencent par `.` (point) + 1 (`$PATH`).
- nombre des fichiers du répertoire d'accueil qui commencent par `.` et qui possèdent au moins 2 lettres supplémentaires (ni `.` lui-même ni `..`)

Solution 8 1.
```
i=0
TROUVE=faux
Tq arge[i] != NULL et non TROUVE
  TROUVE=chercherDebut("PATH=",arge[i]);
  i++
si  non TROUVE
  afficher "Pas de PATH dans les variables d'env !"
  exit
sinon
  TROUVE=faux
  i--
  TROUVE=rechercher(getcwd() ou "." , arge[i]+taille("PATH="))
  if TROUVE
    afficher "Ce rép. est exécutable!"
  sinon
    afficher "Ce rép. n'est pas exécutable!"
```

2.
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char** argv, char** arge){
  char * motif="PATH=", *delim=":", *trouve=NULL; // UNIX delim=";"
  int i = 0 ;
  while(arge[i] != NULL && arge[i]!=(trouve=strstr(arge[i],motif)))
    i++;
  if (arge[i]==NULL){
    printf("La chaîne %s n'existe pas !\n",motif);
    return 1;
  } else {
    char * cwd=(char *)malloc(1000);
    getcwd(cwd,1000); // répertoire courant

    char * source=arge[i]+strlen(motif);
    char * tok=strtok(source,delim); // appel initial avec la source
    while(tok && strcmp(tok,cwd) && strcmp(tok,".")){
      tok=strtok(NULL,delim);
    }
    if (tok)
      printf("Ce rép. %s est exécutable grâce à : %s\n",cwd, tok);
    else
      printf("Ce rép. %s n'est pas exécutable !\n",cwd);
  }
}
```

Solution 9
1.
```
#include <stdio.h>
#include <stdlib.h>
int fact(int n){
  if (n<=1)
    return 1;
  else
    return n*fact(n-1);
}
int main(int argc, char** argv, char** arge){
  if(argc!=2){
    printf("Syntaxe incorrecte : %s 8 \n", argv[0]);
    return 1;
  } else {
    int n=atoi(argv[1]);
    printf("%d ! = %d\n",n,fact(n));
    return 0;
  }
}
```
2.
```
fact(3)
  fact(2)
    fact(1)
    ret 1;
  ret 2*1
ret 3*2=6
```

Solution 10
1. On choisit un tableau à 2 dimensions dynamique où la largeur de chaque ligne est égale à son indice +1.
2. **triangle(n)**

```
tab=malloc(n+1 entiers)
Pour i=0 à n
  tab[i]=malloc(i+1 entiers)
  tab[i][0]=1
  tab[i][i]=1
  pour j=1 à i-1
    tab[i][j]=tab[i-1][j-1]+tab[i-1][j]
```

3.
```
#include <stdio.h>
#include <stdlib.h>
int** triangle(int n){ /* ret un triangle de pascal */
  int** tab=malloc((n+1)*sizeof(int));
  for(int i=0; i<=n; i++){
    tab[i]=malloc((i+1)*sizeof(int));
    tab[i][0]=1;
    tab[i][i]=1;
    for(int j=1; j<i; j++){
      tab[i][j]=tab[i-1][j-1]+tab[i-1][j];
    }
  }
  return tab;
}
void afficherTriangle(int** tab, int n){
  for(int i=0;i<=n;i++){
    for(int j=0; j<=i; j++){
      printf("%6d ", tab[i][j]);
    }
    printf("\n");
  }
}
int main(int argc, char** argv, char** arge){
  if(argc!=3){
    printf("Syntaxe incorrecte : %s 8 4\n", argv[0]);
    return 1;
  } else {
    int n=atoi(argv[1]);
    int p=atoi(argv[2]);
    int **tab=triangle(n);
    afficherTriangle(tab,n);
    printf("Nombre de combinaisons C(%d,%d) = %d\n",n,p,tab[n][p]);
    return 0;
  }
}
```

Solution 11 En liaison dynamique, la taille minimale d'un exécutable est petite : il faut juste pouvoir retourner au système après exécution afin de nettoyer les segments utilisés. Les données statiques non initialisées prennent moins de place car l'espace sera alloué au chargement du programme. Idem pour la pile et le tas alloués au chargement.

Solution 12
1. La deuxième fonction affiche les mêmes valeurs que la première. En effet, le second tableau est alloué au même endroit que le premier, par conséquent, on voit encore les anciennes valeurs !
2. Il faut initialiser soit même les objets automatiques (pile) et dynamiques (tas) sous peine de grosses déconvenues.

Solution 13 idem exercice précédent