

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 épuiera 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 - 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

	masquer une interruption	<i>oui</i>		lire la date du jour	<i>non</i>
fonctions.	modifier la date du jour	<i>oui</i>		modifier l'allocation mémoire	<i>oui</i>
	appeler l'ordonnanceur	<i>oui</i>		appel d'une fonction en C	<i>non</i>

Solution 6 1. afficher "Nb e paramètres (arguments) : " argc "\n"

```
Pour i=0 à argc-1
    afficher i ":" argv[i] "\n"
afficher "\nVariables 'environnement : \n"
i=0
Tq arge[i] != NULL
    afficher i ":" arge[i] "\n"
    i++
afficher "Nb e variables 'environnement : " i "\n"
```

2. #inclu e <st io.h>

```
int main(int argc, char** argv, char** arge){
    printf("Nb e paramètres (arguments) : % \n", argc);
    for(int i=0; i<argc; i++){
        printf("% : %s\n",i, argv[i]);
    }
    printf("\nVariables 'environnement : \n");
    int i=0;
    while(arge[i] != NULL){
        printf("% : %s\n", i, arge[i]);
        i++;
    }
    printf("Nb e variables 'environnement : % \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 e PATH ans les variables 'env !"
    exit
sinon
    TROUVE=faux
    i--
    TROUVE=rechercher(getcw () 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=", * elim=":", *trouve=NULL; // UNIX  elim=";"
    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 * cw =(char *)malloc(1000);
        getcw (cw ,1000); // répertoire courant

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

```

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("%6 ", tab[i][j]);
        }
        printf("\n");
    }
}

int main(int argc, char** argv, char** arge){
    if(argc!=3){
        printf("Syntaxe incorrecte : %s\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(% ,% ) = % \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