Examen de SELC/INF104

1h30 – Sans document

Note : les signatures des principales fonctions vues en cours, et utiles pour l'examen, sont listées en annexe.

Questions de cours (5 points)

Répondez aux questions suivantes en sélectionnant une (et une seule) réponse parmi celles proposées (a, b, ou c). Chaque réponse peut rapporter 0.5 point. Les mauvaises réponses seront pénalisées de -0.5 point. Vous pouvez répondre sur l'énoncer si vous voulez (dans ce cas, n'oubliez pas de le rendre en indiquant votre nom dessus).

1. L'état d'un processus peut passer de
   a) « bloqué » à « en cours d'exécution »
   b) « prêt » à « bloqué »
   c) les deux réponses précédentes sont vraies
2. Un défaut de page survient lorsqu'un processus essaie d'accéder à une zone mémoire
   a) qui n'est pas chargée en mémoire
   b) qui a été corrompue par un autre processus
   c) qui ne lui appartient pas
3. Dans un système d'exploitation, l'ordonnancement consiste:
   a) à trouver le meilleur agencement des objets en mémoire pour optimiser le temps de calcul
   b) à sélectionner un processus pour le faire passer de l'état « prêt » à « en cours d'exécution »
   c) à ordonner les fragments de fichiers sur le disque pour récupérer de l'espace de stockage
4. Un interblocage survient lorsque:
   a) un processus A ne se termine jamais et monopolise l'accès au processeur, empêchant un processus B de s'exécuter
   b) un processus A attend que B ait libéré un sémaphore alors que B attend que A ait libéré un autre sémaphore
   c) un processus père attend qu'un de ses fils se termine, ce qui n'arrive jamais
5. Lorsqu'un processus exécute l'opération V sur un sémaphore S, que se passe-t-il si le compteur associé à S est strictement positif après incrément?
   a) le processus considéré passe dans l'état bloqué
   b) rien, le compteur du sémaphore a été incrémenté et c'est tout
   c) un processus en attente sur S passe dans l'état prêt
6. La pagination permet
   a) de stocker des fichiers, page par page, sur le disque
   b) de restaurer partiellement l'état d'un fichier corrompu
   c) d'exécuter des programmes dont l'espace d'adressage est plus grand que la capacité de la mémoire « vive » (i.e. non-permanente) de la machine
7. Lors d'une division par zéro dans un programme, le système d'exploitation
   a) notifie l'utilisateur du système avec un message d'erreur
   b) termine le processus fautif
   c) envoie un signal au processus fautif
8. L'appel système alarm(k) permet
   a) de faire passer le processus qui l'exécute dans l'état bloqué pendant k secondes
   b) de lancer l'alarme « k », i.e. notifier une erreur à l'utilisateur identifiée par la valeur k
   c) d'émettre un signal à destination du processus qui l'exécute dans k secondes

9. La table des pages permet de
    a) savoir sur quel bloc disque est stockée une page d'un fichier
    b) savoir sur quelle page physique est stockée une page logique
    c) de réaliser des branchements conditionnels dans le code d'un programme exécutable
10. La redirection d'entrée/sortie permet
    a) de changer le traitement associé à la réception/émission d'un signal
    b) de réaliser une interaction de type producteur/consommateur
    c) de changer les fichiers associées aux descripteurs 0, 1, et/ou 2

Exercices d'application du cours (8 points)

Exécution des processus (4 points)

On cherche à développer une application pour savoir si une liste de fichiers contient un certain motif (séquence de caractères sans espace) sur une ligne. Chaque fichier contient un texte différent composé d'un mot par ligne (on supposera que les mots contenus dans les fichiers sont composés au maximum de 30 caractères). L'application doit afficher « trouvé» si le motif a été trouvé au moins une fois dans un fichier, ou « absent » sinon. La fonction `find` ouvre un fichier et le parcourt jusqu'au bout : si il existe une ligne du fichier correspondant exactement au motif, la fonction retourne 0, sinon elle retourne -1.
Voici le code correspondant à une première version de l'application complète :

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <unistd.h>
5. int find (char * filename, char*
   motif){
6.  FILE * f=  fopen (filename, "r");
7.  int ret=0;
8.  char wordread[31];
9.    while (ret!= EOF){
10.     ret=fscanf(f,"%s",wordread);
11.     if (strcmp(motif,wordread)==0)
12.       {return 0;}
13.     }
14.     return -1;
15. }
```

```
16. int main( int argc ,
              char* argv[])
17. {
18.   int i;
19.   for (i=2; i<argc;i++){
20.   if (find(argv[i],
              argv[1])==0)
21.     {printf("trouvé\n");
22.      return 0;}
23.   }
24.   printf("absent\n");
25.   return -1;
26. }
27.
```

En supposant que l'exécutable **appli** résulte de la compilation de ce programme, on peut lancer la recherche en exécutant par exemple : **./appli bonjour file1 file2 file3** . Dans cet exemple, file1 file 2 et file3 sont des fichiers accessibles en lecture.

Question 11 (1 point) : Modifier ce code pour que la recherche sur chaque fichier soit exécutée en parallèle dans N processus différents, où N est le nombre de fichiers passés en paramètres sur la ligne de commande. On s'autorise dans ce cas à avoir plusieurs affichages de trouvé/absent, i.e. un pour chaque fichier. Vous n'avez a priori que la fonction main à modifier, et on vous demande de n'utiliser que l'appel système « fork » dans cette question.

Question 12 (3 points) : On souhaite maintenant produire une unique réponse (trouvé ou absent) tout en conservant l'exécution en parallèle de la recherche du motif dans chaque fichier. De plus, nous ne souhaitons pas utiliser de sémaphores pour réaliser cette synchronisation.

Question 12.a : proposez et expliquez le principe d'une solution reposant sur la synchronisation entre processus père et processus fils. (Dans quel(s) processus se fait à priori l'affichage, comment décide-t-on que le motif est présent ou absent des fichiers passés en ligne de commande) (1,5pts)

Question 12.b : fournissez le code de la fonction main correspondant à votre solution. *Indication : Des macro et signatures de fonctions sont rappelées en annexe du sujet (attention toutes ne sont pas nécessairement pertinentes).* (1,5 pts)

Système de gestion des fichiers (4 points)

Les figures 1 et 2 représentent l'état initial d'un système de gestion de fichiers (SGF) de type *Unix File System* (UFS). Nous considérons cet état initial dans les questions qui suivent.

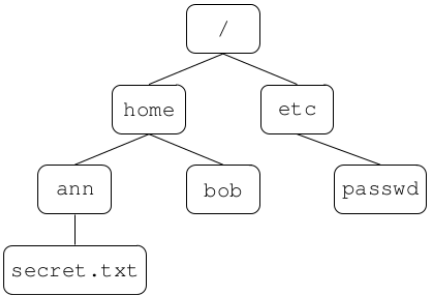


Figure 1 : Organisation logique de l'arborescence.

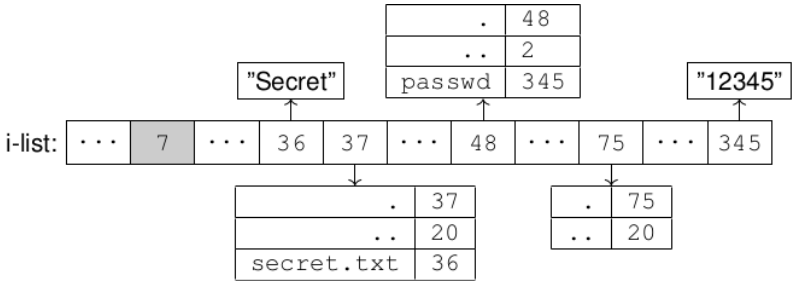


Figure 2 : Organisation physique représentée par une i-list et le contenu sur disque associé aux i-nodes 36, 37, 48, 75, et 345.

Question 13 (1.5 point) : Quel sont les numéros de *i-node* associés à chaque fichiers (secret.txt, passwd) et à chaque dossiers (home, etc, ann, bob) présents dans le SGF ?

Question 14 (1.5 point) : Depuis le dossier /home/bob l'administrateur du système exécute les commandes indiquées ci-dessous (l'une après l'autre) dans le shell. Illustrez l'évolution de l'organisation *physique* (i-list/contenu sur disque associé aux i-nodes telle que illustrée sur la figure 2). Note : on suppose que le premier i-node libre est le i-node 7.

1. cp ../ann/secret.txt ./bobcret.txt
2. rm ../ann/secret.txt
3. mv bobcret.txt ../ann

Attention, les figures à compléter sont fournies en annexe ; vous pouvez donc répondre directement sur ces figures et rendre le sujet d'examen une fois complété. N'oubliez pas dans ce cas d'indiquer votre nom sur le sujet d'examen.

Question 15 (1 point) : Plusieurs attributs sont associés avec chaque i-node dans un SGF de type UFS – comme indiqué sur la figure 3. À partir de l'état initial du SGF, indiquez l'évolution de l'attribut Nbre_de_liens de l'i-node associé au fichier secret.txt après l'exécution de chacune des commandes ci-dessous. Ces commandes sont exécutées successivement depuis le dossier /home/ann. **Votre réponse doit inclure une justification. Il ne suffit pas d'indiquer seulement les valeurs de l'attribut.**

1. ln secret.txt ./ln1.txt
2. cp secret.txt ./secret2
3. ln -s secret.txt secret3
4. rm *.txt

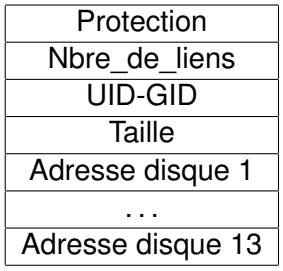


Figure 3 : Représentation d'un i-node.

Problème (7 points)

Nous cherchons à manipuler un très grand nombre d'objets (max_vir_obj) qui pour des raisons d'encombrement ne seront pas tous présents en mémoire à un instant donné de l'exécution du programme. On décide de reprendre les principes vus en cours: virtualiser l'espace mémoire associé aux objets en les stockant temporairement dans un fichier « vir_obj_file » (i.e. sur le disque).

Les adresses des objets virtuels, au nombre de max_vir_obj, sont stockées dans le tableau vir_obj_tab. Les objets physiquement présents en mémoire, au nombre de max_phy_obj, sont stockés dans le tableau phy_obj_tab. Certaines adresses du tableau vir_obj_tab correspondent donc à des adresses physiques d'objets du tableau phy_obj_tab, les autres sont nulles (NULL) car les objets ne sont pas physiquement présents en mémoire. Tous les objets virtuels (max_vir_obj) sont dans un fichier de descripteur vir_obj_file. On rappelle en annexe les signatures des principales fonctions d'Unix vues en cours. Le code à compléter est également fourni en annexe.

Tout objet virtuel a un identifiant unique, un entier, qui correspond à l'indice de l'objet dans le tableau des objets virtuels (vir_obj_tab). L'association entre un objet virtuel et un objet physique est stockée dans le tableau vir_phy_map. Ce tableau contient des structures vir_obj_t composées de:

- un champ phy_obj_id qui contient l'indice de l'objet dans le tableau des objets physiques lorsque celui-ci est effectivement présent en mémoire

- un champ last_use qui contient la dernière date à laquelle l'objet virtuel a été utilisé: si obj1 et obj2 sont deux objets virtuels, obj1.last_use > obj2.last_use signifie que obj1 a été utilisé plus récemment que obj2.

La figure 4 illustre l'organisation du stockage des objets, à un instant donné, afin de clarifier la description qui précède :

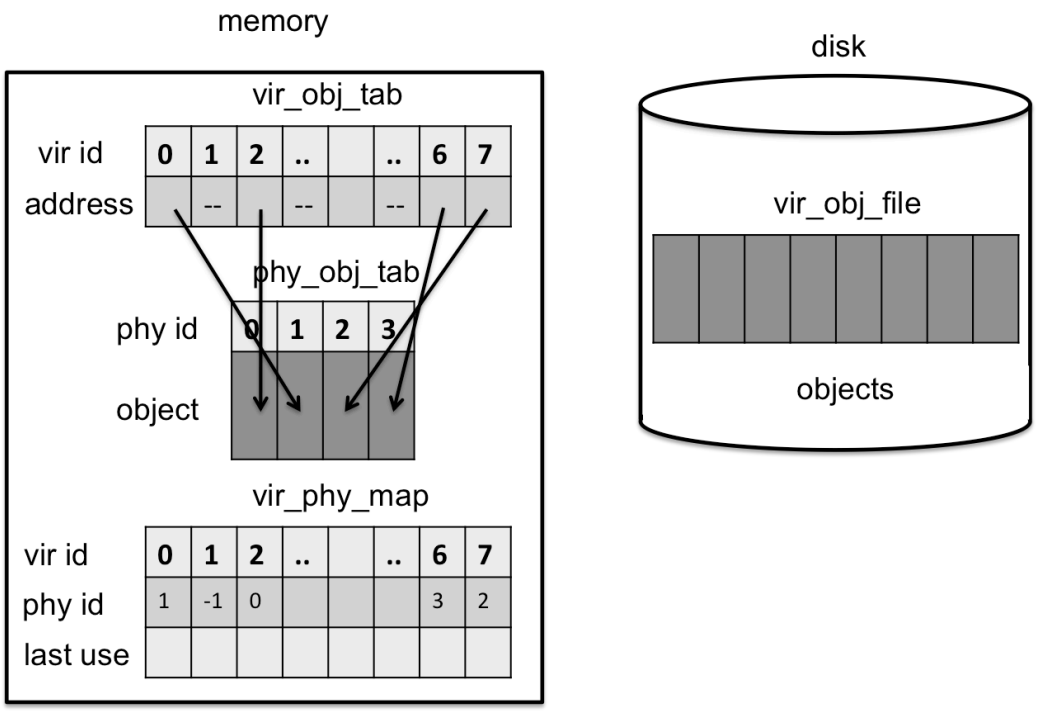


Figure 4 : illustration des structures de données pour la virtualisation d'objets

Question 16 (1.5 point) : comme indiqué précédemment, la première étape consiste à stocker les objets virtuels initialisés avec une valeur par défaut. En complétant le code autour du commentaire "Question 16", créer un fichier du nom de "vir_obj_file" et remplissez le avec des objets virtuels initialisés avec la valeur l'objet initial_obj (voir ligne 62). On prendra éventuellement soin d'effacer un fichier existant.

Question 17 (1.5 point) : Comme indiqué précédemment lorsque l'on accède à un objet virtuel, celui-ci peut ne pas être associé à un objet physique. Pour tester notre outil, nous avons écrit un programme de test (main) qui permet de saisir l'identifiant de l'objet virtuel à modifier, puis de saisir les attributs de cet objet. Les modifications risquent de produire une erreur lors de la modification d'un objet virtuel qui ne serait pas présent physiquement en mémoire. Précisez comment se manifeste concrètement cette erreur et comment l'intercepter avec la fonction catch_segflt. Complétez le code autour du commentaire "Question 17".

Question 18 (1.5 point) : Nous allons maintenant compléter le code pour traiter du cas d'un "défaut d'objet" en analogie avec un "défaut de page" comme nous l'avons vu en cours. Nous allons tout d'abord trouver l'objet de remplacement en complétant le code autour du commentaire "Question 18". Vous appliquerez la politique LRU pour stocker dans rpl_obj_id l'identifiant de l'objet à remplacer.

Question 19 (1.5 point) : Maintenant que l'objet de remplacement a été trouvé, nous allons achever le traitement du "défaut d'objet". En complétant le code autour des deux commentaires "Question 19", vous sauvegarderez l'objet de remplacement à l'endroit adéquat dans le fichier et inversement restaurerez dans l'objet physique libéré l'objet virtuel en retrouvant celui-ci dans le fichier à l'endroit adéquat.

Question 20 (1 point) : Dans la question 17, nous avons intercepté une erreur qui se produirait lors de la modification d'un objet virtuel qui ne serait pas présent physiquement en mémoire. Dans les questions 18 et 19, nous avons enrichi le code de sorte que l'objet virtuel soit désormais présent en mémoire. Complétez le code autour du commentaire "Question 20" pour faire sorte qu'une fois l'objet virtuel rendu présent en mémoire physique on reprenne l'exécution avant l'erreur. Ayant corrigé l'erreur, la modification pourra en effet avoir lieu.

ANNEXES

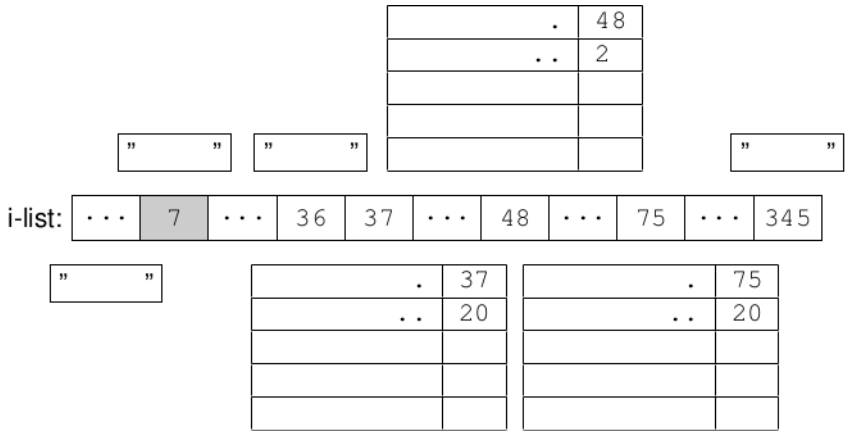
Rappel des signatures des principales fonctions vues en cours

```
pid_t fork(void);
int execl(const char *path, const char *arg0, ...);
void exit(int status);
int creat(const char *path, mode_t mode);
int open(const char *path, int oflag, ...);
ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);
int close(int fildes);
off_t lseek(int fildes, off_t offset, int whence);
pid_t wait(int *wstatus);
```

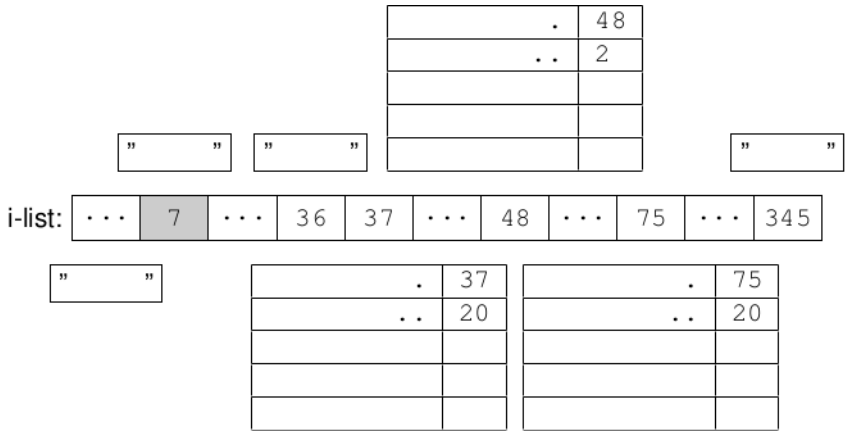
```
/* WEXITSTATUS return the low-order 8 bits of the status integer value */
#define WEXITSTATUS(status) (((status) & 0xff00) >> 8)
typedef void (*sig_t) (int);
sig_t signal(int sig, sig_t func);
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Figures à compléter en réponse à la question 14 (partie « Système de gestion des fichiers »)

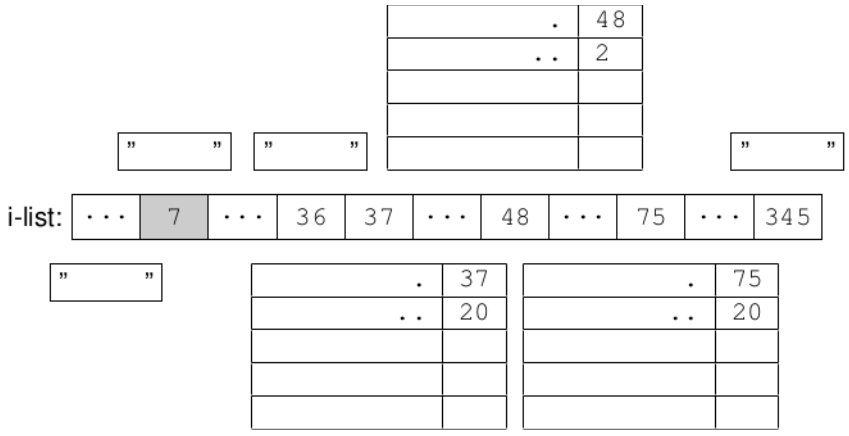
1. cp ../ann/secret.txt ./bobcret.txt



2. rm ../ann/secret.txt



3. mv bobcret.txt ../ann



Code à compléter en réponse à la partie « Problème »

```
1   #include <fcntl.h>
2   #include <setjmp.h>
3   #include <signal.h>
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <unistd.h>
7
8   #define max_vir_obj 8
9   #define max_phy_obj 4
10
11  typedef struct {
12    int x;
13    int y;
14  } object_t;
15
16  object_t * vir_obj_tab[max_vir_obj];
17  int        vir_obj_id; // Current object id
18
19  #define NONE -1
20  unsigned int global_last_use = 0;
21
22  typedef struct {
23    unsigned int phy_obj_id;
24    unsigned int last_use;
25  } vir2phy_t;
26
27  object_t  phy_obj_tab[max_phy_obj]; // Table of physical objects
28  vir2phy_t vir_phy_map[max_vir_obj]; // Map virtual to physical ids
29
30  int vir_obj_file; // Descriptor of the file used to store virtual
31  objects
32
33  void initialize ();
34  void catch_segflt (int sig);
35  void modify_object (unsigned int id);
36
37  jmp_buf ctxt;
38
```

```
39
40  int main (int argc, char * argv[]) {
41
42    initialize ();
43
44    // Catch access to virtual objects that are not physically in memory.
45
46    /* Question 17 */
47
48    while (1){
49      printf ("vir obj id> ");
50      if (scanf ("%d", &vir_obj_id) <= 0) exit (0);
51
52      /* Question 20 */
53
54      modify_object (vir_obj_id);
55
56      global_last_use++;
57      vir_phy_map[vir_obj_id].last_use = global_last_use;
58    }
59  }
60
61  void initialize(){
62    object_t initial_obj = {0, 0};
63
64    // Initialize file used to store virtual objects.
65    // Initialize them with the initial object value.
66
67    unlink ("vir_obj_file");
68    /* Question 16 */
69
70    for (vir_obj_id = 0; vir_obj_id < max_vir_obj; vir_obj_id++) {
71      if (vir_obj_id < max_phy_obj) {
72        // Map the first virtual objects as physical objects
73        // Initialize the physical object with their value
74        vir_phy_map[vir_obj_id].phy_obj_id = vir_obj_id;
75        vir_obj_tab[vir_obj_id] = &phy_obj_tab[vir_obj_id];
76        phy_obj_tab[vir_obj_id] = initial_obj;
77      } else {
78        // Unmap remaining virtual objects
79        vir_phy_map[vir_obj_id].phy_obj_id = NONE;
80        vir_obj_tab[vir_obj_id] = NULL;
81      }
82      vir_phy_map[vir_obj_id].last_use   = 0;
83    }
84  }
85
```

```
 86
 87  void catch_segflt (int sig){
 88    unsigned int phy_obj_id;
 89    unsigned int rpl_obj_id;
 90    unsigned int obj_id;
 91
 92    // Find the least recently used object to replace
 93    /* Question 18 */
 94
 95    phy_obj_id = vir_phy_map[rpl_obj_id].phy_obj_id;
 96    vir_phy_map[rpl_obj_id].phy_obj_id = NONE;
 97    vir_phy_map[rpl_obj_id].last_use = 0;
 98
 99
100    // Save replaced physical object (find its location in memory) on
101    // disk (find its location in file)
102    /* Question 19 */
103
104    vir_phy_map[vir_obj_id].phy_obj_id = phy_obj_id;
105    vir_phy_map[vir_obj_id].last_use   = global_last_use;
106
107    // Unmap old virtual object to its physical object
108    vir_obj_tab[rpl_obj_id] = NULL;
109
110    // Map this physical object to new virtual object
111    vir_obj_tab[vir_obj_id] = &phy_obj_tab[phy_obj_id];
112
113    // Load virtual object (find its location in file) from disk in
114    // physical object (find its location in memory)
115    /* Question 19 */
116    // Object fault has been caught. Try again
117    /* Question 20 */
118  }
119
120  void modify_object (unsigned int id) {
121      printf("x = ");
122      scanf ("%d", &(vir_obj_tab[id]->x));
123      printf("y = ");
124      scanf ("%d", &(vir_obj_tab[id]->y));
125  }
```