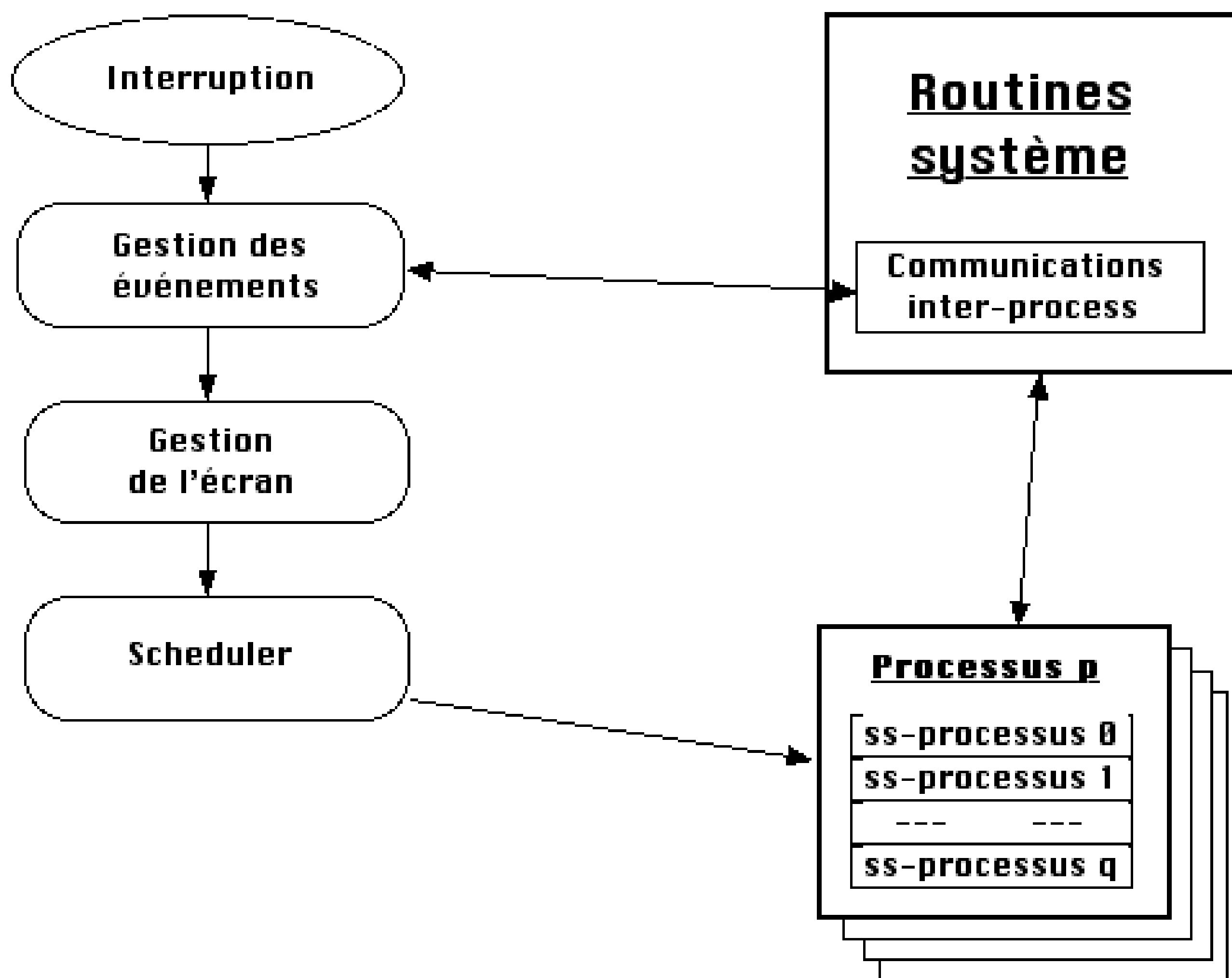
Projet Info

Environnement Multitâche et Multifenêtrage

Sylvain HUET (91)


Interruption
Gestion des événements
Gestion de l'écran
Scheduler
Routines système
Communications inter-process
Processus p
ss-processus 0
ss-processus 1
--- ---
ss-processus q

PRESENTATION

1. Objet .

Concrêtement, il s'agit d'offrir au programmeur un jeu de routines permettant de développer des applications multitâches et multifenêtrages. Il convient donc dans cette optique d'insérer une couche entre le système du Macintosh et l'utilisateur. Pour des raisons qui seront exposées plus loin, cette couche n'est pas transparente : les applications devront être spécifiquement développées pour ce système.

2. Catactéristiques .

Les routines proposées fonctionnent quel que soit le microprocesseur 680x0, (même si le 68000 se révèle un peu lent).
Les routines graphiques ont été développées pour un écran monochrome (de taille quelconque). On relève cependant une incompatibilité des Mac munis d'une carte couleur (ou avec nuance de gris), même lorsqu'on les utilise en mode monochrome. Le système idéal est donc le PB 145 ou 170.

Le système permet alors d'exécuter simultanément un nombre indéterminé de tâches et d'afficher un nombre indéterminé d'objets à l'écran.
Les tâches sont exécutées proportionnellement à leur niveau de priorité ; elles peuvent se placer en attente d'événement, ou d'un signal provenant d'une autre tâche.
Les objets sont de deux types. Il y a d'abord les fenêtres, caractérisées par une forme rectangulaire ; toutes les routines graphiques leur sont dédiées : tracé de points, de lignes, affichage de caractères, scrolling verticaux, gestion de curseurs, etc... Il y a ensuite les 'sprites', caractérisés par un masque d'affichage ; l'exemple le plus simple en est le curseur de la souris. Ces objets sont traités indifféremment, et peuvent être définis les uns par rapport aux autres.

3. Avertissement .

Le programme proposé n'est pas un système d'exploitation au vrai sens du terme, même s'il en a l'architecture. Il lui manque certaines commodités, telles qu'une gestion des fichiers de type Finder, une interface même sommaire telle qu'un shell, et bien sûr un compilateur. La raison de ces oublis est évidemment le temps de conception de tels outils.
Quant à la compatiblité avec les applications existantes, elle pourrait être rétablie en reprogrammant tous les appels Trap, ce qui est plutôt long et fastideux.
Il convient cependant de remarquer que tous ces ajouts pourraient être réalisés dans le cadre de la structure adoptée. Seul le temps les rend inenvisageables dans le cadre du projet.

I. ASPECT THEORIQUE

1. Le Multitâche

a. processus et sous-processus

Le système gère un nombre indéterminé de processus et de sous-processus.
- un processus n'est pas un programme : c'est simplement une structure logique regroupant un nombre indéterminé de sous-processus. En fonctionnement usuel, on peut identifier les termes processus et application. Outre la liste des sous-processus, le processus est déterminé par un état (actif / suspendu / en attente), un niveau de priorité (pour le scheduler), un nom et une 'table de communication par défaut' (voir plus loin).
- un sous-processus correspond par contre à un bout de programme. Il est caratérisé par une pile système (dont la taille est réglable), un état, un niveau de priorité (au niveau du processus), une minuterie (pour les sous-processus que l'on souhaite exécuter en fonction de l'horloge) et un système de réception de communications.

Processus et sous-processus sont repérés par un numéro d'identification ou ID.

Le principe du scheduler est très simple :
- dans un premier temps il détermine quel processus va être exécuté : ceci se réalise proportionnellement au niveau de priorité du processus.
- dans un second temps il détermine quel sous-processus activer : ceci se réalise encore proportionnellement au niveau de priorité du sous-processus.

b. communications inter-processus

Les communications inter-processus (ou 'comm') sont une partie importante du système : c'est notamment par elles que s'effectue la gestion des événements.
- il est possible de définir 32 types de 'comm' (le programme proposé en utilise quatorze) : entier, pointeur, coordonnées d'un point, déplacement, click, touche clavier, code erreur, horloge, chaîne de caractères, etc...
- il existe une 'table de comm par défaut' au niveau du système et au niveau de chaque processus.
- chaque sous-processus possède un masque de réception (un bit par type de 'comm'), et une liste des 'comm' reçues, en attente de traitement.

- pour envoyer une 'comm' à une tâche, il y a trois possibilités :
-donner l'ID du sous-processus destinataire.
-donner l'ID du processus destinataire (le sous-processus destinataire étant déterminé par la 'table de comm par défaut' du processus).
-ne pas donner d'ID : le processus destinataire est déterminé par la 'table de comm par défaut' du système. Le sous-processus est ensuite déterminé comme précédemment.
- une fois le sous-processus destinataire déterminé, le système regarde si son masque de réception laisse passer la 'comm'.
- enfin, la 'comm' est ajoutée à la liste de réception du sous-processus. Si celui-ci était en attente, il est réactivé, de même que le processus père.

La gestion de la minuterie consiste à envoyer périodiquement une 'comm' de type horloge au sous-processus (mode always ; ceci permet d’exécuter une tâche à une fréquence donnée). Il y a aussi moyen de n’envoyer qu’une seule 'comm' au bout d’un temps déterminé (mode once).
Les applications doivent respecter certaines règles évidentes, à savoir regarder fréquemment la liste des 'comm' reçues : ceci est particulièrement important dans le cas de l’utilisation de la minuterie en mode always.

c. processus système

Lors de l’initialisation du système, un processus est installé : le processus système. Son rôle principal est de gérer certaines fonctions standard.
Dans le programme proposé, on y trouve un sous-processus qui gère les opérations de disque, un autre qui gère les ascenceurs, un autre qui gère l’affichage des erreurs survenues au niveau du système (mémoire insuffisante, pas assez de canaux pour les opérations de disque...)
Dans un système plus complet, on pourrait y trouver toute la gestion des entrées-sorties, des ressources communes, etc...

2. Le Multifenêtrage.

a. structure adoptée

Le principe adopté repose sur la notion d’objets. Un objet est une structure correspondant soit à une fenêtre, soit à un sprite. Outre divers paramètres tels que le processus père, la position dans la pile, ..., on distingue trois parties :
- la partie ‘bitmap’, qui contient le dessin de l’objet
- la partie ‘fenêtres de travail’, qui permet de définir des zones de travail dans l’objet
- la partie ‘cases logiques’, qui permet de définir des boutons ou des zones manipu-lables à la souris.

L’idée fondamentale est que le bitmap des fenêtres est stocké intégralement dans la mémoire et qu’une application ne travaille jamais directement sur l’écran, mais toujours dans le bitmap de ses objets. La seule routine qui touche à l’écran est celle qui recalcule une zone (x,y)-(x’,y’) : dans un buffer, cette routine réaffiche les objets qui apparaissent dans cette zone, dans l’ordre de la pile, en commençant par l’objet le plus au fond ; puis le buffer est transféré sur l’écran. Cette routine n’est appelée que lors de l’interruption, juste avant le scheduler.

b. aspect graphique.

Les fenêtres sont gérées comme les sprites. La partie ‘bitmap’ comprend donc les coordonnées de l’objet par rapport à l’objet père, les dimensions de l’objet, l’adresse du buffer contenant le dessin, et bien sûr, le type de l’objet (fenêtre ou sprite) qui permet

d’interprêter correctement le buffer.
Le programme propose plusieurs routines graphiques : tracé de ligne, affichage de caractères ... Elles ne fonctionnent qu’avec les objets de types fenêtre, mais rien n’empêche d’ajouter leur équivalent pour les sprites.
On peut souhaiter définir certaines zones de travail dans un objet : à cet effet, toutes les opérations s’effectuent dans une ‘fenêtre de travail’ (ou ‘work’) préalablement définie, entre autre, par :
- les coordonnées de cette sous-fenêtre
- les coordonnées du dernier point affiché (ce qui permet les instructions du type line -(x,y)).
- les coordonnées du curseur
- l’adresse de la fonte
(On peut définir autant de ‘work’ qu’on le souhaite).

Pour effectuer une opération graphique, il faut donc préciser l’objet concerné et le numéro du ‘work’.
Deux mots sur les fontes : le programme proposé gère des fontes non-vectorisées de largeur inférieure à huit points et de hauteur indéterminée. La largeur est la même pour tous les caractères d’une même fonte.

c. aspect logique.

Dans chaque objet, on peut définir un nombre indéterminé de ‘cases logiques’ ; chacune est définie, entre autre, par :
- son type,
- ses coordonnées dans l’objet,
- l’ID du sous-processus qui doit être averti lorsque l’on clique dans cette case,
- l’ID du sous-processus qui doit être averti lorsque la souris rentre dans cette case,
- un nombre indéterminé de registres dans lesquels le programmeur peut stocker ce qu’il souhaite (utilisé par exemple pour les ascenseurs).

Le programme proposé utilise le ‘type’ pour déterminer si, en cliquant sur cette case, la souris peut ‘attraper’ l’objet et le déplacer, soit devant, soit derrière. Il sert aussi à déterminer si la case doit être inversée (avec un Not du dessin) lorsque l’on clique dessus. Ces fonctions sont gérées par le gestionnaire d’événement, ce qui peut paraître inutile. Ce choix permet en fait de pallier le manque de puissance du Mac et de répondre plus rapidement aux sollicitations de l’utilisateur.

3. Le gestionnaire d’événements.

Le gestionnaire d’événements est une partie centrale du système : il est le lien entre l’utilisateur, les objets et les processus. Dans le programme proposé, il gère la souris et le clavier. On pourraît aussi ajouter les entrées de toutes sorte, telles que les liaisons série. On peut diviser sa tâche en plusieurs parties :

a. gestion du déplacement de la souris

A chaque déplacement de la souris une 'comm' ‘déplacement’ est envoyée sans précision

d’ID : le processus intéressé doit régler la ‘table des comm par défaut’.
Lorsque le curseur de la souris pénètre dans une case d’un objet, le gestionnaire d’événements envoye une 'comm' ‘entrée de zone’ au sous-processus concerné (voir §2.c). De même, une 'comm' ‘sortie de zone’ est envoyée lorsque le curseur de la souris quitte la zone.
Ceci permet de gérer facilement des menus déroulants, ou de modifier le dessin du curseur de la souris lorsque celle-ci pénètre dans une certaine zone.

b. gestion des click.

Lorsque l’utilisateur clique sur une case d’un objet, une 'comm' ‘click’ est envoyée au sous-processus concerné. Celui-ci reçoit également la 'comm' ‘fin de click’ lorsque l’utilisateur relâche le bouton. Le programme proposé possède également la notion de double-click (et même de n-click !).
Comme il a été mentionné plus haut, dans un souci de vitesse, le type de la case indique s’il faut déplacer l’objet lorsqu’on clique dessus.

c. gestion du clavier.

Le gestionnaire d’événements scrute le clavier et envoye une 'comm' de type ‘caractère’ sans précision d’ID : là encore, le processus intéressé doit régler la ‘table des comm par défaut’.

II. ASPECT TECHNIQUE

1. Routine d’interruption.

La programmation d’une interface multitâche nécessitel’utilisation d’une interruption qui se produit un nombre suffisamment important de fois par seconde. Comme on veut modifier le point de retour de l’interruption, le plus simple est de prendre le contrôle de celle-ci dès son commencement, et donc de modifier le vecteur d’interruption utilisé par le microprocesseur.
L’interruption choisie dans le programme proposé est l’Auto-vecteur d’interruption de niveau 1 (adresse $64). Celle-ci gère normalement, entre autres, l’horloge et le VBL. Elle a lieu environ 120 fois par seconde, mais a priori la durée entre deux interruptions successives n’est pas constante. Pour réguler un peu cette irrégularité potentielle, seule une interruption sur deux est prise en compte.
La programmation du multitâche est alors habituellement assez simple : on sauvegarde tous les registres dans la pile système, on exécute la routine standard d’interruption, puis on appelle les routines spécifiques (gestionnaire d’événement et de l’écran dans le programme proposé), avant de laisser au scheduler le soin de déterminer la valeur du pointeur de pile système (SP) de la tâche dont on souhaite poursuivre l’exécution jusqu’à la prochaine interruption.
Dans le cas du Macintosh, ceci ne marche pas : les programmeurs d’Apple ont installé un ‘stack sniffer’, qui vérifie soixante fois par seconde (c’est-à-dire durant la routine

d'interruption VBL standard) si le SP n'a pas débordé de la zone qui lui était attribuée. Il faut donc modifier deux fois le SP : une fois avant l'appel de la routine standard en lui redonnant sa valeur 'normale', et une fois avant la fin de l'interruption en lui donnant la valeur correspondant à la tâche dont on souhaite poursuivre l'exécution.

Il faut prévoir aussi le cas où aucune tâche n'est active : il s'agit alors d'effectuer une routine qui ne fait rien, en attendant la prochaine interruption. Il faut encore prévoir une tâche de fin qui désinstalle le système (en l'occurence il s'agit de restaurer la valeur initiale du vecteur d'interruption). Le programme proposé utilise un seul sous-processus pour ces deux fonctions. Ce sous-processus est 'fantôme' : il est considéré comme appartenant au processus système mais n'apparaît pas dans la liste de ses fils et n'est donc pas examiné par le scheduler comme les autres sous-processus. Il n'est exécuté que lorsque le scheduler détecte que l'on demande à quitter le système ou lorsqu'il n'y a pas de sous-processus actif.

2. Structures utilisées.

a. gestion de la mémoire

La gestion mémoire a été simplifiée pour tenir compte des capacités du Macintosh. Les routines Malloc et Free ont été refaites (à cause des bugs relevés dans les routines fournies avec Think C).
Le système tient à jour une liste des blocs alloués. Chaque liste contient les adresses de début et de fin du bloc, et l'ID du processus propriétaire. Pour éviter une limitation au nombre de blocs allouables, ces données sont stockées au début de chaque bloc.

b. processus et sous-processus

Le système tient à jour la liste des processus installés. De même chaque processus tient à jour la liste de ses fils. Ce sont des listes avec pointeurs dans les deux sens pour accélérer le traitement de certaines fonctions (notamment celles de destruction).

Les pointeurs vers les processus et les sous-processus sont en fait des entiers (c'est tout simplement l'ID). La gestion des ID est la suivante : le système gère un nombre déterminé d'ID (défini par nmaxid), et un 'curseur', qui parcourt circulairement les ID. A chaque ID est associée la valeur d'un pointeur, qui peut être :

- NULL lorsque l'ID n'est pas utilisé
- un pointeur vers un processus ou vers un sous-processus
- Hot lorsque l'ID vient d'être libéré.

Lorsqu'on demande un nouvel ID, le curseur avance jusqu'au premier ID libre. Ce faisant il change les ID 'Hot' en NULL. Pour rendre un ID, il suffit de mettre 'Hot' dans le pointeur qui lui est associé.

Lorsqu'on détruit un sous-processus, seules ses 'comm' en attente dans la liste de réception sont rendues au système. Lorsqu'on détruit un processus, les objets, les blocs mémoire et les canaux de fichiers lui appartenant sont rendus au système.

Le fonctionnement du scheduler est le suivant :
- le scheduler tient à jour régulièrement un registre 'total' qui contient la valeur maximale de la priorité des processus installés.
- pour choisir quel processus exécuter, le scheduler ajoute au registre 'kprio' de chaque

processus sa priorité ‘pr’. Si ‘kprio’ devient plus grand que ‘total’, ‘total’ est retranché à ‘kprio’ et le scheduler détermine ensuite de manière similaire quel sous-processus exécuter.
- si aucun sous-processus actif n’est découvert, le processus est placé en attente.
On constate que la méthode employée consiste, en “régime permanent”, à effectuer une sorte de division qui rend l’exécution de chaque processus proportionnelle à sa priorité.

c. communications inter-process.

La gestion des communications s’effectue autour de plusieurs listes.
- la liste des cases ‘comm’ libres (pour éviter un usage excessif de malloc, un certain nombre de structures ‘comm’ sont réservées lors de l’installation du système).
- les listes de réception (une par sous-processus) dans lesquelles sont stockées les ‘comm’ reçues. A chaque modification du masque de réception, la liste de réception est vidée des ‘comm’ qui ne sont plus autorisées par le masque.

d. objets .

Deux listes permettent de gérer les objets :
- le système tient à jour la liste des objets créés par l’ensemble des processus. Elle permet de détruire les objets d’un processus lors de la destruction de celui-ci.
- une seconde liste contient les objets qui sont posés sur le bureau. Cette fois, l’ordre est primordial : le premier objet de cette liste est celui qui se trouve au premier plan du bureau.

3. Utilisation de l’assembleur.

La majeure partie du projet a été réalisée en C (remarquons au passage l’absence en plus de 10.000 lignes de tout ‘goto’, ‘break’ et autre ‘continue’). Cependant l’assembleur est présent dans certaines routines, pour diverses raisons.

a. routine d’interruption.

La routine d’interruption est obligatoirement programmée en assembleur car :
- il faut manipuler la pile système : sauvegarde de registres, modification du pointeur
- il faut appeler la routine standard du Mac et en contrôler le retour : le RTE qui finit la routine standard doit se comporter comme un RTS.
- il faut contrôler le retour de l’interruption, pour l’aiguiller vers une tâche différente de celle qui a été interrompue.

b. routine de création d’un sous-processus

La routine de création d’un sous-processus comporte également quelques lignes en assembleur. En effet, il faut remplir la pile système du nouveau sous-processus. Celle-ci sera utilisée pour terminer l’interruption au cours de laquelle le sous-processus aura été choisi pour la première fois par le scheduler. Dans cette pile, on doit donc assigner dès le début des valeurs aux registres du microprocesseurs (valeurs quelconques, sauf pour A5, que le C utilise pour un usage particulier), et préparer le retour de l’interruption. Attention à la compatibilité entre les 680x0 : à partir du 68010, des informations supplémentaires sont stockées dans la pile lors d’une interruption. La routine proposée fonctionne cependant

avec tous les 680x0.

c. routines graphiques.

Les routines graphiques doivent être particulièrement rapides, notamment celle qui 'rafraîchit' l'écran. L'assembleur est de plus parfaitement adapté au traitement graphique, notamment avec un écran de type monochrome.

III. MISE EN OEUVRE

Introduction

L'objet de cette partie est de montrer comment programmer une application en C qui utilise l'environnement multitâche et multi-fenêtrage. On distinguera les opérations suivantes : inclure l'environnement dans l'application, gérer des tâches, gérer des fenêtres, utiliser les communications inter-processus.

1. Processus main

Tout d'abord, il faut inclure dans le projet les routines 'maclib', 'objets.c', 'process.c', 'desk.c' et 'utildesk.c'. La bibliothèque des fonctions de l'environnement se trouve dans 'desk.h'. Toute application développée pour le système doit donc comporter :

```
#include <desk.h>
```

Ensuite, l'appel de l'environnement se fait par :

```
desk(   nom_du_premier_processus,
    priorité_du_premier_processus,
    routine_du_premier_sous-processus,
    priorité_du_premier_sous-processus,
    taille_de_la_pile_système_du_premier_sous_processus,
    paramètre_quelconque (NULL par exemple)
    );
```

Cette routine installe le système, initialise et affiche le bureau, crée la processus système, et lance le premier processus utilisateur. Elle ne rend la main que lorsqu'un processus le demande : le système du mac est alors totalement restauré et l'application peut se poursuivre normalement. Celle-ci peut de nouveau appeler la procédure `desk()`.
Quelques indications :
- le nom du processus est une chaîne d'au plus 15 caractères

- à titre indicatif, la priorité du processus système vaut 256
- la routine du sous-processus est une procédure normale du C (void), qui doit cependant s'achever par EndSProcess() ou EndProcess()
- la taille de la pile système doit être choisie en fonction du nombre de variables locales utilisées, et de la récursivité de certaines routines : on peut considérer 4Ko comme une valeur moyenne
- le paramètre est du type (void*) : il sera accessible par le premier sous-processus par la fonction `Getparam()`. Son intérêt dans la fonction `desk()` est a priori limité.

Dans l'application proposée avec ce projet, le premier processus gère le menu des divers programmes. Il utilise le fait que le gestionnaire d'événement envoye 'dans le vide' une communication de type 'CClick' lorsque l'utilisateur clique sur le fond du bureau. On peut donc faire en sorte qu'un processus la reçoive et gère alors un menu déroulant (qui propose entre autre de quitter l'environnement).

A noter : un processus peut demander à quitter l'environnement en mettant à 1 le bit 15 de sys->state.

```
    sys->state|=0x8000;
```

2. Programmation des tâches.

La gestion des tâches est la base de l'environnement. Elle s'articule autour d'un nombre réduit d'instructions de création et de destruction.

Rappelons brièvement ce qui a été esquissé plus haut ainsi que quelques règles :

_ Un processus ne correspond pas à une procédure de C, c'est simplement une structure regroupant quelques paramètres dont la liste des sous-processus.

_ Par contre, un sous-processus correspond à une procédure de C et à quelques paramètres qui resteront invisibles à l'utilisateur.

_ Un processus contient au moins un sous-processus : il est détruit lorsque la liste de ses fils est vide.

_ La création d'un processus s'accompagne donc de celle d'un premier fils.

_ Un sous-processus ne peut créer qu'un sous-processus frère ou un autre processus.

_ Processus et sous-processus sont identifiés par un ID. Un sous-processus peut à chaque instant connaître son ID et celui du processus dont il est le fils en lisant respectivement les variables `sprocess` et `process` .

L'instruction de **création d'un sous-processus** frère est :

```
int CreateSProcess( routine_du_sous-processus,
           priorité_du_sous-processus,
           taille_de_la_pile_système_du_sous_processus,
           paramètre_quelconque (NULL si aucun)
           );
```

Cette instruction retourne l'ID du sous-processus créé : il vaut Nid si la création a échoué (lorsqu'il n'y a pas assez de mémoire). L'utilisation du paramètre se révèle cette fois particulièrement intéressante.

L'instruction de **création d'un nouveau processus** est :

```
int CreateProcess(  nom_du_premier_processus,
            priorité_du_processus,
            routine_du_sous-processus,
            priorité_du_sous-processus,
            taille_de_la_pile_système_du_sous_processus,
            paramètre_quelconque (NULL si aucun)
            );
```

Cette instruction retourne l'ID du processus créé : il vaut Nid si la création a échoué (lorsqu'il n'y a pas assez de mémoire). L'utilisation du paramètre peut se révéler encore ici très intéressante.

Pour **détruire un sous-processus**, il suffit de faire :

```
    KillSProcess(ID_du_sous-processus);
```

De même, pour **détruire un processus** (et du même coup tous les sous-processus fils), on fait :

```
    KillProcess(ID_du_processus);
```

Attention : ces routines ne doivent pas être utilisées pour **se détruire soi-même**. Pour faire cela, on utilise respectivement :

```
    EndSProcess();
et
    EndProcess();
```

On doit donc trouver ces routines à la fin des sous-processus (ne jamais l'oublier).

Pour **récupérer le paramètre** passé lors de la création du sous-processus, il suffit de faire :

```
    GetParam();
```

Ceci retourne le paramètre (type : void*).

Les **autres routines** accessibles sont moins importantes :

```
SuspendSProcess(ID) : suspend le sous-processus ID
SuspendProcess(ID)  : suspend le processus ID
ContinueSProcess(ID)    : libère le sous-processus ID
ContinueProcess(ID) : libère le processus ID
LevelSProcess(ID,n) : donne la priorité n au sous-processus ID
```

```
LevelProcess(ID,n)  : donne la priorité n au processus ID
```

Attention : un sous-processus ne peut se suspendre, il peut par contre se mettre en attente (voir 4ème paragraphe).
La minuterie étant liée aux communications inter-processus, elle sera étudiée au 4ème paragraphe.

Chaque sous-processus peut **demander une zone mémoire** au système. Les instructions qui permettent ce type d'opération sont `Malloc(long)` et `Free(void*)`.

3. Utilisation des fenêtres.

La gestion des objets, et notamment des fenêtres s'organise autour de quelques instructions simples. Le principal point à retenir est qu'aucune opération ne se fait directement sur l'écran, et que pour faire apparaître les modifications apportées, il faut utiliser les routines de rafraîchissement.

L'instruction de base est celle qui permet de **créer une nouvelle fenêtre** :

```
objet newwind(  largeur,
        hauteur,
        coordonnée_x_du_coin_supérieur_gauche,
        coordonnée_y_du_coin_supérieur_gauche,
        nombre_de_registres(généralement 0),
        nombre_de_'work' (généralement 1),
        nombre_de_cases_logiques (voir large : 10
                    par exemple )
        );
```

Si on met 0x8000 en coordonnée x, la fenêtre est positionnée au hasard. 0 en nombre de 'work' est interprété comme 1.
Le dessin de la fenêtre ainsi créée est initialisé, avec juste un cadre, en mode blanc sur noir. Attention, cette fenêtre n'est pas empilée sur le bureau : il est normal qu'elle n'apparaisse pas.

Pour créer un **habillage standard**, il y a deux fonctions :

```
cas setbarre(   objet_à_habiller,
        nom_de_la_fenêtre,
        ID_du_sous-processus_relié_à_la_case_de
            _destruction (Nid : pas de case
            de destruction)
        );
```

Cette routine définit une barre titre, une case de 'déplacement devant' (grande comme la fenêtre), une case de 'déplacement derrière', et éventuellement une case de destruction.Dans le cas de la création d'une telle case, un pointeur vers celle-ci est retourné par la fonction.

La **seconde instruction** inclut l'utilisation des ascenseurs :

```
void standard( ... ); (voir le détail dans <desk.h>)
```

Il convient toujours de faire attention au nombre de cases utilisées qui ne doit jamais dépasser celui déclaré lors de la création de la fenêtre : setbarre en nécessite au plus trois, standard au plus neuf.

On peut aussi **définir soi-même des cases logiques**, avec ou non du texte dans la case : fonctions `newc(...)` et `newcvide(...)` .

Le principe des ascenseurs est le suivant : on donne à l'ascenseur un pointeur vers un entier, et on définit une valeur maximale (qui correspond en gros à la longueur de l'ascenseur). L'entier est alors lié au curseur de l'ascenseur : il est déconseillé de le modifier directement ; par contre, on peut le lire à tout moment. De plus une comm de type 'CPnt' est envoyée là chaque déplacement du curseur de l'ascenseur ou seulement à la fin, lorsqu'on relâche le curseur (on choisit entre les deux modes lors de la création de l'ascenseur).

On peut **définir un sprite** avec la fonction :

```
objet newsprite(    largeur,
        hauteur,
        x,
        y,
        nombre_de_registres, (généralement 0)
        adresse_du_buffer_bitmap
        );
```

C'est l'équivalent de newwind(...) pour les sprites.

Pour **détruire un objet** : `killob(objet);`

Les objets ainsi définis n'apparaissent toujours pas à l'écran : il faut d'abord les empiler dans la pile bureau. Les principales instructions de gestion de cette pile sont :

```
void empile(objet,n);
```

Cette fonction **empile l'objet** à la n-ième place dans la pile ; le premier plan correspond à la place 0. Attention : le curseur de la souris est lui aussi dans la pile ; on empile donc habituellement un nouvel objet à la place 1.

```
int depile(objet);
```

Cette fonction **dépile l'objet** et retourne la place à laquelle il se trouvait (-1 si l'objet n'était pas dans la pile).

Tout ceci permet de placer un objet sur le bureau, mais le contenu de l'écran n'est pas modifié ; pour ce faire, il faut encore **rafraîchir l'écran**.

```
affscreen();    recalculer toutl'écran
```

```
affobjet(objet);    recalculer la zone occupée par l'objet
```

En résumé, l'utilisation d'un objet se résume souvent à ceci :

```
{ ...
    objet   ob;
ob=newwind(...);
empile(ob,1);
affobjet(ob);
... }
```

Les routines graphiques sont tout-à-fait classiques. Notons seulement qu'il convient de préciser à chaque fois l'objet et le numéro du 'work' dans lequel on travaille(usuellement, on travaille dans le 'work' 0).

Précisons encore que les routines graphiques sont exécutées dans le bitmap de la fenêtre : il faut de nouveau **rafraîchir l'écran** après des modifications du bitmap.
On pourraît faire un `affobjet(...)` , mais pour accélérer le programme, il vaut mieux utiliser `afworkmodif(objet);`
En effet, une fenêtre de modifications est gérée pour chaque objet. A chaque opération graphique dans cet objet, cette fenêtre est élargie de manière à contenir les nouvelles modifications du bitmap. La fonction afworkmodif() rafraîchit la zone correspondante de l'écran.

Les fonctions graphiques proposées sont de type : pset(), line(), box(), print() ...

4. Utilisation des communications inter-processus.

La gestion des communications est au coeur du système. Pour envoyer une communication, le plus simple est d'utiliser les routines de type 'SendInt', 'SendPnt', 'SendClick', 'SendClock', ... (il y en a une par type de comm). En plus des informations à transmettre, il faut fournir l'ID du destinataire. Comme il a été expliqué dans la partie I, on peut passer par les tables de communication par défaut.

Le problème est légèrement plus complexe pour la réception des 'comm' ; la procédure usuelle est la suivante : **le sous-processus se place en attente** de 'comm', en réglant son masque de réception. Ceci se fait par la commande :

```
long WaitforComm (long);
```

Ceci place le sous-processeur en état d'attente de communication et le traitement ne se poursuit que lorsqu'une comm compatible avec le masque est reçue : son type est retourné par la fonction. Le traitement continue suivant le type de la 'comm' reçue.

Par exemple :

```
{ ...
    long    k;
...
k=WaitforComm(CInt+CPnt+CClick+CClock);
if (k==CInt)    {   int i;
        i=GetInt();
```

```
        ... }
    if (k==CPnt)    {   void    *p;
          p=GetPnt();
           ... }
    if (k==CClick)  {   Mouse   m;
          GetClick(&m);
           ... }
    if (k==CClock)  {   int i;
          i=GetClock();
           ... }
    ... }
```

On peut **régler le masque de réception** par : `SetMaskComm(long);` mais ceci est également fait lors d'un `WaitforComm();`

Ce système permet bien sûr de communiquer avec un autre sous-processus, mais aussi de recevoir les événements.

Lors d'un click sur une case logique, une comm de type 'CClick' est envoyée (de même 'CClickE' pour une fin de click, 'CEnter' pour une entrée de zone, 'CQuit' pour une sortie de zone, 'CClickn' pour un n-click).

Les fonctions de lecture correspondantes sont GetClick(&m), GetClickE(&m), (...) où m est de type Mouse. Lors d'un click ou d'une fin de click, le système communique sur quel objet on a cliqué, sur quelle case, et les coordonnées du point cliqué dans cette case. Lors d'une entrée ou d'une sortie de zone, le système communique de quel objet et de quel case il s'agit.
Ces données sont accessibles par :

```
m.ob    : objet concerné
m.ca    : case concernée
m.dx    : coordonnée x dans la case (pour CClick et CClickE)
m.dy    : coordonnée y dans la case (pour CClick et CClickE)
```

A noter : dans le cas du n-click, m.dx contient n.

La **minuterie** est également organisée autour des communications. Il y a deux fonctions importantes :

```
OnceRing(n) : une comm de type 'CClock' sera reçue n/60 secondes
        plus tard (et c'est tout)
AlwaysRing(n)   : une comm de type 'CClock' sera désormais reçue
        toutes les n/60 secondes.
```

On peut interrompre la minuterie par : `AlwaysRing(0);`

Mentionnons pour finir les **opérations de fichier** : Fopen(), Fclose(), Getc(), Putc(),

Fseek(), Fread(), Fwrite() fonctionnent exactement comme leurs homologues ANSI. Le système peut gérer ‘nmaxfil’ fichiers en même temps.

CRITIQUE

1. Système de rafraîchissement

La puissance limitée a conduit à quelques ‘astuces’ qui compliquent un peu la programmation d’applications dédiées. La principale est certainement le rafraîchissement de l’écran.
Le système idéal pour le programmeur serait que le système recalcule lui-même l’écran lorsqu’un objet a été modifié. Ceci fonctionnerait bien sur le Mac lorsque les applications n’utilisent pas trop de fonctions graphiques. Le problème serait tout autre lors d’une accumulation de modifications de l’écran.
En guise d’illustration, le programme ‘Test_Rnd’ affiche aléatoirement 65.536 points, efface l’écran puis recommence. Pendant un tel cycle sur deux, le rafraîchissement de l’écran est commandé après avoir affiché 512 points. Pendant l’autre, il a lieu après l’affichage de chaque point. On peut constater la différence sensible de vitesse d’exécution entre deux cycles consécutifs.
On pourrait avoir l’idée de comparer cette différence de vitesse en lançant une seconde application ‘Test_Rnd’ lorsque la première a terminé son premier cycle : on pourrait alors s’attendre à ce que la première soit plus lente .On constate qu’il n’en est rien : effectivement le rafraîchissement est assuré pendant l’interruption.

On remarquera ainsi dans le programme ‘mandelbrot’ que le rafraîchissement n’est effectué qu’une fois que toute une ligne a été calculée, entre les deux boucles ‘for...next’.

2.Routines absentes

Faute de temps, un certain nombre de routines n’ont pu être programmées. Voici quelques suggestions, parmi beaucoup d’autres :
- routines graphiques pour les sprites,
- routines graphiques avec gestion d’échelles,
- gestion de fontes vectorielles,
- gestion plus poussée du clavier (touches spéciales, événement ‘touche relâchée’)
- gestion des liaisons séries :
    - identification des processus d’une autre machine
    - communications entre processus de machines différentes

Certaines routines de bases, auxquelles on accède sans difficulté sur la plupart des machines, sont a priori difficiles à implanter sur Mac, et seraient pourtant bien utiles :
- lecture du catalogue d’un disque
- lancement d’une application par une autre

Les procédures du système du Mac capables d’effectuer ce genre d’opération ne peuvent

s’empêcher d’afficher un certain nombre de choses à l’écran : fenêtre de selection des fichiers pour la première, barre de menu pour la seconde ... Ce qui les rend inutilisables par le projet.

3. gestion de la mémoire

Le système adopté est particulièrement sommaire. Ce choix est motivé par certaines raisons bien précises.
Commençons par le problème de la mémoire virtuelle, et l’utilisation de la MMU, que l’on peut utiliser à deux niveaux :

1. adressage virtuel pour chaque processus.

Il s’agit de créer un espace d’adressage particulier pour chaque processus : les processus peuvent ainsi s’exécuter tous à la même adresse. L’intérêt principal de cette méthode est de permettre une protection entre les diverses applications.
Cependant, l’adoption de cette méthode se traduirait tout d’abord par une perte de puissance : en effet il faudrait définir un arbre d’adressage pour chaque processus et donc en changer à chaque scheduling. Le câche dédié à l’adressage serait donc vidé une centaine de fois par seconde.
Autre problème : la plupart des routines du système du Mac sont court-circuitées dans le projet, mais il en reste encore quelques-unes : Button, GetMouse, _GetOSEvent, fopen, fclose,... A priori ces routines n’ont pas été prévues pour fonctionner en mode mémoire virtuelle.
Dernier problème : la création d’un arbre d’adressage logique nécessite une parfaite connaissance de l’organisation de l’adressage physique, qui varie sensiblement d’un Mac à l’autre. La carte de la mémoire du PB145 n’est à ma connaissance pas dans la documentation publique.

2. adressage virtuel commun

Il s’agit ici d’éviter les trous dans la mémoire qui peuvent apparaître éventuellement après un usage intensif de Malloc et de Free. En attribuant à chaque bloc une adresse logique, il est possible de déplacer les blocs pour combler les trous, sans prévenir les processus, qui travaillent toujours à la même adresse logique. Le câche n’est plus modifié un nombre important de fois par seconde : l’arbre d’adressage est le même pour tous.
Cependant les deux autres problèmes soulevés au §1 subsistent. Et il faut malheureusement abandonner cette idée séduisante.

On pourrait aussi réserver une zone mémoire lors de la création de chaque processus : les malloc des sous-processus se feraient alors dans cette zone. Ceci présente un inconvénient : une application n’utilise pas toujours la même taille de mémoire. Elle doit donc demander la taille maximale qu’elle est susceptible d’utiliser avec le risque de demander trop et de ne pouvoir alors être lancée.

Ainsi la place nécessitée par l’application ‘mandelbrot’ dépend principalement de la taille du dessin. Si on avait adopté ce système de gestion de la mémoire, on aurait dû limiter la surface de la fenêtre à une certaine valeur , rendant éventuellement impossible de calcul d’un petit dessin. La solution, certes rudimentaire, adoptée dans le projet permet de limiter simplement la surface du dessin à la taille de la mémoire disponible lors du début du calcul.

ANNEXE 1

Types de communication

CInt : entier (mot de 16 bits)
CLong : entier (long mot de 32 bits)
CPnt : pointeur de type (void*)
CError : code erreur (mot de 16 bits)
CFile : opération de disque (ne pas utiliser directement)
CMove : couple d'entier
CEnter : entrée de zone (lors du déplacement de la souris)
CQuit : sortie de zone (lors du déplacement de la souris)
CClick : click du bouton de la souris
CClickE : fin du click
CClickn : n-click
CChar : caractère (utilisé pour les comm du clavier)
CClock : horloge (mot de 16 bits)
CData : adresse d'un buffer

Ces différents codes sont des longs mots de 32 bits dont un seul bit est à 1. On peut donc les sommer pour obtenir un masque :
exemple :
`CInt+CChar+CClock` est un masque qui laisse passer les entiers, les caractères et l'horloge.

Les opérations de masque sont :

`long GetMaskComm()` : retourne le masque
`SetMaskComm(long)` : positionne le masque
(exemples : `SetMaskComm(CInt+CChar+CClock);`
`SetMaskComm(GetMaskComm()+CClick);`)
`long WaitforComm(long)` : attend la réception d'une comm de type précisé et retourne le type de la première reçue.
(exemple : `WaitforComm(CMove+CEnter);`)

ANNEXE 2

Structures utiles

1. Principales structures accessibles :

objet : pointeur vers un objet (fenêtre ou sprite)
-> retourné par newwind() et newsprite()
-> à indiquer lors des opérations graphiques, de pile et de destruction d'objet
->à utiliser lors du traitement des click et des changements de zone

exemple :
```
{   objet   ob;
...
ob=newwind(....);
empile(ob,1);
... };
```

cas : pointeur vers une case d'un objet
->retourné lors de la création de cases newc(), newcvide(), standard(), ...
->à indiquer lors d'opérations graphiques, de menus déroulant de de saisie de chaîne au clavier
->à utiliser lors du traitement des click et des changements de zone

exemple :
```
{   cas c;
...
c=newc(...);
notcase(ob,c);
... }
```

Mouse : structure de communication des opérations de la souris
-> à définir soi-même dans ses routines (12 octets)
-> à utiliser pour recevoir les comm concernant la souris

exemple :
```
{   Mouse   m;
...
GetClick(&m);
if (m.ob==ob)   {   ... } /*click sur l'objet ob*/
if (m.ca==c)    {   ... } /*click sur la case c*/
... }
```

2. utilisation plus poussée

mainp : pointeur vers le bloc bitmap d'un objet

exemple :

```
{   mainp   p;
...
p=caclmainp(ob);
x=p->xoe;   /* récupère la coordonnée x de la fenêtre */
p->typo&=0xDFFF;    /*met la fenêtre en mode noir sur blanc */
...
}
```

work : pointeur vers un bloc work

exemple :

```
{   work    w;
...
w=calcworkn(ob,0);      /*premier bloc work de l'objet ob*/
w->fonte=....   ;   /*modifie la fonte*/
...
}
```

struct Fen : fenêtre de rafraîchissement

-> permet de gérer le rafraîchissement de manière plus précise

exemple :

```
{   struct Fen  f,g;
...
clearfenclc(&f);    /*initialise la fenêtre*/
ajfenclc(&g,&f);    /*englobe la fenêtre g*/
ajobjclc(ob,&f);    /*englobe l'objet ob*/
affenclc(&f);   /*rafraîchit la zone f*/
... }
```

ANNEXE 3

Variables Système

On y accède par le pointeur ‘sys’ défini comme variable globale. Elle permet l’exploitation de certaines possibilités ; voici les plus intéressantes :

sys->maskf : pointeur vers le motif de fond de bureau(16 mots)
sys->fontesys : adresse de la fonte standard
sys->xml,sys->yml : position de la souris
sys->btl : état du bouton
sys->oblm : objet considéré comme étant la souris
sys->obms : nouvel objet à considérer comme étant la souris
sys->duree : demi-période du clignotement (en 1/60 s)
sys->state : état du système
sys->always : routine utilisateur exécutée à chaque interruption
sys->adec : adresse de l’écran
sys->voffs : offset pour passer d’une ligne de l’écran à l’autre
sys->nbln : nombre de lignes de l’écran
sys->nbcl : nombre de colonnes de l’écran

Détail de ‘sys->state’ :

b0=0 : la gestion des zones est désactivée (plus de CEnter, CQuit)
b1=0 : la gestion du bouton est désactivée (plus de CClick, CClickE, CClickn)
b2=0 : la gestion du clavier est désactivée (plus de CChar)
b3=0 : déplacement de la souris désactivé
b4=1 : seul l’objet apparaîssant au premier plan est reconnu par le gestionnaire d’événements
b5=1 : la souris est actuellement affichée
b15=1 : le système sera désactivé et désinstallé lors de la prochaine interruption

ANNEXE 4

Exemples d’application

Le projet propose, en plus des routines propres à l’environnement, quelques applications illustrant les possibilités offertes au programmeur. On peut ouvrir un nombre indéterminé de fois une ce ces applications.

1. Mandelbrot

Ce programme permet de dessiner un ensemble de Mandelbrot ainsi que des ensembles de Julia. Attention, la routine de calcul utilise la multiplication 32x32bits du 68020. Ce programme n’est donc pas exécutable sur un 68000.
Les fonctions accessibles sont :
- réglage de la profondeur de calcul par menu déroulant (ce réglage prend effet immédiatement, même lorsqu’un calcul est en cours)
- agrandissement : il suffit de sélectionner avec la souris la zone à agrandir. Le programme se débrouille ensuite pour conserver le même échelle sur les x et les y.

- agrandir la fenêtre : utiliser la case en bas à droite
- calcul d'un ensemble de Julia : cliquer sur la case Julia en maintenant le bouton enfoncé. Placer ensuite la souris sur un point du dessin (pour choisir la constante c). Lâcher le bouton. Ceci illustre la possibilité de passer des paramètres (ici la constante c) lors de la création d'un nouveau processus.

Les mêmes fonctions sont accessibles sur Julia.

2. Othello

Cette application est tout simplement un jeu d'othello programmé selon la méthode minimax avec alpha-bêta. Il contient un peu d'assembleur : affichage du tableau de jeu routines de base et fonction d'évaluations sont ainsi sensiblement accélérés.
On peu régler le niveau de jeu des deux joueurs par un menu déroulant (en cliquant sur les cases 'noirs' et 'blancs'. Le niveau correspond au nombre de demi-coups de profondeur.
La fonction d'évaluation tient compte de la différence de pions, du nombre de pions frontière, du nombre de coups possibles et de la disposition des pions sur le bord.

3. Parking

Ce programme démontre les possibilités de communication inter-process. Il fait intervenir les sous-processus suivants :
- 16 sous-processus identiques, un par voiture.
- 1 sous-processus de garde d'entrée
- 1 sous-processus de garde de sortie

Chaque voiture reste un temps aléatoire à l'extérieur du parking, demande ensuite à entrer, attend la permission, reste un temps aléatoire à l'intérieur, annonce sa sortie, et ainsi de suite.
Le programme utilise trois fenêtres : la première montre l'état de chaque voiture (à l'intérieur du parking, à l'extérieur ou dans la file d'attente), les deux autres indiquent les opérations des gardiens d'entrée et et de sortie. Le parking contient 5 places.

4. Scanning

Ce programme permet d'observer la gestion des tâches et de la mémoire.
La plan de la mémoire indique pour chaque bloc alloué le début et la fin du bloc ainsi que le nom du processus propriétaire. '.../...' comme adresse de fin signifie qu'elle est aussi l'adresse de début du bloc suivant. De même, '...' comme nom de processus indique que c'est le même que précédemment. Utiliser l'ascenseur vertical pour se déplacer dans la liste des blocs.
La liste des processus indique pour chacun son nom, son ID, son état et sa priorité. Il suffit de cliquer sur celle-ci pour pouvoir la modifier. On peut également cliquer sur un processus pour le selectionner et accéder ainsi à la liste des sous-processus.
La liste des sous-processus indique pour chacun son ID, son état, sa priorité (modifiable comme précédemment) et son masque de réception. Les sous-processus présentés sont ceux du processus sélectionné dans la liste des processus.

Ce programme permet de constater si une application est proprement programmée : l'idéal en effet est que tous les sous-processus soient en attente, sauf lorsqu'ils effectuent un calcul. Ainsi on remarquera que les sous-processus des voitures du programme 'parking' sont constamment en attente : leur attente aléatoire à l'extérieur ou à l'intérieur du parking n'est pas réalisée par une boucle à vide (l'état de ces sous-processus serait alors 'actif'), mais en utilisant la minuterie et donc en attendant un signal d'horloge (comm

de type CClock). Par conséquent, pendant cette attente, le scheduler les ignore.

5. Test du Rnd (fonction random)

Ce programme permet de tester la fonction random. Il affiche 65.536 points aléatoirement et compte le nombre de points distincts affichés : cette somme est donnée à la fin du cycle (à comparer donc avec 0x10000).

En fait le programme calcule une suite aléatoire (Un) d'entiers compris entre 0 et 255, et affiche les points de coordonnée (Un+1,Un).
Comme il a été expliqué dans la partie 'Critiques', ce programme permet essentiellement de montrer le rôle du rafraîchissement : lors de la première série de 65.536 points, le rafraîchissement est commandé tous les 512 points. Lors de la seconde, celui-ci est commandé après chaque point, ce qui permet d'apprécier la différence de vitesse et l'intérêt de laisser au programmeur la tâche du rafraîchissement.

6. Hochet

Ce petit programme a pour seule vocation de montrer la possibilité de définir un objet par rapport à un autre.
Pour "accrocher" l'objet 'fils' à l'objet 'pere', il suffit de faire :

```
fils->pere = pere;
```

Remarquons alors que pour faire passer un objet au premier plan ou à l'arrière plan, il faut aussi déplacer toute la descendance d'un objet (on pourra examiner les procédures `devant(objet)` et `derrière(objet)`).