Problèmes sur les disques

On a un Disque Western Digital Caviar AC2850 avec les spécifications suivantes:

Capacité = 850 MO
Temps de recherche minimal = 1msec
Temps de recherche moyen = 10 msec
Temps de recherche maximal = 22 msec

Vitesse essieu = 4500 rpm
Délai de rotation moyen = 6.6 msec
Taux de transfert maximal = 13.3 msec/piste ou 2419 octets/msec

Octets par secteur = 512
Pistes par cylindre = 16
Secteurs par pistes = 63
Cylindres = 1654

1) Supposons qu'on veut stocker un fichier contenant 60,000 enregistrements de taille fixe (80 octets chacun) avec la restriction que les enregistrements ne peuvent pas être distribués entre plusieurs secteurs. Combien de cylindres sont nécéssaires pour contenir le fichier ?

   Réponse :
   a. Chaque secteur peut contenir $\lfloor 512/80 \rfloor = 6$ enregistrements
   b. Le ficihier a besoin de 60,000/6 = 10,000 secteurs
   c. Un cylindre peut contenir 63 x 16 = 1008 secteurs
   d. Le nombre approximatif de cylindres nécéssaires est donc 10,000/1008 = 9.93 cylindres

2) Calculer la fragmentation interne causée par le fait qu'on ne peut pas distribué l'enregistrement entre plusieurs secteurs ?

   Réponse

   a. Chaque secteur contient 512 – 6 x 80 = 32 octets non utilisables
   b. Comme le fichier a besoin de 10,000 secteurs, 32 x 10,000=320,000 octets sont donc perdus à cause de la fragmentation interne

3) Supposons que le fichier est lu séquentiellement secteur par secteur. Combien de temps on a-t'il besoin pour lire les 60,000 enregistrements dans le pire des cas ?

Réponse :

a. Pendant une lecture séquentielle, une piste est lue au plus dans,
Temps de recherche maximal + délai de rotation maximal + taux de transfert maximal = 22 msec + (6.6 msec x 2 (le disque doit faire une rotation complète dans le pire des cas)) + 13.3 msec = 48.5 msec

b. Puisque Chaque cylindre contient 16 pistes, le fichier maintient 9.93 cylindres x 16 pistes/cylindre = 158.88 pistes ≈ 159 pistes

c. Dans le pire des cas la lecture séquentielle du fichier va prendre 159 x 48.5 msec = 7711.5 msec = 7.72 secondes

4) Supposons que l'accès au secteur est aléatoire et non séquentielle. Quel est le temps nécessaire pour lire tout le fichier, dans le pire des cas ?

Réponse :

a. Si l'accès est aléatoire, chaque secteur est lu pendant
Temps de recherche maximal + Délai de rotation maximal + Taux de transfert maximal/secteur = 22 msec + (6.6 msec x 2) + 512/2419 msec = 35.42 msec

Noter que le taux maximum de transfert par secteur est calculé comme suit :
2419 octets → 1 msec
512 octets (1 secteur) → 1 x 512/2419 msec

b. Dans le pire des cas et comme le fichier a besoin de 10,000 secteurs, le fichier va etre lu pendant
10,000 x 35.42 = 345,200 msec = 354.2 sec ≈ 6 minutes

Noter la grande différence entre l'accès séquentiel et aléatoire.

Problèmes avec les bandes magnétiques

On a une bande magnétique à 9-pistes avec les caractéristiques suivantes :

- densité : 1600 bits par pouce (bpp) par piste
- vitesse : 150 pouces par seconde (pps)
- l'éspace inter-bloc : 0.5 pouce

1) Supposons qu'on veut stocker un fichier d'enregistrements de taille fixe (80 octets/enregistrement ; le même fichier qu'avant). Si un facteur de bloc de 25 est utilisé, combien de bandes on a-t'il besoin ?

   Réponse :

   a. La taile physique d'un bloc de données est
      b = (25 * 80) / 1600 = 2000/1600 =1.25 pouces

   b. Le nombre de blocs de données requis pour sauvegarder le fichier est
      n = 60,000/25 = 2400

   c. L'espace inter-bloc est 0.5 pouce

   d. L'espace requis pour sauvegarder le fichier est
      s = n * (b + g) = 2400 * (1.25 + 0.5) =
      4200 pouces = 4200/12 pieds = 350 pieds

2) Quelle est la densité effective d'enregistrement ?

   Réponse :

   a. Nombre d'octets par bloc / nombre de pouces requis pour stocker un bloc
      = 25 * 80 /1.75 = 1142.9 octets/pouce

3) Quel est le taux effectif de transmission ?

   Réponse :

   a. Taux effectif d'enregistrement * vitesee de la bande
      = 1142.9 * 150 = 171,435 octets/sec ≈ 171.44 Koctets/sec

C++

La Surcharge des Opérateurs

Définition : la surcharge des opérateurs est la possibilité syntactique que C++ offre pour rédéfinir les actions d'un opérateur dans un contexte donné.

Pourquoi utilisons-nous la surcharge ?

Les opérateurs C++ ne s'appliquent pas généralement aux objets des classes : ils s'appliquent seulement aux types prédéfinis. Par exemple, l'opérateur d'égalité "= =" n'accepte pas des chaines de caractères comme des opérandes – cela signifie qu'au lieu d'écrire :
if (a = = b) … (a et b sont des chaines de caractères)
on a besoin d'ecrire des expressions plus compliquées comme
if (strcmp(a.string,b.string) = = 0) … or
if (compstr(a,b) = = 0) …

La surcharge permet à redéfinir l'opérateur "= =" pour les chaines de caractères et à écrire des expressions come "if (a = = b)" (a et b sont des chaines de caractères).

Exemple

Soit "rat" une classe des nombres rationels, c-à-d, fractions avec numérateur et dénominateur, tous les deux sont des entiers.

```
class rat
{
        int z;    // numérateur
        int n;    // dénominateur
        public :
        rat(int zr=0, int nr=1) : z(zr), n(nr) {}     // contructeur

        void show()
        {
                if ((long)z*n<0)            // fraction négative
                        cout << "-";        // afficher le signe –
                if (z*n = = 0)              // la fraction à la valeur 0
                        cout << 0;
                else                        // autrement, afficher la fraction
                        cout << (z>0 ? z :-z) << "/" << (n>0 ? n :-n);
        }
};
```

Notes

1. Dans le constructeur ":z(zr),n(nr)" correspond à la liste d'initialisation, équivalente à

   ```
   {
           z = zr;
           n = nr;
   }
   ```

   dans le corps du constructeur

2. cout << (z>0 ? z :-z) <==>

   ```
   if (z>0)
           cout << z;
   else
           cout << -z;
   ```

Surcharge de l'opérateur ++

```
rat rat ::operator++()      // version préfixe
{
        z += n;             // ajoute 1 à z/n
        return(*this);      // retourne la copie
                            // de l'objet modifié
}

rat rat ::operator++(int)   // version suffixe
{
        rat tmp=*this;      // sauvegarde l'ancienne valeur
                            // de l'objet
        z += n;             // ajoute 1 à z/n
        return(tmp);        // retourne la copie de
                            // l'objet non modifié
}
```

Usage

```
rat a1(3,5), a2(3,5), b;

b = ++a1;       // appelle la version préfixe de l'opérateur ++
                // a1 est incrémentée avant qu'elle soit assignée
                // à b qui reçoit la valeur 8/5
```

```
b = a2++;       // appelle la version suffixe de l'opérateur ++
                // b reçoit la valeur de l'objet a2
                // avant d'être incrémentée, c-à-d, b
                // reçoit la valeur 3/5
```

Note : dans la version suffixe, le paramètre formel "int" est présent pour indiquer que "++" vient après l'objet est pas le contraire – c'est sa seule fonction.

Exemple

```
cout << "a\tb\n";
b = ++a1;
a1.show();
cout << "\t";
b.show();
cout << "\n";

b = a2++;
a2.show();
cout << "\t";
b.show();
```

résultat affiché :

```
  a       b
8/5     8/5
8/5     3/5
```

Surcharge de l'opérateur "mettre dans" <<

Soit la classe "Person" définie dans person.h comme :

```
class Person
{
        public :
                // constructeurs et déstructeurs
                Person();
                Person(char const *n, char const *a, char const *p);
                ~Person();

                // fonctions d'interface
                void setname(char const *n);
                void setaddress(char const *a);
                void setphone(char const *p);
                char const *getname(void) const;
                char const *getnaddress(void) const;
                char const *getphone(void) const;
```

```
    private :
            // champs de données
            char *name;
            char *address;
            char *phone;
};
```

Note : les fonctions getname, getaddress et getphone sont des fonctions membre constantes, c'est pourquoi le mot-clé **const** apparaît après la liste de paramètres - les fonctions membre constantes ne modifient pas les champs de données de l'objet, mais elles les inspectent seulement.

Si on veut appeler le code suivant ,

```
Person kr("Ken Ross", "unknown", "unknown");
Cout << "Nom, adresse et numéro de téléphone de la personne kr : \n" << kr << "\n");
```

Donc on doit surcharger "<<" pour qu'il prend kr comme un argument.

L'opérateur "<<" et les deux opérandes (ostream & et Person &) interviennent dans l'instruction cout << kr. L'action proposée est définie par
opérateur fonction opérateur <<( ) utilisant deux arguments:

```
// déclaration dans person.h
ostream &operator<<(ostream &, Person const &);

// définition dans un fichier source
ostream &operator<<(ostream &stream, Person const &pers)
{
        return
        (
                stream << "Nom :                " << pers.getname()
                        << "Adresse :           " << pers.getaddress()
                        << "Téléphone:          " << pers.getphone()
        );
}
```