RICM 2ème année

Janvier 2010

Examen de Système d'Exploitation

Durée : 2h. Documents autorisés, à l'exception des livres. Le barème est donné à titre indicatif.

I. Gestion de mémoire

Question 1 (1 pt)

Combien de défauts de page sont produits avec l'algorithme de remplacement FIFO pour la suite de références de pages suivantes, en supposant que l'on dispose de 4 blocs en mémoire centrale?

1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

Question 3 (3 pts)

On considère un système de gestion de mémoire qui opère selon un principe d'allocation contiguë de zones de taille variable (la taille est exprimée en termes d'un nombre de blocs de taille fixe, ici 1K).

Une requête d'allocation d'une zone de mémoire contiguë de taille *nK* effectuée pour le compte d'un processus Pi sera notée Pi :+n (respectivement Pi :–n). On considère maintenant un état du système dans lequel la liste des zones libres est composée d'une zone de taille 1200K. Indiquer pour les algorithmes *first-fit* et *worst-fit* les partitions successives de cette zone, pour la liste de requêtes suivantes :
P1 :+400, P2 :+150, P3 :+350, P2 :-150, P4 :+100, P5 :+250

On exprimera le résultat sous la forme suivante :

Zones : (Libre,1200) // 1 seule zone libre au départ
Requete : P1 :+400
Zones : (P1,400),(Libre,800)
Requete : P2 :+150
Zones : ...

II. Gestion de fichiers (4 pts)

Le problème porte sur la conception d'un système de gestion de fichiers (SGF). Du point de vue d'un processus utilisateur, un fichier est considéré comme une suite d'**articles** de **taille fixe** accédés **séquentiellement.** On s'intéresse aux opérations suivantes (sous-ensemble des opérations classiques):

***int ouvrirFichier (String nom)* :** ouvre le fichier de nom symbolique *nom* et retourne un identifiant entier. Le pointeur courant est positionné au début du fichier.
void positionnerFichier(int fid, int pos) : place le pointeur courant sur l'article *pos* pour le fichier désigné par l'identifiant *fid.*
void lireFichier (int fid, byte[] article) : lit l'article suivant dans le fichier désigné par l'identifiant *fid*.

Toutes ces méthodes renvoient l'exception *FileException* si l'un des arguments (e.g., *fid*) est erroné.

Un fichier est réalisé physiquement comme un ensemble de **blocs de taille fixe** (BLOC-SZ octets) sur disque. Les blocs ne sont pas nécessairement disposés de façon contiguë sur le disque. Un bloc contient précisément ***N* articles** (il n'y a pas de fragmentation interne). Chaque fichier dispose en outre d'un **descripteur physique** qui contient des informations générales (taille du fichier, nom du propriétaire, date de création, etc), ainsi que la *Table d'implantation sur disque* du fichier qui décrit l'adresse sur disque de chaque bloc physique. Une adresse sur disque correspond à l'index d'un numéro de bloc physique sur le disque.

Le descripteur d'un fichier est physiquement stocké dans un bloc disque (on suppose que la taille d'un bloc permet de stocker l'ensemble d'un descripteur). Une fois chargé en mémoire, un bloc contenant un descripteur est manipulable au travers des fonctions suivantes :

int getTaille(byte[] fileDesc) : renvoie la taille du fichier en nombre d'articles.

int getAdBloc(byte[] fileDesc , int blocId) : renvoie l'adresse disque du bloc d'index *blocId*.

Le système maintient un **catalogue** qui décrit la correspondance entre noms symboliques de fichiers et les adresses de leur descripteur sur disque. Chaque entrée du catalogue est un couple *<nom symbolique, ad-descripteur-disque>*. On suppose que le catalogue est lui-même entièrement contenu dans un bloc disque préalloué. Le système dispose de la fonction suivante pour gérer les catalogues :

int chercher(String nom) : cherche le nom *nom* dans le catalogue et retourne l'adresse du descripteur sur disque, ou -1 si le nom de fichier n'existe pas dans le catalogue.

Le système dispose de la fonction suivante pour gérer les entrées depuis le disque :

void lireBloc (int adBloc, byte[] buffer) : lit le bloc d'adresse *adBloc* dans le buffer passé en argument.

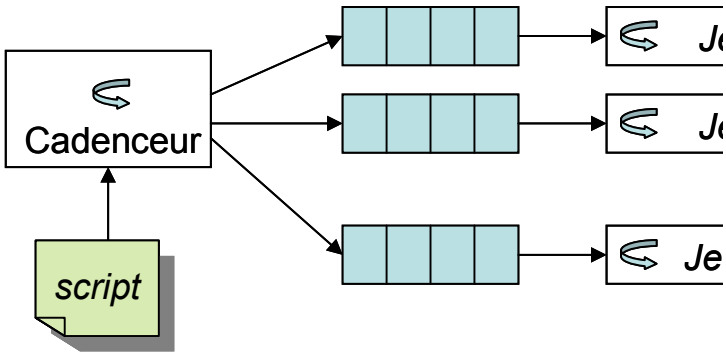
Question. Décrire les principes de mise en œuvre (structures de données et pseudo-algorithmes) des primitives *ouvrir-fichier, positionner-fichier* et *lire-fichier* en utilisant les fonctions systèmes introduites ci-dessus pour la gestion des catalogues et des blocs sur disque. Dans cet exercice, on ne cherchera pas à optimiser (par gestion de caches) les échanges entre le disque et la mémoire.

III. Problème (12 pts)

On s'intéresse au contrôle automatisé d'un ensemble de *N* jets d'eau au travers d'un programme Java dont la classe principale est appelée *GestionJets*. Chaque jet est contrôlé par un *thread* (de la classe *Jet*), qui a la capacité d'exécuter des commandes permettant de démarrer, arrêter, et régler le débit d'eau du jet.

Un *script d'animation* du jet est défini par une suite de commandes. Toute commande est associée à l'identification du jet concerné par la commande ainsi qu'au moment auquel la commande doit être déclenchée, exprimé en secondes par rapport à la date de lancement de l'animation. Les commandes sont ordonnées par date de déclenchement.

Au lancement du programme *GestionJets*, un thread particulier, appelé *Cadenceur*, est chargé de déclencher les commandes au bon moment. Chaque commande devant être exécutée par un jet donné (d'identifiant *J* compris entre 0 et N-1) est envoyée au jet *J* au travers d'un *buffer* de type producteur-consommateur. Cet envoi est effectué par le *Cadenceur* au moment où une commande doit être déclenchée. La figure suivante illustre cette architecture logicielle. On notera que l'on utilise un *buffer* de type producteur-consommateur par jet.



Exemple de Script

Moment	Commande	Jet
0	Start	0
0	Start	1
10	Stop	0
10	Start	2
10	Start	3
30	Terminate	0
30	Terminate	1
30	Terminate	2
30	Terminate	3

Question 1. Compléter le pseudo-code Java de la classe principale *GestionJets*, chargée de créer les structures de données et de contrôle nécessaires à la gestion des jets (création et démarrage des threads, création des buffers de production-consommation, etc). On suppose disponible la classe *ProdCons* telle qu'utilisée dans le TP. Dans un premier temps, on ne s'intéresse pas à la gestion de la terminaison du programme.

Question 2. Compléter le pseudo-code du *Cadenceur*, qui doit déposer les commandes à exécuter dans les buffers de production-consommation des jets au moment adéquat, en fonction des moments de déclenchement associés aux commandes. Pour ce faire, le *Cadenceur* maintient une horloge interne comptabilisant (de manière approximative) le nombre de secondes écoulées depuis le lancement du programme.

Question 3. On suppose que tout script se termine par l'envoi de la commande *Terminate* aux différents jets. En quelques lignes, complétez les classes *GestionJets*, *Jet* et *Cadenceur* de manière à gérer l'arrêt du programme lorsque toutes les commandes ont été exécutées.

Question 4. La commande *Start* est en fait associée à un argument indiquant le débit d'eau requis par rapport au jet concerné par la commande. Le débit global d'eau disponible pour l'ensemble des jets est stocké dans l'attribut public *debitGlobal* de la classe *GestionJets*. Chaque jet doit s'allouer le débit requis avant d'exécuter la commande *Start*. Les allocations et libérations de débit sont gérées au travers des méthodes *allouerDebit(int debit, int id)* et *libererDebit(int debit, int id)* de la classe *GestionJets*. Ecrire le peuso-code Java de ces méthodes correspondant à la solution directe.

Question 5. Afin d'optimiser la gestion de l'allocation de débit, on souhaite faire en sorte qu'un thread mis en attente ne puisse être réveillé que lorsque sa demande peut être servie. On demande de compléter les structures et méthodes suivantes définies dans la classe *GestionJets*.

```
private class requete {
    boolean traite = false;
    int debit;
    Requete(int debit) {this.debit = debit;}
}
 // List to add(Object) / remove(Object)
java.util.List requetes = new java.util.LinkedList();

public void allouerDebit(int id, int debit) {
    synchronized(this) {
        if (debit < debitGlobal) {
            debitGlobal -= debit;
        } else  {
            ...
    }
}
public void libererDebit(int id, int debit) {
        ...
        java.util.Iterator it = requetes.iterator();
        while (it.hasNext())
            Requete r = it.next();
                    ...
        }
}
```

Question 6. On souhaite fournir une commande *Incr* (resp. *Decr*) permettant d'incrémenter (resp. décrémenter) le débit d'eau au niveau d'un jet. Par exemple *Incr 3 100* demande à incrémenter le débit du jet 3 de 100 unités. Les incrémentations et décrémentations de débit sont gérées au travers des méthodes *allouerDebit(int debit, int id)* et *libererDebit(int debit, int id)* de la classe *GestionJets,* telles que définies à la question 4 (par souci de simplicité). Expliquez en quelques lignes (donner un scénario) pourquoi ces commandes peuvent engendrer un problème d'interblocage, qui n'était pas présent jusque-là.

Question 7. On souhaite détecter et guérir les situations d'interblocages. La détection repose sur l'usage de deux tables (*Acquis* et *Demande*) dans la classe *GestionJets*, qui mémorisent, pour chaque jet, les débits acquis et demandés. Un thread dédié, créé au lancement du programme est chargé de détecter les interblocages à intervalles de temps réguliers (*IB-DELAI* secondes).

A- Adapter les méthodes *allouerDebit(int debit, int id)* et *libererDebit(int debit, int id)* de la question 4.
B- Définir la méthode *detecterIB()* dans la classe *GestionJets*.
C- Définir la classe *Detecteur* mettant en œuvre le thread de détection.

Question 8. On suppose que chaque jet baisse automatiquement sa demande de débit de 10% (avec une baisse minimale de 1 unité) après chaque attente. Modifiez les méthodes *allouerDebit()* et *detecterIB()* en conséquence, de manière à assurer la guérison automatique des interblocages après leur détection.

```
-------------------------------------------
public class GestionJets {
    public int N = ..;
    public int debitGlobal = .. ;
    public Commande[] listeCommandes ;
    …

    static public void main(String args[]) {
        new GestionJets(args);
    }
    public GestionJets(String args[]){
        // args[0] contient le script à exécuter
        listeCommandes = lireCommandes(args[0]);
        …
    }
    …
}
-------------------------------------------

public class Cadenceur … {
    …

    public Cadenceur(GestionJets gj) {
        …
    }
    …
}
-------------------------------------------
public class Jet extends Thread{
    GestionJets gj ;
    int id ;   // identifant du jet


    public Jet(GestionJets gj, int id) {
        this.id = id;
        this.gj = gj;
    }

    public void run() {
        while (...) {
            Commande c = ...  // récupère la prochaine commande à exécuter
            ...
            exec(c);   // exécute la commande
    }
    …
}
-------------------------------------------

public class Commande {
    String name ;
    long id ;     // jet concerné par la commande
    long date ;   // date de déclenchement en secondes
    …
}
```

SOLUTION PROBLEME

```
public class GestionJets {
    public int N = ..;
    public Commande[] listeCommandes
    public ProdCons[] prodCons;
    public Jet[] jets;
    public Cadenceur cadenceur;

    public Integer prios[] = new Integer[N] ;
    …

    static public void main(String args[]) {
        new GestionJets(args);
    }
    public GestionJets(String args[]){
        // args[0] contient le script à exécuter
        listeCommandes = lireCommandes(args[0]);
```

```
        prodCons = new ProdCons[N];
        jets = new Jet[N];
        for (int i=0; i<N; i++){
            jets[i] = new Jet(this, i);
            jets[i].start();
        }
        cadenceur = new Cadenceur(this);

        // terminaison
        for (int i=0; i<N; i++)
            jets[i].join();


    }
    public synchronized void allouerDebit(int id, int debit) {
        while (debit >= debitGlobal)
            wait();
        debitGlobal -= debit;
    }

    public synchronized void libererDebit(int id, int debit) {
        debitGlobal += debit;
        notifyAll();
    }

    ******************** Version avec priorités ***************
    private class requete {
        boolean traite = false;
        int debit;
        Requete(int debit) {this.debit = debit;}
    }
    java.util.List requetes = new java.util.LinkedList();

    public void allouerDebit(int id, int debit) {
        synchronized(this) {
            if (debit < debitGlobal) {
                debitGlobal -= debit;
            } else  {
                r = new Requete(debit);
                this.Requetes.add(r);
                // ne pas faire wait(r) ici, cela bloquerait car
                // le verrou sur l'objet courant ne serait pas relâché

            }
         }
            // un autre thread peut avoir appelé liberer...
            if (r != null) {
             synchronized(r) {
                if (! r.traite)
                    r.wait();
             }
            }
    }

    public void libererDebit(int id, int debit) {
        synchronized(this) {
            this.debit += debit;
            java.util.Iterator it = requetes.iterator();
            while (it.hasNext()) {
                Requete r = it.next();
                synchronized(r) {
                If (r.debit <= this.debitGlobal)
                    it.remove();
                    this.debitGlobal -= debit;
                    r.traite = true;
                    r.notify();
                }
             }
          }
    }

******************** Version avec gestion des IB ***************
scenario IB :
    debitGlobal = 200
    J1 allouer(100)
    J2 allouer(50)
    J1 incr (60)
    J2 incr (60)
```

```
   Public class GestionJets {
    ...
     int[] Demande = new int[N];
    int[] Acquis = new int[N];

   Public GestionJets(..) {
       ...
       int[] Demande = new int[N];
       int[] Acquis = new int[N];
       for (int i=0; i<N; i++)
           Demande[i] = 0;
           Acquis[i] = 0;
   }

   public synchronized void allouerDebit(int id, int debit) {
       while (debit >= debitGlobal) {
           Demande[id] += debit;
           wait();
           debit -= max (debit div 10, 1);
       }
       debitGlobal -= debit;
       Demande[id] -= debit;
       Acquis[id] += debit;
   }

   public synchronized void libererDebit(int id, int debit) {
       debitGlobal += debit;
       Acquis[id] -= debit;
       notifyAll();
   }

   public boolean detecterIB() {
          for (int i = 0 ; i<N ; i++)
              if (gj.Demande[id] <= gj.debitGlobal)
                  return false;  // au moins un thread non bloqué donc pas d'IB
          }
          notifyAll() ;
          return true ;

    }

    ...

   }

   class Detecteur extends Thread {
   GestionJets gj ;
       public Detecteur(GestionJets gj) {
           this.gj=gj ;
       }
       public void run() {
           sleep(IB-DELAI * 1000) ;
           gj.detecterIB();
       }
}
-------------------------------------------------------------------------

public class Cadenceur extends Thread {
   long horloge = 0 ;
   …

   public Cadenceur(GestionJets gj) {
       this.setDaemon(true) ;
   }


   public void run () {
       for (int i=0 ; i < gj.listeCommandes.length ; i++) {
           Commande c = gj.listeCommandes[i] ;
           if (c[i].date <= horloge) {
               prodCons[c.id].deposer( c ) ;
               i++ ;
           } else {
               sleep((c[i].date – horloge) * 1000);
               horloge = c[i].date;
           }
   }
```

```
   …
}
-------------------------------------------------------------------------
public class Jet extends Thread{
    boolean terminate = false ;
    GestionJets gj ;
    int id ;

    // id contient l'identifant du jet
    public Jet(GestionJets gj, int id) {
        this.id = id;
        this.gj = gj;
    }

    public void run() {
        while (!terminate) {
            Commande c = gj.prodcons[id].consommer();
            if c.name.equals("terminate")
                terminate = true;
            c.exec();
        }
    }
    …
}
```