TD1 partie 1: Gestion des disques

Corrigé

Question 1

Le temps nécessaire pour exécuter les 5 requêtes en attente suivant leur ordre d'arrivée est équivalent à 3 tours et 7 secteurs. En effet, deux requêtes demandant la lecture de secteurs se chevauchant ne peuvent être traitées dans un même tour de disque. Les requêtes (1) et (2) sont traitées dès le premier tour. Les requêtes (3) et (4) sont traitées respectivement au second et troisième tour. Enfin, le traitement de la requête (5) sera achevé lorsque le septième secteur (piste 16, cylindre 53) sera passé sous les têtes de L/E lors du quatrième tour.

Question 2

Les quatre séquences de traitement permettant de satisfaire les 5 requêtes en un temps minimum sont:

(a) 4 2 5 1 3 (b) 4 1 2 5 3 (c) 5 2 4 1 3 (d) 5 1 2 4 3

Ces quatre séquences assurent la lecture des cinq enregistrements en 1 tour et 43 secteurs. Elles n'ont cependant pas toutes un comportement équivalent. Les séquences (a) et (c) sont à rejeter car l'enregistrement 1 est sauté lors du premier tour, rallongeant inutilement le temps de réponse à cette requête. D'autre part, la séquence (d) est préférable à la séquence (b) car le premier enregistrement lu est le plus proche des têtes de L/E. Cette séquence offre plus de souplesse en cas d'arrivée de nouvelles requêtes remettant en cause la séquence optimale de traitement. Ainsi, supposons qu'une nouvelle requête démarrant au secteur 7 arrive au contrôleur disque lors du traitement de l'enregistrement 5, elle pourra être traitée dans le même tour si on l'intègre à la séquence (d) alors qu'elle attendra le second tour si on l'intègre à la séquence (b).

Question 3

Soient d_i et f_i les numéros de secteur correspondant au début et à la fin de l'article i. Le temps d'attente T_{ij} (exprimé en nombre de secteurs) pour accéder à l'article j après avoir lu l'article i est:

$T_{ij} = d_j - f_i$ si $d_j \geq f_i$

$T_{ij} = Tr - (f_i - d_j)$ si $d_j < f_i$ avec Tr = temps d'une rotation (en nombre de secteurs).

Considérons le graphe complet dont les sommets sont les enregistrements à lire et dont les arcs sont valués de la manière suivante :

valuation arc (i,j) = $T_{ij} + L_j$ avec L_j = temps de lecture de l'enregistrement j.

Chercher une séquence optimale de traitement des requêtes revient à chercher un chemin de longueur minimale passant par tous les sommets du graphe. Ce problème, connu en théorie des graphes sous le nom de recherche d'un chemin hamiltonien, est de complexité non linéaire. Pour obtenir la séquence optimale, il faut en outre tenir compte de deux critères supplémentaires (cf. Question 2):

– Ne jamais sauter inutilement d'enregistrements au cours d'une révolution du disque. Ce critère permet d'éliminer les séquences (a) et (c) au profit des séquences (b) et (d) dans l'exemple précédent. Pour cela, on doit vérifier que:

$T_{ij} < T_{ik} + T_{kj} + L_k$ avec L_k = temps de lecture de l'enregistrement k.

Si cette condition n'est pas satisfaite, on peut insérer la lecture de l'enregistrement k entre celle des enregistrements i et j.

– Commencer la séquence par la requête la plus proche possible des têtes de L/E. Ce critère permet d'évincer la séquence (b) au profit de la séquence (d) dans l'exemple précédent. Pour cela, on élimine la requête la plus proche des têtes de L/E puis on applique l'algorithme sur les requêtes restantes. Si la séquence de traitement ainsi trouvée n'est pas optimale, on ré-exécute l'algorithme en supprimant la requête qui suit.

Pour présenter un intérêt pratique, le temps de réponse de l'algorithme doit être inférieur au temps de passage inter-secteurs. En effet, la séquence optimale doit être recalculée après chaque traitement de requête, afin de prendre en compte les nouvelles requêtes arrivées. L'algorithme doit donc permettre de traiter une nouvelle requête démarrant au secteur suivant immédiatement le dernier secteur lu pour le compte de la requête en cours. Le coût engendré par la simple recherche d'un chemin hamiltonien dans un graphe (sans même tenir compte des critères supplémentaires introduits dans le parcours du graphe) est incompatible avec le temps de passage inter-secteurs.

Question 4

La stratégie SLTF conduit à l'exécution de la séquence (5 1 3 4 2) en un tour et 44 secteurs. On peut démontrer que l'algorithme SLTF demande au plus un tour supplémentaire par rapport à l'algorithme étudié question 3. En effet, si aucune requête de la séquence ne demande la lecture de secteurs se chevauchant, la stratégie SLTF donne le même résultat que la stratégie optimale. Etant donné que deux requêtes ne peuvent pas se chevaucher sur plus d'un tour, un mauvais choix de la requête à traiter en premier conduit au maximum à un tour supplémentaire. Ce résultat, excellent compte tenu de la simplicité de l'algorithme employé, fait de la stratégie SLTF la meilleure candidate pour la gestion des disques à têtes fixes (DTF).

Question 5

Dans la stratégie SLTF, une requête peut attendre indéfiniment si de nouvelles requêtes la chevauchant arrivent au contrôleur disque et se placent devant les têtes de L/E. Pour éviter cette situation de famine, on peut soit traiter les requêtes par paquets, soit définir une borne supérieure pour le temps d'attente d'une requête et satisfaire cette requête quelle que soit sa position si la borne est dépassée.

Question 6

Le nombre de déplacements de bras engendrés par la stratégie SCAN est indépendant du nombre de requêtes. En cas de requêtes peu nombreuses, la stratégie SSTF produit des mouvements de bras peu nombreux mais désordonnés. Si les requêtes deviennent plus nombreuses la stratégie SSTF tend vers la stratégie SCAN. De plus, on peut remarquer qu'il existe un danger de famine (cf. Question 5) pour la stratégie SSTF. On préférera donc généralement la stratégie SCAN à la stratégie SSTF.

Une fois les têtes de L/E positionnées sur le cylindre choisi par la stratégie SCAN ou SSTF, on applique la stratégie SLTF (cf. Question 4) pour traiter les différentes requêtes portant sur ce même cylindre.

Question 7

La stratégie optimale de placement de toutes les données d'un fichier sur une unité DTM consiste à stocker ces données sur les pistes appartenant à un même cylindre. Ainsi, un seul déplacement de bras est nécessaire pour se positionner sur le cylindre contenant le fichier et permet des accès répétitifs à tous les articles de ce fichier sans déplacement de bras additionnel. Si la taille du fichier dépasse la capacité d'un cylindre, on le stocke sur des cylindres contigus.

TD1 partie 2: Représentation des données sur disque

Corrigé

Question 1

Une première solution consiste à marquer (par exemple par un bit) tous les blocs libres (i.e., trous) du fichier. L'insertion d'un article dans le fichier va alors nécessiter un parcours séquentiel du fichier jusqu'à atteindre un bloc libre de taille suffisante. Cette solution n'est pas envisageable dès que le fichier atteint une taille significative. Une solution plus optimale consiste à chaîner les blocs libres entre eux. Chaque bloc libre contient deux informations : sa taille et l'adresse disque du bloc libre suivant. Ces informations ne consomment pas de place puisque le bloc en question est, par définition, inutilisé. La recherche d'un bloc libre de taille supérieure ou égale à une valeur donnée se fait par un parcours de liste.

Question 2

Lorsque la taille du bloc alloué est supérieure à la taille demandée, le résidu devient à son tour un bloc libre dont la taille correspond à la différence entre la taille allouée et la taille demandée. La dégradation qui en résulte est l'accroissement du nombre de blocs libres de faible taille, appelée *émiettement de la mémoire*.

Une solution au problème d'émiettement de la mémoire est le recompactage d'un ensemble de blocs libres contigus en un bloc libre unique dont la taille est égale à la somme des tailles des blocs compactés. Dans la pratique, la probabilité que deux blocs libres soient contigus est cependant faible. On est donc amené à recompacter périodiquement les pages d'un fichier. Le principe consiste à déplacer les articles présents dans une page afin de les ranger de façon contiguë à partir du début de la page. Cette réorganisation a pour effet de regrouper tous les trous en fin de page, ceux-ci pouvant alors être fusionnés en un trou unique de taille importante.

Question 3

Quatre stratégies sont possibles pour rechercher un bloc libre dont la taille est supérieure ou égale à celle demandée:

- *First-Fit:* cette stratégie choisit le premier bloc de la liste dont la taille est compatible avec celle demandée. La liste des blocs libres n'est pas ordonnée en fonction de la taille des blocs. Cette stratégie à l'avantage de la simplicité mais l'utilisation des blocs libres est sous-optimale puisqu'un grand bloc peut être alloué pour insérer un article de faible taille. De plus, compte tenu du phénomène d'émiettement de la mémoire (cf. Question 2), les blocs de faible taille ont tendance à s'accumuler en début de liste et donc à ralentir les recherches.

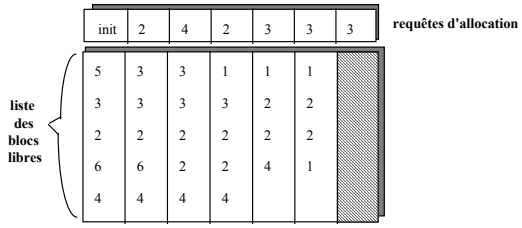
- *Next-fit*: cette stratégie est une optimisation de la stratégie First-fit. La liste des blocs libres est gérée ici comme une liste circulaire et chaque recherche dans la liste démarre à l'endroit où la dernière recherche s'est arrêtée. L'insertion d'un nouveau bloc libre se fait également à partir de la dernière position atteinte dans la liste. Cette stratégie évite l'accumulation des blocs de faible taille en début de liste.

- *Best-Fit:* Cette stratégie choisit le bloc dont la taille est la plus proche de celle demandée. La stratégie Best-Fit permet une gestion optimale de la place mémoire mais est coûteuse lors de la recherche. Si l'on trie la liste par taille de bloc croissante, la recherche est optimisée mais l'insertion d'un nouveau bloc libre s'en trouve ralentie. Pour optimiser insertion et recherche, on peut partitionner la liste par intervalles de taille. l'inconvénient majeur de la stratégie Best-fit est la création de blocs libres de si faible taille qu'ils peuvent difficilement être réutilisés.

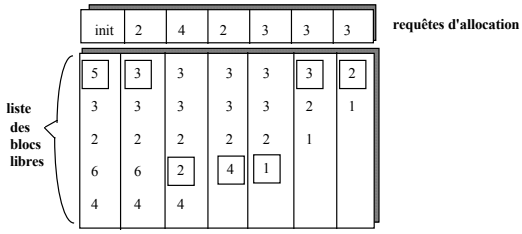
- *Worst-Fit:* la liste des blocs libres est triée par taille décroissante et le bloc sélectionné à chaque requête d'insertion est le premier de la liste. Cette méthode privilégie la rapidité d'allocation d'un bloc au détriment de l'optimisation de la gestion mémoire. L'insertion d'un nouveau bloc libre dans la liste doit respecter l'ordre du tri.

Question 4

L'évolution de la liste des blocs libres est résumée pour chaque stratégie par un tableau (cf. Figure 3.2). La colonne init montre l'état de la liste initiale des blocs libres. Chaque colonne suivante montre l'état de cette liste après exécution de la requête d'allocation correspondante. Une colonne hachurée indique que la requête d'allocation correspondante ne peut être satisfaite compte tenu des ressources disponibles dans la liste des blocs libres.

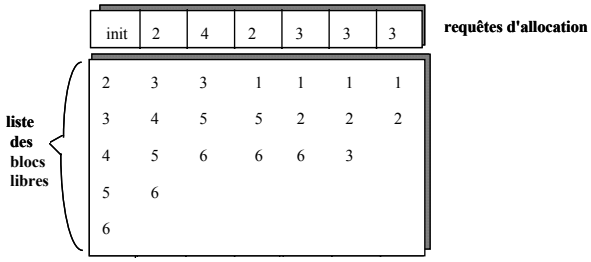


Stratégie First-fit

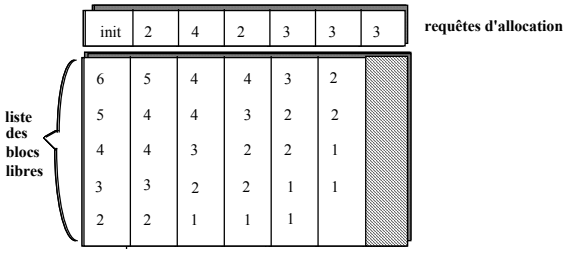


les chiffres encadrés indiquent la position courante dans la liste

Stratégie Next-fit



Stratégie Best_fit



Stratégie Worst-fit

Figure 3.2. *Comparaison des stratégies d'allocation*

Question 5

La stratégie d'allocation proposée consiste à partitionner l'espace disque en ensembles de blocs de taille fixe dont la taille est une puissance de deux. L'espace disque contient ainsi des blocs dont la taille (en unité d'allocation) s'échelonne de la façon suivante : 1, 2, 4, 8, 16, ..., 512 et ainsi de suite. Tous les blocs de même taille sont chaînés entre eux. Lors d'une demande d'allocation d'un bloc de taille *t*, la stratégie d'allocation choisira en fait un bloc de taille $2^{\lceil \log 2t \rceil}$, où $\lceil n \rceil$ dénote la partie entière supérieure d'un nombre n). Ainsi, une demande d'allocation d'un bloc de taille 9 se verra allouer un bloc de taille 16. Contrairement aux stratégies étudiées dans les questions précédentes, le résidu non occupé du bloc alloué n'est pas recyclé en bloc libre. Ce résidu constitue donc une place perdue tant que le bloc reste alloué.

L'avantage de cette stratégie réside dans sa simplicité. La recherche du bloc a allouer est immédiate et les désallocations de blocs n'engendrent pas d'émiettement de la mémoire, évitant par là même des techniques complexes de compactage. L'inconvénient de cette stratégie est par contre sa gestion sous-optimale de l'espace occupé. Notons cependant que la place perdue à chaque allocation reste proportionnelle à la taille du bloc alloué.

3.3.2. Représentation des données

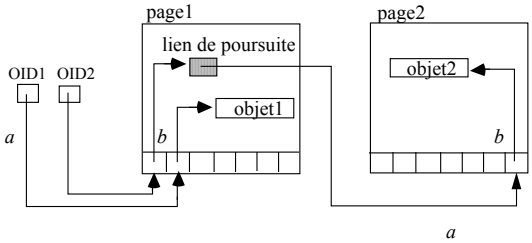
Question 6

La notion de pointeur disque peut être construite par l'intermédiaire d'adresses disque absolues (n° de cylindre, n° de piste, n° de secteur, déplacement en octets dans le secteur) ou d'adresses relatives dans un fichier (n° de page, déplacement en octets dans la page). Le décodage d'une adresse relative est plus coûteux car cette adresse doit être traduite en une adresse disque en accédant à la table des descripteurs de fichiers puis à la table des pages du fichier concerné. Les adresses relatives offrent cependant l'avantage d'être indépendantes du support physique de stockage. Ce paramètre est important si les données sont amenées à migrer d'un disque vers un autre. Cependant, ni adresses disque ni adresses relatives ne sont invariantes aux déplacements des articles référencés (suite par exemple à une mise à jour ou un compactage de page).

Question 7

L'invariance complète des pointeurs disque est une propriété fondamentale pour assurer la cohérence des références entre articles, surtout si l'on considère qu'un article peut être référencé de différents endroits (index, liens avec plusieurs articles, ...). La construction de pointeurs disque invariants, appelés communément *OID* (*Object Identifier*), peut être effectuée de différentes façons.

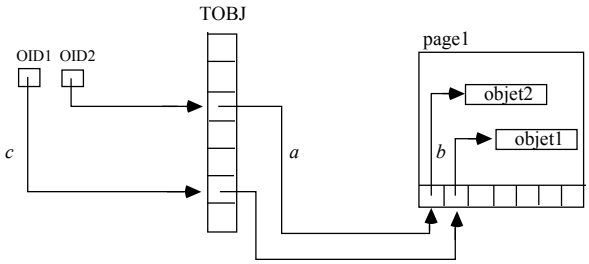
OID physiques : un index est maintenu en bas de chaque page et maintient les adresses, exprimées par un déplacement en nombre d'octets par rapport au début de page, de tous les articles stockés dans cette page. Un OID physique est de la forme (n° de page, n° d'entrée dans l'index). Si l'article change de place dans la page, suite à une mise à jour ou à un compactage de page, son entrée dans l'index est mise à jour. Si l'article est amené à changer de page au cours d'une mise à jour (par manque de place dans la page), on stocke la nouvelle adresse de l'article à l'emplacement qu'il occupait antérieurement dans la page. Cet information est appelée *lien de poursuite* ou *forwarder*. Un nouveau changement de page de l'article se traduit simplement par une mise à jour du lien de poursuite. Ce mécanisme est illustré Figure 3.3.



L'objet2 a changé au moins une fois de page alors que l'objet1 est toujours resté dans la même page. Une indirection de type b correspond à un simple positionnement dans la page alors qu'une indirection de type a correspond à l'accès à la page référencée puis au positionnement dans l'index.

Figure 3.3. *Adressage par OID physique*

OID logiques : l'invariance des OID est assurée par une table d'indirection contenant les adresses de tous les articles d'un même fichier. Un OID correspond à un indice dans cette table, appelée TOBJ (table des objets) dans la figure 3.4. Si l'adresse de l'article change, l'entrée correspondante dans la TOBJ est mise à jour.



Une indirection de type c correspond à un positionnement dans la table TOBJ. Les indirections de types a et b sont identiques à la figure précédente. L'indirection de type b peut être évitée mais elle simplifie grandement le mécanisme de compactage de page.

Figure 3.4. *Adressage par OID logique*

Comparaison des deux méthodes : tant qu'un objet n'a pas changé de page, le décodage d'un OID physique, de coût *(a+b)*, est plus performant que le décodage d'un OID logique, coût *(c+a+b)*. Par contre, si l'objet a changé de page, le coût de décodage d'un OID physique devient *2(a+b)* alors que celui d'un OID logique reste invariant. La table TOBJ étant généralement chargée en mémoire primaire (du moins si sa taille le permet), l'indirection c ne génère aucune E/S disque et le mécanisme d'adressage par OID logiques devient meilleur.

Question 8

La gestion de pointeurs invariants inter-fichiers peut se faire par une extension directe des deux mécanismes de décodage d'OID étudiés dans la question 7. Il suffit désormais de considérer des OID globaux de la forme (n° de fichier, OID local), où OID local est un OID physique ou logique.

Question 9

Quand tous les articles sont de taille fixe, ils sont rangés de façon contiguë dans les pages du fichier. Chaque article est lui-même stocké comme une juxtaposition d'attributs. Il n'y a aucune information sur le type ni sur la taille des attributs à l'intérieur du fichier (donc aucune place perdue). Le type et taille des attributs est mémorisée dans une métabase (base de données système décrivant la base de données des utilisateurs). Pour accéder à l'attribut âge d'un article de type Personne, on consulte d'abord la métabase qui indique que cet attribut commence au 36ème octet de l'article et à une longueur de 4 octets.

Question 10

Trois modes de stockage sont possibles pour représenter des articles contenant des attributs de taille variable. Ces différents modes sont illustrés par la représentation d'un article de type Personne en considérant désormais que les attributs Personne.nom et Personne.prénom ont été déclarés comme des chaînes de caractère de taille variable.

Séparateur : chaque attribut de longueur variable est séparé de l'attribut suivant par un caractère spécial appelé séparateur. Lors de la lecture de l'article, le nombre de caractères lus entre deux séparateurs détermine la longueur de l'attribut considéré. L'inconvénient majeur de cette solution réside dans la difficulté de trouver un séparateur dont le codage binaire n'apparaisse jamais dans les données de l'utilisateur.

Exemple :

$	baudelaire	$	charles	$	30

Champ longueur : chaque attribut de taille variable est préfixé par sa longueur, codée sur un octet si la taille des attributs est limitée à 255 octets et codée sur 2 octets sinon. Cette représentation, bien que plus coûteuse en place pour des attributs de grande longueur, évite le problème du choix du séparateur. De plus, l'accès au i^{ème} attribut se fait de façon plus performante, en sautant de champ longueur en champ longueur.

Exemple :

10	baudelaire	7	charles	30

Références en tête : une référence adressant le début de chaque attribut est stockée dans l'entête de l'article. Cette référence est un déplacement en nombre d'octets, relatif au début de l'article. Cette technique est la plus optimale puisqu'elle offre un accès direct à chaque attribut de l'article.

Exemple :

1	11	18	baudelaire	charles	30

Question 11

Il suffit de découper l'article en composants dont la taille est inférieure ou égale à celle d'une page et de chaîner ces composants via des OIDs ou bien de stocker les OIDs de l'ensemble des composants dans l'entête de l'article.