

INFO-F201 - Système d'exploitation

Joël GOOSSENS

Résumé du cours

Rodrigue VAN BRANDE

18 janvier 2015

Table des matières

1	Résumé	3
2	Questions et réponses	3
2.1	Écrit	3
2.2	Oral	3
2.2.1	Processus et threads	3
2.2.2	Mémoire	6
2.2.3	Fichiers	7
2.2.4	E/S	10
2.2.5	Linux	11

1 Résumé

2 Questions et réponses

2.1 Écrit

2.2 Oral

2.2.1 Processus et threads

2.2.1.1 Question 1 (Processus et threads). Quel est le principal intérêt d'implémenter les threads dans l'espace utilisateur ? Et quel en est l'inconvénient majeur ?

- L'avantage est l'efficacité. La gestion en espace utilisateur est que le programmeur gère ses threads, et peut faire des choses sur mesure pour ses problèmes. Quand l'ordonnanceur est dans le noyau, on y a beaucoup moins accès. Le changement de threads est également bien plus léger.
- Un inconvénient de l'espace utilisateur est que l'OS ne voit pas les threads. Si un des threads est bloqué pour une raison ou une autre, c'est tout le processus qui va être bloqué.

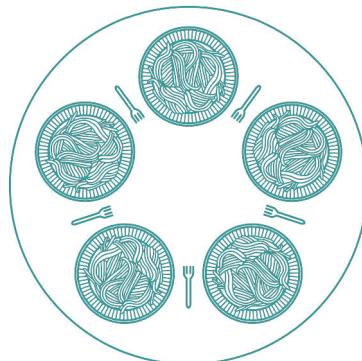
2.2.1.2 Question 2 (Processus et threads).

- Définir le problème d'exclusion mutuelle ;
- La solution de Peterson au problème d'exclusion mutuelle (Figure 1) fonctionne-t-elle lorsque l'ordonnancement est préemptif ? Non préemptif ?
- Elle fonctionne avec l'ordonnancement préemptif. En fait, elle a été conçue pour cela. Lorsque l'ordonnancement n'est pas préemptif, il risque d'échouer. Prenons le cas où turn est initialement à 0 mais que le processus 1 s'exécute en premier. Il va boucler indéfiniment et ne libérera jamais le processeur.

2.2.1.3 Question 3 (Processus et threads). Décrire le problème des philosophes. La Figure 2 présente-t-elle une solution prétendue au problème des philosophes ? Quel est le souci avec cet algorithme, justifiez.

- En 1965, Dijkstra a posé et résolu un problème de synchronisation qu'il a appelé le problème du dîner des philosophes. Depuis lors, toutes les personnes ayant mis au point une nouvelle primitive de synchronisation se sont senties obligées de démontrer son grand intérêt en illustrant à quel point elle résolvait élégamment le problème.

Quel est-il ? Cinq philosophes sont assis autour d'une table ronde. Chacun a devant lui une assiette de spaghetti. Mais ceux-ci sont si glissants qu'il faut deux fourchettes pour les manger. Entre deux assiettes se trouve une fourchette.



Quand un philosophe a faim, il tente de s'emparer des fourchettes droite et gauche sans ordre défini. S'il y parvient, il mange pendant un moment, puis pose les fourchettes et se met à penser. La question est la suivante : pouvez-vous écrire un programme pour que chaque philosophe puisse exercer ces deux activités sans jamais se retrouver bloqué ?

- Ici le code n'est pas une solution au problème. Car il faut gérer le problème avec une section critique.

2.2.1.4 Question 4 (Processus et threads). Expliquez le fonctionnement de l'appel système *fork* de la norme POSIX. Qu'affiche à l'écran le pseudo-programme suivant :

```
cout << "a" ; fork(); cout << "b"; fork(); cout << "c";
```

```

#define FALSE 0
#define TRUE 1
#define N      2 /* number of processes */

int turn; /*whose turn is it?*/
int interested[N]; /*all values initially 0 (FALSE)*/
void enter_region(int process) /*process is 0 or 1*/
{
    int other; /*number of the other process*/

    other = 1 - process; /*the opposite of process*/
    interested[process] = TRUE; /*show that you are interested*/
    turn = process; /*set flag*/
    while (turn == process && interested[other] == TRUE); /*null statement */
}

void leave_region(int process) /*process: who is leaving*/
{
    interested[process] = FALSE; /*indicate departure from critical region*/
}

```

FIGURE 1 – Solution de Peterson

```

#define N      5 /* number of processes */

void philosopher(int i) /*I: philosopher number, from 0 to 4*/
{
    while (TRUE){
        think(); /*philosopher is thinking*/
        take_fork(i); /*take left fork*/
        take_fork((i+1) % N); /*take right fork; % is modulo operator*/
        eat(); /*yum-yum, spaghetti*/
        put_fork(i); /*put left fork back on the table*/
        put_fork((i+1) % N); /*put right fork back on the table*/
    }
}

```

FIGURE 2 – Tentative de solution au problème des philosophes

- Ce code affiche "abbccc". Le programme démarre, affiche "a" puis se sépare en deux processus qui vont chacun afficher "b" et se séparer eux aussi en 2 processus qui vont afficher "c", soit un total de 4 processus au final. La partie "bcc" peut être permutée en "cbc" ou "ccb" selon l'ordre de scheduling des processus.

2.2.1.5 Question 5 (Processus et threads). Considérez un ordinateur doté de deux processeurs, chacun disposant de deux threads (hyperthreading). Supposez que trois programmes, P_0 , P_1 et P_2 , sont lancés avec des temps d'exécution de 5, 10 et 20 ms. Combien de temps mettra cet ordinateur pour exécuter globalement ces trois programmes ? Considérez que ces programmes sont 100% en mémoire, qu'il n'y a pas de blocage pendant l'exécution et qu'il n'y a pas de changement de processeur une fois qu'il est assigné.

- L'ordinateur peut mettre 20, 25 ou 30 ms pour exécuter globalement les programmes selon l'organisation de leur exécution par le système d'exploitation.

1. Si P_0 et P_1 sont sur le même processeur et que P_2 est lancé sur l'autre, il mettra 20 ms.
2. Si P_0 et P_2 sont sur le même processeur et que P_1 est lancé sur l'autre, il mettra 25 ms.
3. Si P_1 et P_2 sont sur le même processeur et que P_0 est lancé sur l'autre, il mettra 30 ms.
4. Si les trois sont lancés sur le même processeur, cela mettra 35 ms.

2.2.1.6 Question 6 (Processus et threads). Pour chacun des appels système fork, exec et unlink, donnez une condition d'échec.

- fork peut échouer s'il n'y a pas de connecteur libre dans la table de processus et éventuellement, s'il ne reste plus de mémoire ou d'espace d'échange ;
- exec peut échouer si le nom du fichier donné n'existe pas ou s'il ne s'agit pas d'un fichier exécutable valide ;
- unlink peut échouer si le fichier à détruire n'existe pas ou si le processus appelant n'a pas l'autorité pour le faire.

2.2.1.7 Question 7 (Processus et threads). Pourquoi une table de processus est-elle nécessaire dans un système à temps partagé ? Est-elle également requise dans un système personnel où un seul processus existe, avec accès à toute la machine durant son exécution ?

- La table de processus sert à stocker l'état d'un processus suspendu, qu'il soit prêt ou bloqué. Elle n'est pas nécessaire dans un système à processus unique, puisque celui-ci n'est jamais suspendu.

2.2.1.8 Question 8 (Processus et threads). Sur tous ordinateurs actuels, au moins une partie des gestionnaires d'interruption est écrite en langage d'assemblage. Pourquoi ?

- Généralement, les langages évolués n'autorisent pas le type d'accès requis au matériel du processeur. Par exemple, il peut être nécessaire de disposer d'un handler d'interruption pour activer et désactiver l'interruption servant un périphérique particulier. En outre, les routines de service d'interruption doivent s'exécuter aussi rapidement que possible.

2.2.1.9 Question 9 (Processus et threads). Dans un système comprenant des threads, trouve-t-on une pile par thread ou une pile par processus lorsqu'il s'agit de threads utilisateur ? Que se passe-t-il lorsque l'on utilise des threads noyau ? Expliquez.

- Chaque thread appelle ses propres procédures et doit donc disposer de sa propre pile pour les variables locales, les adresses de retour, etc. Cela est vrai pour les threads utilisateur et les threads noyau.

2.2.1.10 Question 10 (Processus et threads). Qu'est-ce qu'une section critique ? Quelles sont les règles à respecter ?

- Les sections critiques sont la solution pour résoudre les problèmes de concurrence (Accès à une même donnée). Une section critique est une partie de code dans laquelle un seul processus à la fois peut rentrer. Elle n'est libérée que quand le traitement est complètement terminé.

Pour avoir une section critique, il faut respecter un certain nombre de règles, pour éviter les bugs :

1. On ne peut pas avoir plusieurs processus simultanément dans la région critique.
2. On ne peut pas faire de suppositions sur les fréquences du processeur et sa vitesse.

3. Aucun processus hors de sa section critique ne peut bloquer un autre processus. Quand on n'est pas intéressé, on laisse tranquille.
4. Tout cela doit être équitable. Tout processus qui un jour ou l'autre veut entrer dans la section critique doit un jour rentrer dedans.

Quand un processeur entre dans la section critique, il peut d'abord avoir à attendre. Quand un processus sort de la section critique, un autre processus attendant la section critique est réveillé. Ce sont les opérations "lock" et "unlock".

2.2.1.11 Question 11 (Processus et threads). À la Figure 3, le jeu de registres est classé par éléments de thread, et non par éléments de processus. Pourquoi ? Après tout, l'ordinateur ne possède qu'un seul jeu de registres.

—

2.2.2 Mémoire

2.2.2.1 Question 1 (Mémoire). Pour chacune des adresses virtuelles décimales suivantes, donnez le numéro de page virtuelle et le déplacement pour des pages de 4 Ko et de 8 Ko : 20.000, 32.768 et 60.000.

- $Ko_{Reel} = Ko * 2^{10} = Ko * 1.024$
- $Deplacement = Adresse - (Ko_{Reel} * Page) - Page$
- Une page de 4 Ko contient des adresses de 0 à 4.095.
- $f(x) = x * 4.095$
- $f(4 ; 5 ; 8 ; 9 ; 14 ; 15) = 16.380 ; 20.475 ; 32.760 ; 36.855 ; 57.330 ; 61.425$
 - Donc pour 20.000 on est à la page 4.
 - $Deplacement = 3.616$
 - Donc pour 32.768 on est à la page 8.
 - $Deplacement = 0$
 - Donc pour 60.000 on est à la page 14.
 - $Deplacement = 2.656$
- Une page de 8 Ko contient des adresses de 0 à 8.192.
- $f(x) = x * 8.192$
- $f(2 ; 3 ; 4 ; 7 ; 8) = 16.384 ; 24.576 ; 32.768 ; 57.344 ; 65.536$
 - Donc pour 20.000 on est à la page 2.
 - $Deplacement = 3.614$
 - Donc pour 32.768 on est à la page 3.
 - $Deplacement = 8.189$
 - Donc pour 60.000 on est à la page 7.
 - $Deplacement = 2.649$

Remarques : Le $-Page$ dans la formule est là pour corriger les offset du fait qu'on commence à compter depuis 0.

2.2.2.2 Question 2 (Mémoire). Si l'algorithme FIFO est utilisé avec 4 cases mémoire et 8 pages, combien de défaut de pages se produiront avec la suite de référence 0 1 7 2 3 2 7 1 0 3 si les 4 cases sont initialement vides ? Refaites le calcul pour LRU.

- Les cases pour FIFO sont les suivantes :

Départ : xxxx

0 :	0xxx
1 :	10xx
7 :	710x
2 :	2710
3 :	3271
2 :	3271
7 :	3271
1 :	3271
0 :	0327
3 :	0327

- Les cases pour LRU sont les suivantes :

Départ : xxxx
 0 : 0xxx
 1 : 10xx
 7 : 710x
 2 : 2710
 3 : 3271
 2 : 2371
 7 : 7231
 1 : 1723
 0 : 0172
 3 : 3017

FIFO produit 6 défauts de page et LRU en produit 7.

2.2.2.3 Question 3 (Mémoire). Définir les notions de fragmentation externe et interne dans la gestion de la mémoire. Quelles sont les différences ?

- La fragmentation externe est causée par l'allocation dynamique. Le problème est d'allouer des segments (c-à-d plusieurs blocs contigus) en minimisant le nombre de blocs (ou pages) inutilisés entre ces segments (c'est un peu comme Tetris) ;
- La fragmentation interne est causée par la taille fixe des blocs, et a lieu lorsqu'un bloc n'est pas entièrement utilisé.

2.2.2.4 Question 4 (Mémoire). Considérons un système de va-et-vient dans lequel la mémoire est constituée par une succession de zones vides dans l'ordre suivant : 10 Ko, 4 Ko, 20 Ko, 18 Ko, 7 Ko, 9 Ko, 12 Ko et 15 Ko. Quelle zone sera prise pour les requêtes de segments successives suivantes :

1. 12 Ko ;
2. 10 Ko ;
3. 9 Ko.

Pour la première zone libre (*first fit*) ? Répondez également) cette question pour le meilleur ajustement (*best fit*), le plus grand résidu (*worst fit*) et la zone suivante (*next fit*).

- La première section libre (*first fit*) prend 20 Ko, 10 Ko et 18 Ko.
- Le meilleur ajustement (*best fit*) prend 12 Ko, 10 Ko et 9 Ko.
- Le plus grand résidu (*worst fit*) prend 20 Ko, 18 Ko et 15 Ko.
- La section suivante (*next fit*) prend 20 Ko, 18 Ko et 9 Ko.

2.2.2.5 Question 5 (Mémoire). Considérons la séquence de pages de la figure 4 (b). Supposons que les bits *R* des pages *B* jusqu'à *A* sont respectivement 1 1 0 1 1 0 1 1. Quelles pages seront effacées avec l'algorithme de la deuxième chance ?

- La première page contenant un bit 0 sera choisie, dans ce cas *D*.

2.2.2.6 Question 6 (Mémoire). Une page peut-elle se trouver dans deux ensembles de travail en même temps ? Expliquez.

- Si l'on peut partager les pages, oui. Par exemple, si deux utilisateurs d'un système en temps partagé exécutent le même éditeur au même moment et que le texte du programme est partagé au lieu d'être copié, certaines de ces pages peuvent se trouver simultanément dans l'espace de travail de chaque utilisateur.

2.2.3 Fichiers

2.2.3.1 Question 1 (Fichiers). Comme nous l'avons vu, l'allocation contiguë de fichiers conduit à une fragmentation du disque. Cette fragmentation est-elle interne ou externe ? Faites une analogie avec un point du chapitre "gestion de la mémoire".

- La fragmentation externe est causée par l'allocation dynamique. Le problème est d'allouer des segments (c-à-d plusieurs blocs contigus) en minimisant le nombre de blocs (ou pages) inutilisés entre ces segments (c'est un peu comme Tetris) ;
- La fragmentation interne est causée par la taille fixe des blocs, et a lieu lorsqu'un bloc n'est pas entièrement utilisé.

2.2.3.2 Question 2 (Fichiers). Certains systèmes d'exploitation proposent un appel système **rename** pour donner un nouveau nom à un fichier. Existe-t-il une différence entre cette opération et celle qui consiste à faire une copie du fichier dans un nouveau fichier avec un nouveau nom et ensuite effacer le premier fichier ?

- Oui. L'appel **rename** ne change pas l'heure de création ou l'heure de la dernière modification. En revanche, un nouveau fichier se voit attribuer l'heure actuelle comme heure de création et heure de la dernière modification. En outre, si le disque est plein, la copie peut échouer.

2.2.3.3 Question 3 (Fichiers). Considérons l'i-node de la Figure 5. S'il contient 10 adresses directes de 4 octets chacune et si tous les blocs sont de 1.024 octets, quelle est la taille maximale d'un fichier ?

- Le bloc indirect peut contenir 256 adresses disque. Avec les 10 adresses disque directes, le fichier contient au maximum 266 blocs. Dans la mesure où chaque bloc a une taille de 1 Ko, la taille maximale d'un fichier est de 266 Ko.

2.2.3.4 Question 4 (Fichiers). On peut garder une trace de l'espace disque disponible à l'aide d'une liste de blocs libres ou d'une table de blocs libre (*bitmap*). Les adresses disque nécessitent D bits. Dans le cas d'un disque comprenant B , dont F sont libres, donnez la condition dans laquelle la liste de blocs libres utilise moins de place que la table de blocs libres. Pour D égal à 16 octets, exprimez votre réponse en pourcentage de l'espace disque qui doit être libre. Faites une analogie avec un point du chapitre "gestion de la mémoire".

- La table de blocs demande B bits et la liste des blocs libres DF bits. La liste de blocs libres nécessite moins de bits si $DF < B$ ou si F/B est une fraction des blocs libres. Pour des adresses disque 16 bits, la liste des bits libres est plus courte si 6% ou moins du disque sont libres.

2.2.3.5 Question 5 (Fichiers). Est-il raisonnable de défragmenter le disque ?

- S'il est correctement effectué, oui. Pendant le compactage, chaque fichier doit être organisé pour que tous ses blocs soient consécutifs, afin d'accélérer l'accès. Les utilisateurs sont encouragés à l'exécuter périodiquement pour améliorer les performances du système. Compte tenu du temps nécessaire à cette tâche, une exécution par mois représente une bonne fréquence.

2.2.3.6 Question 6 (Fichiers). Le début d'une table de blocs libres juste après le formatage d'une partition disque ressemble à 1000.0000.0000.0000 (le premier bloc est utilisé par le répertoire racine). Le système recherche toujours les blocs libres à partir du bloc qui a le plus petit nombre; ainsi après l'écriture du fichier *A*, qui requiert 6 blocs, la table des blocs libres est de la forme : 1111.1110.0000.0000. Donnez la table après chacune des opérations suivantes :

1. Le fichier *B* est écrit en utilisant 5 blocs.
2. Le fichier *A* est effacé.
3. Le fichier *C* est écrit en utilisant 8 blocs.
4. Le fichier *B* est effacé.

– Le début de la table de blocs ressemble à :

1. 1111.1111.1111.0000
2. 1000.0001.1111.0000
3. 1111.1111.1111.1100
4. 1111.1110.0000.1100

2.2.3.7 Question 7 (Fichiers). Donnez un avantage des liens matériels par rapport aux liens symboliques et un avantage des liens symboliques vis-à-vis des liens matériels.

- Les liens matériels ne demandent pas d'espace disque supplémentaire, mais simplement un compteur dans leur i-node pour suivre leur nombre, alors que les liens symboliques demandent de l'espace pour stocker le nom du fichier pointé. Les liens symboliques peuvent pointer vers des fichiers situés sur d'autres ordinateurs, voire sur internet, alors que les liens matériels sont limités aux fichiers de leur propre partition.

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

FIGURE 3 – Processus et thread

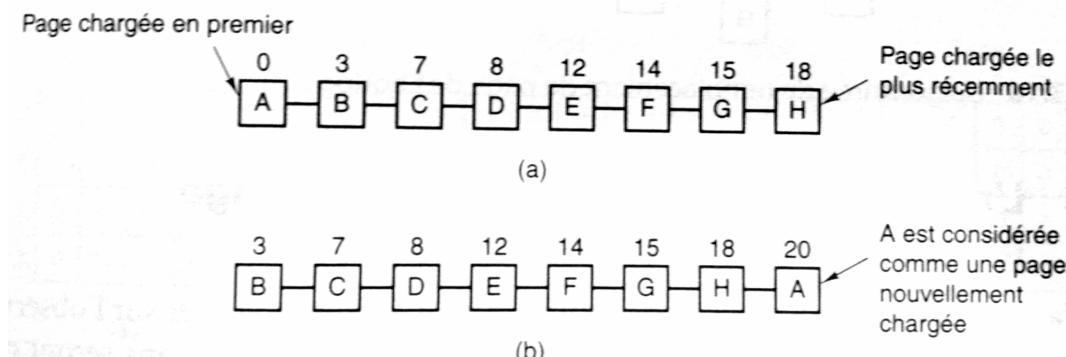


FIGURE 4 – Séquence de pages

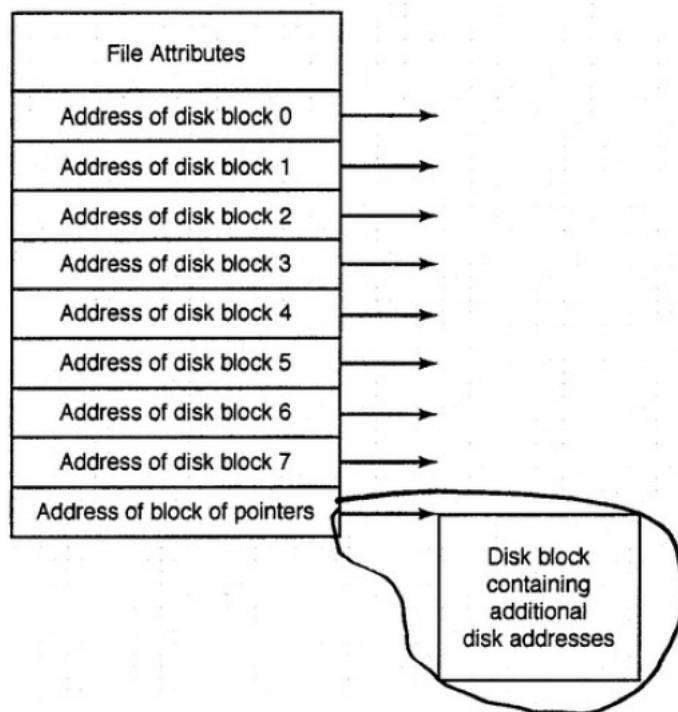


FIGURE 5 – i-node

2.2.3.8 Question 8 (Fichiers). Nous avons étudié en détail les sauvegardes incrémentales. Sous Windows, il est facile de savoir quand sauvegarder un fichier, car chaque fichier possède un bit d'archivage. Mais ce bit est absent sous UNIX. De quelle manière les programmes de sauvegarde d'UNIX ont-ils connaissance des fichiers à sauvegarder ?

- Ils doivent conserver l'heure de la dernière sauvegarde dans un fichier sur le disque. À chaque sauvegarde, on ajoute une entrée à ce fichier. Au moment de la sauvegarde, on lit le fichier et l'heure de la dernière entrée notée. Tout fichier modifié depuis cette heure est sauvegardé.

2.2.4 E/S

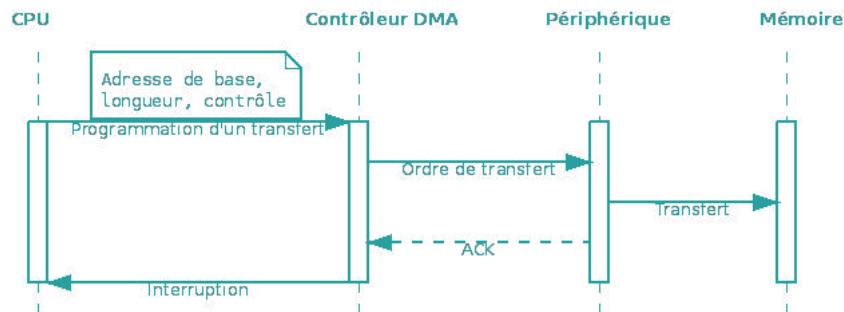
2.2.4.1 Question 1 (E/S). Énoncer les propriétés importantes des "interruptions précises" et de leur intérêt pour les concepteurs des systèmes d'exploitation.

- Ce sont des interruptions qui garantissent que l'état des registres est cohérent après son exécution par 4 propriétés :

1. Le PC (program counter) est sauvé dans un endroit connu.
2. Toutes les instructions précédant l'instruction pointée par PC ont été totalement exécutées.
3. Aucune instruction après n'a été exécutée.
4. L'état de l'instruction pointée par PC est connu.

Ces conditions peuvent paraître évidentes, mais on doit y faire attention, particulièrement dans le cas de pipelines où des instructions pourraient être partiellement exécutées, ou dans le cas d'instructions d'entrées sorties dont le principe est basé sur des effets de bords. Certains processeurs comme les x86 actuels réordonnent aussi les instructions pendant l'exécution, il faut donc en tenir compte. (Infos en plus : <http://1wn.net/images/pdf/LDD3/ch09.pdf>)

2.2.4.2 Question 2 (E/S). À l'aide d'un schéma expliquer l'accès direct à la mémoire (DMA, *Direct Memory Access*). En particulier décrivez les 4 étapes vues au cours.



1. Le CPU programme le contrôleur DMA : il initialise un registre avec l'adresse de base, un autre avec la longueur du transfert et des registres de contrôle (direction du transfert (read/write) par exemple).
2. Le contrôleur DMA envoie le message au périphérique (pendant ce temps le CPU fait autre chose).
3. Le périphérique effectue le transfert vers ou depuis la mémoire et le signale au contrôleur DMA quand il a fini.
4. Le contrôleur DMA déclenche une interruption pour signaler que la transaction est finie.

2.2.4.3 Question 3 (E/S). Pourquoi les fichiers de sortie de l'imprimante sont-ils normalement "spoulés" sur disque avant d'être imprimés ?

- Le service spouleur d'impression permet de charger en mémoire les travaux d'impression pour une impression ultérieure, c'est-à-dire comme une imprimante ne peut imprimer qu'un fichier à la fois on va le mettre dans la file d'attente afin de ne pas bloquer l'accès au fichier lors de l'impression. Quand l'imprimante a fini avec un fichier, c'est le fichier suivant dans le spouleur qui est envoyé automatiquement à l'imprimante.

Si des fichiers restent bloqués dans le spouleur c'est que l'imprimante à un problème (plus papier, plus d'encre,...) ou que l'impression du fichier a été suspendu. Sur un ordinateur personnel, on exploite rarement le spouleur d'entrée mais en revanche, on utilise le spouleur de sortie.

2.2.4.4 Question 4 (E/S). Supposez qu'un ordinateur puisse lire ou écrire un mot mémoire en 10 ns. Supposez également que, lorsqu'une interruption se produit, les 32 registres du processeur, plus le compteur ordinal et le mot d'état du programme, soient placés sur la pile. Quel est le nombre maximal d'interruptions par seconde que cet ordinateur peut traiter ?

- Il faut placer 34 mots sur la pile pour chaque interruption. Le retour de l'interruption nécessite l'extraction de 34 mots de la pile. Cette seule surcharge représente 680 ns. Ainsi, le nombre maximum d'interruptions par seconde est de 1,47 million, sans effectuer le moindre travail pour les interruptions.

2.2.4.5 Question 5 (E/S). De nombreuses versions d'UNIX utilisent un entier 32 bits non signé pour suivre l'heure sous la forme d'un nombre de secondes écoulées depuis l'origine du temps. Quand ces systèmes vont-ils boucler (année et mois) ? Pensez-vous que cela se produira réellement ?

- Le nombre de secondes d'une année moyenne est de $365,25 * 24 * 3600 = 31.557.600$. Le compteur boucle après 2^{32} secondes à partir du 1^{er} janvier 1970. La valeur de $2^{32}/31.557.600$ donne 136,1 ans. La boucle se produira donc à 2.106,1 soit début février 2106. D'ici là, tous les ordinateurs utiliseront au moins 64 bits, donc cela ne se produira pas.

2.2.4.6 Question 6 (E/S). Une page de texte imprimée classique contient 50 lignes et 80 caractères chacune. Imaginez qu'une imprimante donnée puisse imprimer 6 pages/minutes et que le temps d'écrire un caractère dans le registre de sortie de l'imprimante soit si court qu'il puisse être ignorée. Est-il logique d'exécuter cette impression avec des E/S pilotées par les interruptions si chaque caractère imprimé demande une interruption qui prend 50 s en tout ?

—

2.2.5 Linux

2.2.5.1 Question 1 (Linux). Un enseignant partage des fichiers avec ses étudiants en les plaçant dans un répertoire Linux de son département accessible publiquement. Un jour, il réalise qu'un fichier placé la veille est en écriture pour tous. Il modifie les permissions et vérifie que le fichier est identique à la version d'origine. Le lendemain, il découvre que le fichier a été modifié. Comment est-ce possible et comment aurait-on pu l'empêcher ?

- Un étudiant a pu garder le fichier ouvert avant et pendant que le prof modifiait les permissions, puis il a sauvegardé le fichier. Le professeur aurait dû supprimer le fichier puis placer une copie du fichier maître dans le répertoire public.

2.2.5.2 Question 2 (Linux). Est-il possible qu'avec l'algorithme buddy pour l'allocation de mémoire physique en Linux que deux blocs de mémoire adjacents, libres et de même taille, coexistent sans être réunis en un bloc de taille double ? Si oui, expliquer comment. Sinon, prouvez-le.

- C'est possible si les deux blocs ne sont pas des buddies. Prenons l'exemple de la figure 6 (e). Deux nouvelles requêtes entrent pour 8 pages chacune. À ce point, les 32 pages du bas sont la propriété de quatre utilisateurs différents, possédant 8 pages chacun. Les utilisateurs 1 et 2 libèrent leurs pages, mais les utilisateurs 0 et 3 conservent les leurs. Cela produit une situation dans laquelle 8 pages sont utilisées, 8 pages sont libres, 8 pages sont libres et 8 pages sont utilisées. Ces deux blocs adjacents de taille égale ne peuvent pas fusionner parce qu'il ne sont pas des buddies.

2.2.5.3 Question 3 (Linux). Quand un nouveau processus est créé un nombre entier lui est attribué pour PID. Est-ce suffisant d'avoir un compteur dans le noyau qui s'incrémente à chaque création de processus et qu'on utilise pour distribuer les PID ? Justifiez votre réponse.

- Non ce n'est pas suffisant, tout PID doit être unique. Tôt ou tard, le compteur boucle et revient à 0. Il monte ensuite à 15, par exemple. S'il se trouve que le processus 15 a été démarré il y a plusieurs mois mais qu'il s'exécute toujours, on ne peut pas assigner 15 à un nouveau processus. Ainsi, une fois que le PID a été choisi par le biais du compteur, on doit parcourir la table des processus pour vérifier si le PID est toujours exploité.

2.2.5.4 Question 4 (Linux). Quel est le fonctionnement de l'ordonnanceur de processus et des threads sous Linux, quel est le principe des 280 têtes de listes *active* / *expired* ? Comment évoluent les priorités ?

—

2.2.5.5 Question 5 (Linux). Libérer la mémoire d'un processus quand ce dernier passe dans l'état zombie a-t-il un sens ? Justifiez votre réponse.

- Oui. Il ne peut plus s'exécuter, donc plus sa mémoire retourne rapidement sur la liste libre, mieux c'est.

2.2.5.6 Question 6 (Linux). En général, pensez-vous que les *daemons* ont une priorité supérieure ou inférieure à celle des processus interactifs ? Pourquoi ?

- En général, les *daemons* s'exécutent à l'arrière-plan pour effectuer des tâches telles que l'impression et l'envoi de courrier électronique. On leur attribue une priorité faible et ils profitent du surplus de temps UC que les processus interactifs n'emploient pas.

2.2.5.7 Question 7 (Linux). Dans chaque entrée de la structure de tâche, on trouve le PID du processus parent. Pourquoi ?

- Quand le processus quitte, le parent reçoit l'état de sortie de son enfant. On a besoin du PID pour identifier le parent afin que l'état puisse être transféré au processus correct.

2.2.5.8 Question 8 (Linux). Quand on lit un i-node sur disque lors de l'ouverture d'un fichier, il est placé en mémoire dans la table des i-nodes. Cette table possède certains champs qui ne figurent pas sur disque. L'un d'eux est un compteur mémorisant le nombre de fois que l'i-node a été ouvert. Pourquoi ce champ est-il nécessaire ?

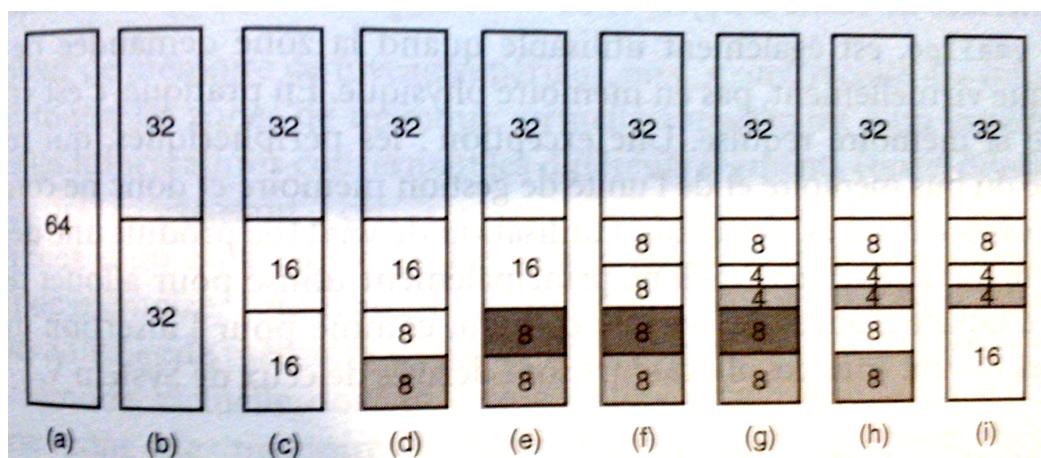
- Lorsque l'on ferme un fichier, le compteur de son i-node en mémoire est décrémenté. S'il est supérieur à zéro, on ne peut pas supprimer l'i-node de la table, puisque le fichier est toujours ouvert dans un processus. C'est uniquement quand le compteur atteint zéro que l'on peut supprimer l'i-node. Sans le compte des références, le système ne pourrait pas savoir quand supprimer l'i-node de la table. Créer une copie séparée de l'i-node chaque fois que l'on ouvre le fichier ne fonctionne pas, dans la mesure où les modifications apportées à une copie ne seraient pas visibles sur les autres.

2.2.5.9 Question 9 (Linux). Est-il possible de faire un *unlink* d'un fichier qui n'a jamais été lié ? Que se passe-t-il ?

- Le fichier est simplement supprimé. C'est la méthode classique (la seule) pour supprimer un fichier.

2.2.5.10 Question 10 (Linux). Donnez le fonctionnement du système de fichier ext2 du noyau linux, exprimez la taille maximale d'un fichier en fonction de la taille des blocs (b un nombre d'octets) et pour des adresses de 32 bits.

—

FIGURE 6 – Déroulement du *buddy algorithm*.