UPMC SORBONNE UNIVERSITÉS

UFR 919

19 janvier 2016

Licence d'Informatique 2015 -2016

L3 – 3I010

Module « Principes des Systèmes d'Exploitation »

TD

Séances 1 à 11

---

1 - MULTI-PROGRAMMATION

1. INTRODUCTION

Un *programme* est l'expression d'un algorithme à l'aide d'un langage de programmation.

Une *tâche* est la conjonction d'un programme et des données auxquelles il s'applique.

Un *processus* est l'exécution d'une tâche. Il est caractérisé par la tâche et son contexte d'exécution, c'est-à-dire la valeur du compteur ordinal, les valeurs des registres, etc.

En *monoprogrammation*, il y a un seul processus à la fois en mémoire. Lorsqu'une tâche est soumise et que le processeur est disponible, on la charge en mémoire puis on exécute le processus associé jusqu'à ce qu'il soit terminé. On passe alors à la tâche suivante.

En *multiprogrammation*, il peut y avoir plusieurs processus à la fois en mémoire. Une tâche soumise est chargée en mémoire s'il y a de la place et donne naissance à un processus. Un processus est prêt s'il n'est pas en attente d'un événement extérieur (fin d'entrée-sortie, libération de ressource, etc.). Les processus prêts s'exécutent à tour de rôle, on parle de commutation de processus. La multiprogrammation peut être utilisée en batch ou en temps partagé.

En *batch*, il n'y a commutation que si le processus actif doit effectuer une entrée-sortie.

En *temps partagé*, il y a commutation si le processus actif est en attente d'un événement ou s'il a épuisé son quantum.

Un *quantum* est une durée élémentaire (de l'ordre de 10 à 100 ms).

L'*overhead* est le temps passé (perdu) à effectuer une commutation entre deux processus en temps partagé.

Une *unité d'échange* sert à effectuer des transferts entre la mémoire principale et un périphérique sans utiliser le processeur (DMA). Elle reçoit un ordre du processeur, effectue le transfert, puis avise le processeur de la fin de l'échange. Durant le transfert, le processeur peut faire d'autres calculs.

On se propose d'étudier analytiquement l'influence des différentes politiques de gestion d'un processeur. On considère un ordinateur doté d'un processeur principal et d'une unité d'échange effectuant les entrées-sorties. On suppose qu'il y a suffisamment de mémoire pour contenir tous les processus.

1.1.

Représentez sous forme de diagramme d'états (noeuds = états d'une tâche, arcs = actions ou événements) l'exécution d'une tâche en mode batch puis en temps partagé.

2. ETUDE DU TAUX D'OCCUPATION

On s'intéresse à l'exécution de trois tâches T_1, T_2 et T3 :

T1 dure 200 ms (hors entrée/sortie) et réalise une unique E/S au bout de 110 ms ;

T2 dure 50 ms et réalise une unique entrée/sortie au bout de 5 ms ;

T3 dure 120 ms sans faire d'entrée/sortie

T1 et T2 sont créées à l'instant 0, T3 est créée à l'instant 140. Chaque entrée/sortie dure 10 ms. La valeur du quantum (pour le temps partagé uniquement) est de 100 ms.

2.1.

Représentez sur un diagramme de Gantt l'exécution des 3 tâches en mode batch et en temps partagé. Calculez le temps total d'exécution T et le taux d'occupation du processeur τ_P.

On considère maintenant deux tâches identiques T_1 et T_2, soumises simultanément à l'instant t = 0 et effectuant n fois le traitement suivant :

- Lecture d'une donnée sur le disque (durée l)
- Opération de calcul sur la donnée (durée c)
- Ecriture du résultat sur le disque (durée e)

2.2.

La tâche T_1 est soumise seule à t = 0. Représentez sur un diagramme de Gantt l'exécution de la tâche T_1. Calculez le temps total d'exécution T, le taux d'occupation du processeur τ_P et de l'unité d'échange τ_U.

Que peut-on en déduire pour T, τ_P et τ_U lorsque les tâches T_1 et T_2 sont exécutées en monoprogrammation ? Application numérique : e = 5 ms, c = 7 ms, l = 8 ms, n = 4.

2.3.

Représentez sur un diagramme de Gantt l'exécution des tâches T_1 et T_2 en supposant maintenant une multiprogrammation en mode batch. On supposera c < l et c > e. Calculez T, τ_P et τ_U avec les mêmes valeurs numériques qu'à la question précédente et comparez les résultats.

2.4.

Y a-t-il un intérêt à la multiprogrammation si l'ordinateur ne dispose pas d'une unité d'échange?

3. TEMPS DE REPONSE EN MODE BATCH

On suppose qu'il y a n utilisateurs exécutant des commandes d'édition. Chaque commande consomme c secondes de temps processeur. Chaque utilisateur attend r secondes (temps de réponse) entre l'instant où il soumet une commande et l'instant où il obtient la réponse, il s'écoule ensuite t secondes (réflexion et frappe) avant qu'il soumette la commande suivante.

3.1.

Tous les utilisateurs commencent à travailler en même temps. Exprimez le temps de réponse en fonction des autres paramètres en régime permanent. Calculez r pour n = 10, t = 2 ms et c = 0,1 ms ; puis pour n = 30, t = 2 ms et c = 0,1 ms.

On suppose qu'il y a une autre classe d'utilisateurs effectuant des compilations. On note N, C, R, T les paramètres correspondant à cette classe. On suppose t = T, t <= (n-1)c+NC et T <= nc+(N-1)C.

3.2.

Tous les utilisateurs commencent à travailler en même temps. Exprimez les temps de réponse r et R en fonction des autres paramètres en régime permanent. Montrez que r = R. Calculez r pour n = 10, t = 2 ms, c = 0,1 ms, N = 5 et C = 3 ms.

3.3.

Est-il raisonnable d'envisager un travail interactif en mode batch?

4. TEMPS DE REPONSE EN TEMPS PARTAGE

Dans cette partie, on note q le quantum et on suppose qu'il n'y a pas d'overhead. On considère à nouveau n utilisateurs effectuant des commandes d'édition et N utilisateurs effectuant des compilations. Pour simplifier l'analyse, on suppose que t = T = 0 et que c et C sont multiples de q.

4.1.

Tous les utilisateurs commencent à travailler en même temps. Exprimez les temps de réponse r et R en fonction des autres paramètres en régime permanent. Calculez r et R pour n = 10, c = 0,1 ms, N = 5 et C = 3 ms.

4.2.

Conclusion ?

2 - INTERRUPTION HORLOGE

1. CHOIX DU QUANTUM

On considère un système en temps partagé et on suppose, pour simplifier, qu'à chaque passage sur le processeur une tâche utilise la totalité de son quantum de temps. On suppose par contre qu'il y a un overhead de s millisecondes à chaque commutation.

1.1.

Quel est le surcoût en temps lié aux commutations ?

1.2.

On prend un quantum de 100 ms et un overhead de 1 ms. On affecte le temps de commutation en fin de tâche à la tâche qui est retirée du processeur.

Quelle est la durée totale (temps CPU consommé, en incluant les commutations) d'une tâche demandant 20 ms de calcul ? D'une tâche demandant 500 ms de calcul ?

Quel est le pourcentage de temps passé par chacune de ces tâches en commutations ?

1.3.

Quels sont les critères de choix du quantum?

2. GESTION D'UNE INTERRUPTION HORLOGE

On considère le processeur JB007 qui dispose d'un registre temporisateur Rtempo, décrémenté automatiquement toutes les micro-secondes *en mode usager* et *en mode système.* Lorsque sa valeur atteint 0, Rtempo provoque une interruption (interruption horloge). Ce registre est initialisé automatiquement à une valeur RTMAX : celle où tous les bits sont à 1, sauf le bit de poids fort qui est à 0. La représentation utilisée est une représentation en complément à 2.

L'interruption horloge est utilisée à la fois pour tenir à jour l'heure universelle dans un mot RHU, pour instaurer un tour de rôle avec quantum d'exécution et pour comptabiliser le temps CT utilisé par une tâche. Lorsque cette variable CT dépasse une valeur CTMAX, la tâche est détruite.

2.1.

Quelle doit être la taille minimale de Rtempo si l'intervalle entre deux interruptions horloge est de 10 ms ?

2.2.

Pourquoi décrémente-t-on Rtempo même en mode système, et pourquoi continue-t-on à le décrémenter lorsqu'il atteint 0 ?

Le système gère une table TDT des descripteurs de tâches : TDT[i] contient l'ensemble des informations relatives à la tâche i. On considère pour l'instant que la valeur QX du quantum est telle que QX = RTMAX.

2.3.

Quelles sont les opérations concernant la gestion du temps que doit effectuer le système la première fois qu'il charge la tâche i pour l'exécuter ?

Le système dispose d'une variable ITE qui contient l'indice de la tâche élue. On suppose qu'une tâche élue se voit toujours attribuer un quantum entier, même si elle avait précédemment perdu le processeur en ayant utilisé partiellement un quantum de temps (suite à une demande d'E/S par exemple).

2.4.

Donnez l'algorithme de la routine de traitement de l'interruption horloge. Cette routine provoque une nouvelle élection.

2.5.

Pourquoi est-il important que la durée de traitement de l'interruption horloge ne soit pas trop longue ? Quelle doit être la durée maximum de traitement pour conserver une gestion du temps cohérente ?

2.6.

Quels sont les avantages et les inconvénients de toujours initialiser le registre Rtempo à la même valeur ?

3. QUANTUM ET TICK

Sous Unix, un tick correspond au passage à 0 du registre temporisateur. Certaines tâches de l'interruption horloge sont effectuées à chaque tick, alors que d'autres ne sont gérées que tous les N ticks, N dépendant de la tâche à traiter. Par exemple, la commutation de tâches est traitée toutes les 100 ms alors que l'intervalle entre 2 ticks est de 10 ms.

Par contre, la mise à jour de l'heure universelle, du temps utilisé par la tâche ou le test des alarmes à déclencher (réveil de tâche, réémission de paquets, …) est effectué à chaque tick.

3.1.

Modifiez la routine de traitement de l'interruption horloge pour gérer une commutation toutes les 100 ms avec un intervalle entre 2 ticks de 10 ms.

Chaque fois qu'un processus perd le processeur, le système réajuste sa priorité en appliquant la formule :

$$p\_pri = p\_user + p\_cpu + p\_nice$$

où p_pri est la priorité de la tâche, p_user la priorité de base d'une tâche utilisateur, p_cpu le temps utilisé par la tâche, comptabilisé en nombre de ticks, et p_nice le paramètre d'une commande nice effectuée par l'utilisateur.

3.2.

Modifiez le traitement précédent pour prendre en compte ce nouveau mécanisme.

3.3.

Lors du positionnement d'une alarme, l'instant de déclenchement peut être choisi à la micro-seconde près. Quelle est, en réalité, la précision du déclenchement ?

3.4.

Comment peut-on mesurer le temps passé par la tâche en mode utilisateur et en mode système ?

3 - ORDONNANCEMENT

1. RAPPELS

Un algorithme d'ordonnancement (scheduling) permet de choisir parmi un ensemble de tâches prêtes celle qui va occuper le processeur. Cet algorithme peut ou non utiliser des priorités affectées aux tâches.

Avec un algorithme batch sans réquisition, la tâche élue conserve le processeur jusqu'à sa terminaison : elle ne peut être interrompue ni par l'arrivée d'une autre tâche, ni par une interruption horloge (ex : FCFS, SJN). Avec un algorithme batch avec réquisition, seule l'arrivée d'une tâche plus prioritaire peut interrompre la tâche élue (ex : PSJN). La tâche élue est alors remise en tête de la file des tâches prêtes.

Dans un système en temps partagé, la tâche élue l'est pour (au plus) un quantum de temps.

Un bon algorithme d'ordonnancement doit à tout prix éviter les problèmes de **famine**. Celle-ci survient lorsque les critères utilisés par l'algorithme pour déterminer la tâche élue sont tels que l'une des tâches n'est jamais choisie et ne peut donc pas terminer son exécution.

2. ORDONNANCEMENT EN MODE BATCH

On considère l'ensemble de tâches suivant :

tâche	A	B	C	D	E
date dispo.	0	0	1	5	5
durée	5	4	2	1	2

2.1.

Le processeur est géré selon une stratégie SJN (Shortest Job Next). Représentez l'exécution des tâches sous forme de diagramme. Donnez pour chacune son temps de réponse et son taux de pénalisation. Y a-t-il un risque de famine ?

2.2.

Même question avec une stratégie PSJN (Preemptive SJN).

3. ORDONNANCEMENT EN TEMPS PARTAGE

L'algorithme Round-Robin

Dans l'algorithme d'ordonnancement circulaire ou **round-robin**, les tâches sont rangées dans une file unique. Le processeur est donné à la **première** tâche **prête** de la file. La tâche perd le processeur en cas d'entrée/sortie ou quand elle a épuisé son quantum de temps. Elle est alors mise en fin de la file d'attente des tâches. Si une tâche arrive au début de la file dans l'état "bloquée" (attente d'une E/S), elle reste en début de file et on parcourt la file pour trouver une tâche prête. Toute nouvelle tâche est mise à la fin de la file.

On dispose d'un ordinateur ayant une unité d'échange travaillant en parallèle avec la CPU. On considère les tâches suivantes :

	Temps CPU	E/S	Durée E/S
T1	300 ms	aucune	
T2	30 ms	toutes les 10 ms	250 ms
T3	200 ms	aucune	
T4	40 ms	toutes les 20 ms	180 ms

A t = 30 (resp. t = 40) la tâche T2 (resp. T4) fait une entrée/sortie avant de se terminer.

3.1.

Décrire précisément l'évolution du système : *instant*, *nature de l'événement* (commutation, demande d'E/S, fin d'E/S), *tâche élue*, *état de la file*. Donner pour chacune des tâches l'instant où elle termine son exécution. On prendra un quantum de 100 ms, et on supposera que les tâches sont initialement prêtes et rangées dans l'ordre de leur numéro.

NB : on suppose que l'UE gère deux disques, et que les E/S des tâches 2 et 4 se font sur des disques différents.

3.2.

Quels sont les avantages et les inconvénients de ce mécanisme d'ordonnancement ?

Y a-t-il un risque de famine ? Pourquoi ?

L'algorithme round-robin ne permet pas de prendre en compte le fait que certaines tâches sont plus urgentes que d'autres. On introduit donc des algorithmes d'ordonnancement qui gèrent des priorités éventuelles entre les tâches.

Ordonnancement avec priorités statiques

Dans ce type d'algorithme, chaque tâche dispose à sa création d'une priorité qui conserve la même valeur tout au long de son exécution.

3.3.

Donner un algorithme de gestion des tâches qui utilise ces priorités. En supposant qu'il existe 4 niveaux de priorité numérotés de 0 à 3 (0 étant la plus faible priorité), reprendre la question 3.1 pour cet algorithme en considérant les tâches suivantes :

	Temps CPU	E/S	durée E/S	Priorité
T1	300 ms	aucune		0
T2	30 ms	toutes les 10 ms	250 ms	3
T3	200 ms	aucune		1
T4	40 ms	toutes les 20 ms	180 ms	3
T5	300 ms	aucune		1

3.4.

Quels sont les avantages et les inconvénients de ce mécanisme d'ordonnancement ?

Y a-t-il un risque de famine ? Pourquoi ?

Ordonnancement avec priorités dynamiques

Dans ce type d'algorithme, les priorités affectées aux tâches changent en cours d'exécution. Plusieurs algorithmes sont possibles suivant le critère de changement de priorité.

3.5.

Trouvez un algorithme où l'évolution des priorités défavorise les tâches les plus longues.

3.6.

Décrivez en détail le début de l'exécution (t < 900 ms) de cet algorithme pour le modèle de tâches suivant :

	Temps CPU	périodicité
T1	15 ms	toutes les 150 ms
T2	200 ms	toutes les 300 ms
T3	1000 ms	-

On supposera qu'à l'instant 0 la file est T1, T2, T3 et qu'aux multiples de 300 ms, les tâches de type T1 arrivent avant les tâches de type T2. On prendra un quantum de 100 ms.

3.7.

La tâche T3 s'exécutera-t-elle entièrement ? Est-ce toujours le cas ? Donnez un exemple.

3.8.

Trouvez un algorithme où les tâches n'utilisant pas tout leur quantum de temps sont favorisées. Vérifiez que votre algorithme ne crée pas de problème de famine.

4 - PROCESSUS UNIX

0. RAPPELS

L'appel système `fork()` crée un processus fils qui diffère de son père uniquement par ses numéros de `pid` et de `ppid`. `fork()` renvoie 0 pour le fils et le `pid` du fils créé pour le père. Juste après le `fork()`, les deux processus disposent des mêmes valeurs de données et de pile. Mais il n'y a pas de partage : chaque processus a sa propre copie.

L'appel système `wait()` bloque un processus en attente de la fin de l'exécution d'un fils qu'il a créé. `wait()` renvoie le numéro du fils qui vient de se terminer. On peut passer à `wait()` un paramètre de type `int*` qui permet de récupérer des informations sur la terminaison du fils.

Les appels système de la famille `exec` lancent un programme exécutable. Si l'appel système est réussi, le processus faisant le `exec` se termine lors de la fin du processus qu'il a lancé : on ne revient pas d'un `exec` réussi. Si l'appel échoue (et dans ce cas seulement), les instructions qui suivent le `exec` sont exécutées.

1. EXECUTION D'UN FORK SIMPLE

Soit le programme C suivant :

```
main(){
   int pid;
   printf("debut\n");
   pid = fork();
   if (pid == 0) {
       printf("execution 1\n"); }
   else {
       printf("execution 2\n"); }
   printf("Fin\n");
   return EXIT_SUCCESS;
}
```

1.1.

Donnez les affichages effectués par le processus père et par le processus fils.

2. EXECUTION D'OPERATIONS FORK IMBRIQUEES

Soit le programme suivant :

```
1:   int main(int argc, char *argv[]) {
2:
3:        int a, e;
4:
5:        a = 10;
6:        if (fork() == 0) {
7:             a = a *2 ;
8:             if (fork() == 0) {
9:                  a = a +1;
10:                 exit(2);
11:            }
12:            printf(" %d \n", a);
13:            exit(1);
14:       }
```

```
15:       wait(&e);
16:       printf("a : %d ; e : %d \n", a, WEXITSTATUS(e));
17:       return(0);
18:  }
```

2.1.

Donnez le nombre de processus créés, ainsi que les affichages effectués par chaque processus.

2.2.

On supprime la ligne 10, reprenez la question 2.1. en conséquence.

2.3.

Modifiez le programme initial pour créer un processus zombie pendant 30 secondes.

3. CREATION DE PROCESSUS EN CHAINE

On souhaite faire un programme qui crée une chaîne de processus telle que le processus initial (celui du main) crée un processus qui à son tour crée un second processus et ainsi de suite jusqu'à la création de N processus (en plus du processus initial).

3.1.

Ecrivez ce programme. Représentez les processus créés pour N = 3.

3.2.

Modifiez le programme pour que le processus initial attende uniquement la fin de son fils.

3.3.

Modifiez le programme pour que le processus initial attende la fin de *tous* les processus créés.

4. MODIFICATION DU CODE EXECUTE : LA PRIMITIVE EXEC

On considère le programme suivant :

```
int main() {
    int p;
    p = fork();
    if (p == 0) {
       execl("/bin/echo", "echo", "je", "suis", "le", "fils", NULL);
    }
    wait(NULL);
    printf("je suis le pere\n");
    return EXIT_SUCCESS;
}
```

4.1.

Donnez le résultat de l'exécution de ce programme (on suppose que l'appel `execl` est réussi).

Soit le programme "nemaxe" suivant en C sous Unix, où "prog" est un programme qui ne crée pas de processus.

```
main(int argc, char*argv[]) {
    int retour;
    printf("%s\n", argv[0]);                                    // A
    switch (fork()) {
        case -1:
            perror("fork1()");
            exit(1);
        case 0:                                                 // B
            switch (fork()) {
                case -1:
                    perror("fork2()");
                    exit(1);
                case 0:                                         // C
                    if (execl("./prog","prog",0) == -1) {
                        perror("execl");
                        exit(1);
                    }
                    break;
                default:
                    exit(0);
            }
        default:
            wait(&retour);
    }
}
```

4.2.

Représentez tous les processus créés par ce programme sous forme d'un arbre, en vous servant des lettres en commentaires.

4.3.

Quels sont les ordres possibles de terminaison des processus ?

On change maintenant le nom de "nemaxe" par "prog" (celui de l'appel à execl).

4.4.

Représentez le début de l'arbre de processus.

4.5.

Toujours dans le cas où le programme s'appelle « prog », proposez une modification pour limiter à 2 le nombre d'occurrences de B qui sont créées.

5 - SEMAPHORES

1. RAPPELS

Une **ressource critique** est une ressource partagée entre plusieurs processus et dont l'accès doit être contrôlé pour préserver la cohérence de son contenu (variable, fichier, etc…). En particulier, ce contrôle peut impliquer un accès en **exclusion mutuelle**, c'est-à-dire que la ressource n'est jamais accédée par plus d'un processus à la fois.

Une **section critique** est une séquence d'instructions au cours de laquelle un processus accède à une ressource critique et qui doit être exécutée de manière indivisible.

L'**indivisibilité** signifie que deux sections critiques concernant une même ressource ne sont jamais exécutées simultanément mais toujours séquentiellement (dans un ordre arbitraire). Il y a ainsi exclusivité entre ces séquences d'instructions.

L'exclusion mutuelle n'est qu'un cas particulier de synchronisation. Une **synchronisation** est une opération influant sur l'avancement d'un ensemble de processus:

- établissement d'un ordre d'occurrence pour certaines opérations (envoyer / recevoir).
- respect d'une condition (rendez-vous, exclusion mutuelle, etc…).

L'attente active est la mobilisation d'un processeur pour l'exécution répétitive d'une primitive de synchronisation jusqu'à ce que la condition de synchronisation permette la continuation normale du processus.

Un sémaphore est un objet permettant de contrôler l'accès à une ressource en évitant l'attente active. Il est constitué d'un compteur et d'une file d'attente. La valeur du compteur dénombre, lorsqu'elle est positive, le nombre de ressources disponibles, lorsqu'elle est négative le nombre de processus en attente de ressource. La politique de gestion de la file d'attente est définie par le concepteur du système. A la création d'un sémaphore, la file est vide. Un sémaphore est manipulé à l'aide des opérations indivisibles suivantes :

`SEM *CS(cpt)` : Création d'un sémaphore dont le compteur est initialisé à "cpt".

`DS(sem)` : Destruction d'un sémaphore. Si la file n'est pas vide un traitement d'erreur doit être effectué.

`P(sem)` : Demande d'acquisition d'une ressource. Si aucune ressource n'est disponible, le processus est bloqué.

`V(sem)` : Libération d'une ressource. Si la file d'attente n'est pas vide, un processus est débloqué.

1.1.

Donnez le code des primitives P et V en utilisant les opérations suivantes :

- TACH *_courant_() : désigne la tâche en cours d'exécution (la tâche ELUE).
- void _insérer_(TACH tache, FILE *file) : insère une tâche dans la file
- TACH *_extraire_(FILE *file) : extrait une tâche de la file.

1.2.

Expliquez pourquoi ces primitives doivent être rendues indivisibles et comment on peut réaliser cette indivisibilité.

1.3.

Pourquoi n'utilise-t-on pas le masquage/démasquage des interruptions pour réaliser une section critique en mode utilisateur ?

1.4.

Un processus en train d'exécuter une section critique peut-il être interrompu par un autre processus ? Expliquez ce qui se passe alors.

On considère les trois processus suivants qui s'exécutent de manière concurrente, et le sémaphore Mutex initialisé à 1.

Processus A	*Processus B*	*Processus C*
(a) `P(Mutex);`	(d) `P(Mutex);`	(g) `P(Mutex);`
(b) `x = x + 1;`	(e) `x = x * 2;`	(h) `x = x - 4;`
(c) `V(Mutex);`	(f) `V(Mutex);`	(i) `V(Mutex);`

1.5.

On considère le scénario suivant : a d b g c e f h i

Donnez après chaque opération sur le sémaphore Mutex

- la valeur du compteur,
- le contenu de la file du sémaphore,
- l'état de chacun des processus (élu, prêt, bloqué).

Le scénario suivant est-il possible ? Justifiez :a d e b c f g h i

On considère les trois processus suivants qui s'exécutent de manière concurrente sur une machine. La variable **a** est une variable partagée, et on a les initialisations suivantes :

int a = 6; S1 = CS(1); S2 = CS(0);

Processus A	*Processus B*	*Processus C*
`P(S1);`	`a = a - 5;`	`V(S1);`
`a = a + 7;`		`P(S2);`
`V(S2);`		`a = a * 3;`

1.6.

Quelles sont les valeurs finales possibles de la variable a ? Donner pour chaque valeur la (les) suite(s) d'instruction qui amènent à cette valeur.

1.7.

- Ajoutez dans les programmes des processus les sémaphores nécessaires (et ceux-là seulement), pour obtenir dans toute exécution a = 34. Donnez les initialisations des sémaphores ajoutés.
- Même question pour a = 24.

1.8.

Quel sémaphore pourrait être supprimé sans modifier l'exécution de l'ensemble des processus ? Justifiez.

1.9.

On modifie la synchronisation de la manière suivante :

Processus A	***Processus B***	***Processus C***
`P(S1);`	`P(S2);`	`P(S2);`

```
P(S2);                  a = a - 5;              P(S1);
a = a + 7;              V(S2);                  a = a * 3;
V(S1);                  V(S1);                  V(S2);
V(S2);
```

S1 et S2 sont **tous les deux initialisés à 1**. Quelles sont les conséquences de cette modification ? Justifiez.

2. SYNCHRONISATION ET INTERBLOCAGE

Soient N processus utilisateurs U_1 à U_N partageant un ensemble de X serveurs et Y ressources (X > 0, Y > 0, N > X + Y). Ces processus sont synchronisés au moyen de 2 sémaphores S et R, représentant la disponibilité des serveurs et des ressources, selon le programme suivant.

```
Boucle
     Prologue:  P(S);  P(R);  V(S);
     Utilisation de ressource;
     Epilogue:  P(S);  V(R);  V(S);
Fin boucle
```

Le sémaphore S est initialisé à X et le sémaphore R est initialisé à Y.

2.1.

Quel est le nombre maximum n de processus pouvant se trouver simultanément en train d'utiliser une ressource ?

2.2.

Montrez, pour N = 2 et X = Y = 1 que l'exécution de ce système peut conduire à un interblocage. Pour X > 0, Y > 0 quelconques, quelle est la valeur minimum de N qui peut amener à un interblocage ?

2.3.

L'interblocage peut être évité en introduisant un nouveau sémaphore L, pour limiter le nombre de processus autorisés à entrer dans le prologue. Introduisez les opérations P(L) et V(L) nécessaires dans le prologue et l'épilogue. Précisez l'initialisation du sémaphore L.

6-7 - SEMAPHORES : PROBLEMES CLASSIQUES

1. LE MODELE PRODUCTEUR-CONSOMMATEUR

Soit un tampon T constitué de n cases dont chacune est destinée à recevoir un message. On nomme Producteur un processus qui dépose de l'information dans le tampon et Consommateur un processus qui retire de l'information. Un tel type de communication est indirect et asynchrone.

- **Indirect** : car les producteurs et les consommateurs ne s'adressent pas l'un à l'autre.
- **Asynchrone** : car il n'est pas nécessaire que les producteurs et les consommateurs travaillent au même rythme.

On se place ici dans le cas où chaque message est lu par un unique consommateur.

1.1.

Quels sont les contraintes à respecter pour l'accès aux cases du tampon ?

On considère les trois tâches utilisateur suivantes :

```
main() {
   sprod = CS(v1);
   scons = CS(v2);
   CT(Production);
   CT(Consommation);
   ...
}
```

```
Production() {
   while (1) {
      Produire(message);
      P(sprod);
      Déposer(message);
      V(scons);
   }
}
```

```
Consommation() {
   while (1) {
      P(scons);
      Retirer(message);
      V(sprod);
      Consommer(message);
   }
}
```

`Production()` est le code exécuté par un producteur et `Consommation()` le code exécuté par un consommateur. La primitive `Produire()` crée un message et la primitive `Consommer()` traite un message reçu.

`main()` est le programme d'initialisation. Il crée ici un processus producteur et un processus consommateur.

1.2.

Dans quel cas faut-il bloquer l'exécution du producteur ? du consommateur ? En déduire les valeurs initiales de v1 et v2.

Système multi-producteurs / multi-consommateurs

On se place ici dans le cas où `main()` a lancé plusieurs producteurs et plusieurs consommateurs. Les tâches `Production()` et `Consommation()` ne sont pas modifiées, mais on s'intéresse maintenant au contenu des procédures `Déposer()` et `Retirer()`. Ces procédures manipulent le tampon avec les indices respectifs `id` et `ir`, initialisés à 0. Ces indices permettent une gestion circulaire du tampon.

```
Déposer(message)
MESS *message;
{
   (a)  T[id] = message;
   (b)  id = (id + 1) % n;
}
```

```
Retirer(message)
MESS *message;
{
   (c)  message = T[ir];
   (d)  ir = (ir + 1) % n;
}
```

1.3.

Montrer que plusieurs producteurs (resp. consommateurs) peuvent déposer (resp. retirer) dans la même case. Comment doit-on modifier les procédures `Déposer()` et `Retirer()` pour que la gestion des cases du tampon reste correcte, même lorsqu'on a plusieurs producteurs et plusieurs consommateurs ?

On désire assurer une plus grande indépendance de fonctionnement des producteurs et des consommateurs. Un producteur en train de produire un message ne doit pas empêcher d'autres producteurs d'acquérir des cases vides pour produire. Pour cela il faut dissocier :

- l'acquisition d'une case vide, de son remplissage.
- l'acquisition d'une case pleine, de son vidage.

1.4.

Réécrire simplement les procédures `Déposer()` et `Retirer()`. Montrer que dans ce cas, il faut rajouter des sémaphores pour garantir qu'un producteur ne produira pas dans une case en train d'être consommée.

1.5.

Comment les performances pourraient-elles être encore améliorées ?

2. LE PROBLEME DES LECTEURS / ECRIVAINS

Soit un fichier pouvant être accédé en lecture ou en écriture. Pour garantir la cohérence des données de ce fichier, les processus qui souhaitent y accéder sont rangés dans deux classes, en fonction du type d'accès qu'ils demandent :

- les processus lecteurs ;
- les processus écrivains.

Le nombre de processus dans chaque classe n'est pas limité. On établit le protocole d'accès de la manière suivante :

- il ne peut y avoir simultanément un accès en lecture et un accès en écriture ;
- les écritures sont exclusives entre elles (au plus un accès en écriture à un instant donné).

Par contre, plusieurs lectures peuvent avoir lieu simultanément.

La structure des processus est la suivante :

```
            Lecteur                          Ecrivain
OuvreLecture() ;                    OuvreEcriture() ;
   /* Accès en lecture */               /* Accès en écriture */
FermeLecture() ;                    FermeEcriture() ;
```

où la procédure OuvreLecture (resp. OuvreEcriture) contient les instructions que doit effectuer un lecteur (resp. un écrivain) avant de pouvoir accéder à la ressource, et la procédure FermeLecture (resp. FermeEcriture) contient les instructions qu'un lecteur (resp. un écrivain) exécute lorsqu'il relâche la ressource.

On considère dans un premier temps la politique d'accès suivante : si la ressource est disponible ou s'il y a déjà des requêtes de lecture en cours, une nouvelle requête de lecture est traitée immédiatement même s'il y a des demandes d'écriture en attente.

2.1.

Quelles sont les conditions de blocage pour un écrivain ? Pour un lecteur ? Donnez les sémaphores et les variables partagées nécessaires pour tester ces conditions, ainsi que leur initialisation.

2.2.

Programmez cette synchronisation de processus. Quel est l'inconvénient de cette stratégie d'accès ?

Pour assurer l'équité, on veut maintenant gérer les accès dans l'ordre FIFO : lors de leur arrivée, tous les processus sont rangés dans une même file. On extrait de cette file soit un unique écrivain, soit le premier lecteur d'un groupe. Dans le deuxième cas, on autorise les lecteurs suivants dans la file à passer, jusqu'à ce que le processus en tête de file soit un écrivain.

On fera dans cette question l'hypothèse que la file d'attente des sémaphores est FIFO (ce qui n'est pas forcément vérifié dans le cas général).

2.3.

Par rapport à la question 2.1, quel sémaphore supplémentaire est nécessaire pour réaliser cette stratégie ? Programmez cette synchronisation de processus.

2.4.

Montrez que, dans la solution de la question précédente, une mauvaise utilisation des sémaphores (inversion de deux opérations P) peut conduire à un interblocage.

8-9 – MEMOIRE

1. MEMOIRE LINEAIRE

Dans ce type de gestion, la totalité du programme et des données d'un processus doivent être chargés en mémoire pour pouvoir exécuter le processus. L'allocation d'un espace mémoire à un processus peut se faire suivant différentes stratégies :

- La stratégie First Fit recherche le premier espace en mémoire de taille suffisante ;
- La stratégie Best Fit recherche le plus petit espace libre pouvant contenir le processus ;
- La stratégie Worst Fit recherche le plus grand espace libre pouvant contenir le processus.

1.1.

Soit une liste des blocs libres composée dans l'ordre de blocs de 100K, 500K, 200K, 300K et 600K. Comment des processus de tailles respectives 212K, 417K, 112K, et 426K, arrivés dans cet ordre, seraient-ils placés en mémoire

- avec une stratégie Best-Fit ?

- avec une stratégie First-Fit ?

1.2.

Quelle est la place réellement occupée par l'ensemble des processus de la question précédente si l'on alloue la mémoire par blocs de 10 K ? Comment s'appelle ce phénomène ?

2. SEGMENTATION

La segmentation divise l'ensemble des informations nécessaires à l'exécution d'une tâche en segments. Un segment, de longueur quelconque, représente une zone de mémoire pour la tâche. Il est associé à un aspect spécifique de la représentation de la tâche en mémoire (code, données, pile, ...). Un segment est chargé entièrement dans des zones de mémoire contiguës.

On note <s, d> une adresse segmentée composée d'un numéro de segment et d'un déplacement. On considère une tâche utilisant 3 segments, dont la table des segments à un instant donné est :

	B	L	p	m	u	xle
0		132	0			
1	7435	400	1	0	0	011
2		325	0			

où `B` est la base, `L` la longueur du segment, `p`, `m` , `u` et `xle` respectivement les bits de présence, de modification, d'utilisation et de droits d'accès au segment. On suppose que la tâche fait un accès en écriture à la donnée située à l'adresse <1, 260>.

2.1.

Quelle est l'adresse physique de cette donnée en mémoire ?

2.2.

Donnez les modifications effectuées dans la table lors de cet accès, en précisant si elles sont effectuées par le système ou le matériel.

2.3.

Que se passe-t-il lors d'un accès à l'instruction d'adresse <0, 125> ?

3. PAGINATION

Le principe de la pagination réside dans la division de la mémoire en zones de tailles fixes appelées "pages". L'espace de travail d'un processus est divisé en pages. Quand le processus est exécuté, seules les pages dont il a besoin sont chargées en mémoire centrale.

On dispose d'une machine monoprocesseur, ayant une mémoire centrale de 32K mots, une page faisant 512 mots (64 pages en mémoires centrales), un bloc (unité de stockage sur le disque) faisant 512 mots. On considère le programme suivant :

Adresse	Instruction	Signification
E	`Loadx N`	reg d'index X = N
	`Load S,X`	reg accumulateur A = S[X]
	`Add V`	A = A + V
	`Store T,X`	T[X] = A
	`Subx K`	X = X – K
	`Brxpz E + 1`	Si X >= 0 alors aller en E + 1
	`Stop`	

Où:

- X est un registre d'index,
- A est un registre accumulateur,
- E est l'adresse du début du programme : 512 * 1 + 508,
- S et T sont des tableaux de taille [0 .. N], S [0] étant à l'adresse 512 * 11 + 168 et T[0] se trouvant à l'adresse 512 * 12 +456,
- N est une constante égale à 799 se trouvant à l'adresse 512 * 6 + 500,
- V est une constante égale à 2 se trouvant à l'adresse 512 * 8 + 100,
- K est une constante égale à 2 se trouvant à l'adresse 512 * 8 + 10.

Ce programme effectue la chose suivante :

```
x = N;
do {T[X] = S[X] + V; x -= K} while (x >= 0);
```

En fin d'exécution, les pages modifiées sont recopiées en mémoire secondaire.

3.1.

Sachant que chaque instruction, constante ou variable simple occupe un mot mémoire, représenter l'espace d'adressage du processus (numéro de page / déplacement dans la page de chaque donnée). Quels droits doit-on affecter aux différentes pages du processus ?

Les actions de gestion de la mémoire peuvent être représentées à l'aide des opérations suivantes :

Trap (p) : Défaut de page pour la page p.

Charg (p,c) : Chargement de la page p à la place c en mémoire centrale.

Dech (p,c) : Déchargement de la page p se trouvant à la place c en mémoire centrale.

Mod (p,c,d) : Modification de la table des pages, p : numéro de page du processus, c place en mémoire centrale, d droits d'accès de la page.

Le processus précédent dispose pour s'exécuter des pages de mémoire centrale 17, 21, 22, 23, 37 et 42.

3.2.

En utilisant un algorithme premier chargé/premier déchargé et en supposant qu'au départ, aucune page n'est chargée, décrire les actions de gestion de mémoire sous la forme d'une suite composée des opérations précédentes. Donner la table des pages finale.

4. SEGMENTATION PAGINEE

On considère une mémoire *segmentée paginée*. La taille des pages est de 512 mots.

Le processus P possède 3 segments : le segment 0 pour le code, le segment 1 pour la pile et le segment 2 pour les données. Le segment 0 a 1500 mots, le segment 1 en a 2000 et le segment 2 en a 3000. On suppose que la table des segments et les tables de pages sont déjà chargées en mémoire.

4.1.

Quels sont les droits associés à chaque segment ? Quel est l'avantage de découper ainsi l'espace d'adressage du processus ?

Seules les pages suivantes ont été chargées (on donne le doublet <numéro de segment, numéro de page>) :

<0, 0> à l'adresse physique 2560

<1, 2> à l'adresse physique 4096

<2, 3> à l'adresse physique 1024.

4.2.

Dans quelle case mémoire est chargée la page <0, 0> ?

4.3.

Décrire brièvement ce qui se passe sur une tentative de lecture en <0, 1678>, <1, 567> et <2, 1600> (Ces adresses sont au format segmenté).

4.4.

Connaît-on l'adresse physique de la variable d'adresse virtuelle <1, 1060> ? Si oui, quelle est-elle ? Sinon, justifiez.

5. CACHE ET MEMOIRE VIRTUELLE

Avec une mémoire paginée, l'accès à une donnée nécessite deux accès à la mémoire : un pour lire la table des pages, l'autre pour lire la donnée. Pour ne pas diviser par deux les performances du système, on utilise un cache spécial, appelé TLB (Translation Look-aside Buffers), qui permet de stocker le couple <n° de page, case mémoire>. Si le couple correspondant à la page accédée est présent dans la TLB, alors l'accès à la table des pages devient inutile. Typiquement, la TLB contient entre 8 et 2048 entrées.

5.1.

On suppose qu'un accès à la TLB demande 20 ns et qu'un accès mémoire demande 100 ns. Pour une TLB avec un taux de hit de 90%, quel est le temps moyen d'accès à une donnée ?

En plus de la TLB qui permet une translation rapide de l'adresse logique à l'adresse physique, le système dispose d'un cache pour améliorer la vitesse d'accès aux données. Le cache est une mémoire à accès rapide, structurée en lignes. Chaque ligne stocke un bloc de données d'une part, tout ou partie de l'adresse du bloc permettant d'identifier le bloc de façon unique, ainsi que diverses informations relatives à la validité, l'ancienneté, ... dudit bloc.

La combinaison de la mémoire cache et de la mémoire virtuelle est délicate : ou bien le cache contient les adresses physiques des blocs, et dans ces conditions la traduction adresse virtuelle - adresse physique doit être faite entre le processeur et le cache; ou bien le cache contient les adresses virtuelles des blocs, et la traduction adresse virtuelle-adresse physique a lieu entre le cache et la mémoire.

5.2.

Représentez schématiquement l'organisation de la hiérarchie mémoire dans les deux cas. Quels sont les avantages et inconvénients de chaque organisation ?

10 - REMPLACEMENT DE PAGES

1. ALGORITHME OPTIMAL

Lors d'un défaut de page, on remplace la page qui sera utilisée le plus tard possible. Il est clair que cet algorithme n'est pas réalisable car il faudrait pour cela connaître l'avenir; cependant, il est d'une grande utilité pour étalonner les autres algorithmes.

On suppose qu'un processus fait référence à ses pages dans l'ordre suivant:

5 0 1 2 0 3 0 4 2 3 0 3 2 1 2

1.1.

Si on ne limite pas le nombre de cases allouées au processus, combien de défauts de page sont provoqués par cette suite d'accès ? Combien de cases au minimum faut-il allouer au processus pour atteindre ce nombre ?

1.2.

Donnez l'évolution de la table des pages ainsi que le nombre de défauts de pages si on alloue au processus 3 cases.

2. ALGORITHME FIFO

Lors d'un défaut de page, on remplace la plus ancienne page qui ait été chargée.

2.1.

On suppose qu'un processus fait référence à ses pages dans l'ordre suivant:

5 0 1 2 0 3 0 4 2 3 0 3 2 1 2

Donnez l'évolution de la table des pages ainsi que le nombre de défauts de pages si on alloue au processus 3 cases.

3. ALGORITHME FINUFO (FIRST IN NOT USED FIRST OUT)

On suppose qu'il y a un bit R dans le descriptif de chaque page. A chaque accès à une page, le matériel met le bit R à 1. Lorsqu'une page est chargée, le bit R est aussi mis à 1.

Une liste des pages triées selon leur date de chargement est gérée : la page en tête de liste est la plus récemment chargée, celle en fin de liste est la plus anciennement chargée. Lors d'un défaut de page, la page la plus anciennement chargée est examinée. Si son bit R vaut 1, elle est remise en tête de liste et son bit R passe à 0 (elle dispose ainsi d'une deuxième chance). La page déchargée est donc celle qui est la plus anciennement chargée et dont le bit R vaut 0.

3.1.

On suppose qu'un processus fait référence à ses pages dans l'ordre suivant:

5 0 1 2 0 3 0 4 2 3 0 3 2 1 2

Donnez l'évolution de la table des pages ainsi que le nombre de défauts de pages si on alloue au processus 3 cases.

4. ALGORITHME LRU (LEAST RECENTLY USED)

Lors d'un défaut de page, on remplace la page la moins récemment utilisée.

4.1.

On suppose qu'un processus fait référence à ses pages dans l'ordre suivant:

```
5 0 1 2 0 3 0 4 2 3 0 3 2 1 2
```

Dans le cas où on alloue au processus 3 cases, donnez l'évolution de la liste des pages chargées classées par ordre de dernière utilisation. Quel est le nombre de défauts de pages ?

5. ALGORITHME PAR GESTION DE FENÊTRE (WS : WORKING SET)

Dans cet algorithme une fenêtre (i.e. l'espace de travail ou *working set*) est un intervalle de la suite des accès aux pages. Le nombre de pages contenues dans la fenêtre varie en fonction des accès. Périodiquement, la fenêtre est mise à jour en ne conservant que les pages utilisées durant la période.

6.1.

On suppose qu'un processus fait référence à ses pages dans l'ordre suivant:

```
5 0 1 2 0 3 0 4 2 3 0 3 2 1 2
```

En supposant que la fenêtre est mise à jour tous les 3 accès mémoire (on ne conserve donc dans la fenêtre que les pages accédées lors des 3 dernières références), donnez l'évolution de la liste des pages chargées. Quel est le nombre de défauts de pages ?

11 - FICHIERS

On considère dans l'ensemble de ce TD une disquette formatée de 1.44 Mo divisée en blocs de 1Ko. Le stockage d'un fichier sur la disquette se fait en lui allouant un certain nombre de blocs qui peuvent ne pas être contigus. Le système doit donc gérer des structures de données lui permettant de savoir quels sont les blocs libres et les blocs occupés de la disquette.

1. GESTION D'ESPACE LIBRE

On suppose ici que les blocs 0 à 13 de la disquette sont occupés, ainsi que tous les blocs à partir de 20.

On considère dans un premier temps que le système gère l'espace libre au moyen d'un vecteur binaire (bitmap).

1.1.

Quelle est la taille du vecteur binaire ? Combien de blocs occupe-t-il ? Donnez la valeur de ses 24 premières entrées.

Une autre stratégie consiste à gérer l'espace libre sous forme groupée : un bloc spécial (appelé superbloc) contient (entre autres) :

- En 1ère position : le numéro du prochain bloc contenant une liste de blocs libres,
- une liste de blocs libres.

Lorsqu'on utilise le dernier numéro de bloc qu'il contient, on recopie dans le bloc spécial le contenu du bloc correspondant à ce dernier numéro.

1.2.

On se place dans le cas où les numéros de blocs sont codés sur 2 octets. Quelle est, initialement, la taille totale nécessaire pour gérer l'information sur les blocs libres de la disquette ?

1.3.

Que pensez-vous de la place utilisée pour la gestion dans les deux stratégies ?

2. GESTION D'ESPACE OCCUPE

Une première façon de gérer l'espace occupé par un fichier est de lui allouer un (ou plusieurs) bloc(s) d'index. Un répertoire est alors une table donnant pour chaque fichier les coordonnées de son bloc d'index. Le bloc d'index contient les numéros des blocs occupés par le fichier.

2.1.

On suppose qu'un fichier F1 occupe sur disque les blocs 25, 26, 37 et 63. Son bloc d'index est stocké dans le bloc 13. Un fichier F2 occupe les blocs 19 et 43, son bloc d'index est stocké dans le bloc 16. Donnez le contenu du répertoire et des blocs d'index.

A l'ouverture d'un fichier, son bloc d'index est chargé en mémoire.

2.2.

Quel est le nombre d'accès mémoire et le nombre d'E/S nécessaires pour lire le 4000ème octet du fichier F1 ?

Dans le système de gestion des fichiers d'UNIX, à chaque fichier est associée une structure de données appelée `inode`. Dans cette structure, on trouve (entre autres) 13 entrées :

- les 10 premières entrées référencent des blocs de données sur le disque,
- la 11ème référence un bloc de contrôle qui référence des blocs de données (simple indirection),
- la 12ème référence un bloc de contrôle qui référence des blocs de contrôle qui référencent des blocs de données (double indirection),
- la 13ème référence un bloc de contrôle qui référence des blocs de contrôle qui référencent des blocs de contrôle qui référencent des blocs de données (triple indirection).

Chaque bloc de contrôle permet de référencer au maximum 128 blocs (contrôle ou données).

On considère trois fichiers F1, F2 et F3 ayant les caractéristiques suivantes :

F1 7 blocs de données,

F2 11 blocs de données,

F3 139 blocs de données.

2.3.

Donnez le nombre d'indirections utilisées pour le stockage des fichiers F1, F2 et F3.

2.4.

Avec des blocs de 512 octets, quelle est la quantité maximum de données que l'on peut stocker dans un fichier par cette méthode ?

3. GESTION MIXTE

La FAT (File Allocation Table) est une structure de données permettant de gérer à la fois l'espace libre et l'espace alloué. Il s'agit d'un tableau avec une entrée par bloc.

Dans un répertoire, on trouve pour chaque fichier le numéro du premier bloc occupé par le fichier. L'entrée correspondante de la FAT contient l'adresse du 2ème bloc occupé, etc… Le dernier contient une valeur spéciale "fin de fichier", les blocs vides sont à 0.

3.1.

On suppose que chaque numéro de bloc est stocké sur 2 octets. Quelle est la taille de la FAT pour une disquette de 1.44 Mo avec des blocs de 1 Ko ?

3.2.

On suppose qu'un fichier F1 occupe sur disque les blocs 25, 26, 37 et 63. Donnez le contenu du répertoire et des entrées de la FAT correspondant au stockage de ce fichier.

3.3.

Si la FAT est chargée en mémoire, quel est le nombre d'accès mémoire et le nombre d'E/S nécessaires pour lire le 4000ème octet du fichier F1 ?

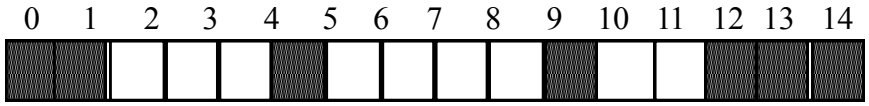
4. ALLOCATION D'UN FICHIER SUR DISQUE

On veut gérer l'allocation de données sur disque pour des fichiers utilisateurs. Le disque est structuré en **blocs**. On suppose que le système alloue à chaque fichier un bloc d'**index**.

4.1.

Quelle est la méthode de gestion d'espace libre qui vous semble la plus adaptée si l'on veut optimiser (en minimisant les déplacements de tête) l'accès séquentiel aux fichiers ?

La configuration de la piste 24 de la disquette est la suivante :



où les parties grisées représentent des blocs occupés, les parties blanches des blocs libres.

Un utilisateur réalise les allocations successives dans l'ordre suivant :

Création du fichier F1 avec 3 blocs de données

Création du fichier F2 avec 1 bloc de données

Création du fichier F3 avec 2 blocs de données

4.2.

En essayant d'optimiser l'accès séquentiel, faites un schéma des blocs du disque après l'allocation. Indiquez par un I les blocs d'index et par un D les blocs de données. Indiquez également le nom du fichier auquel le bloc appartient.

Le système réserve `m` tampons en mémoire pour stocker les données du disque. Ces tampons servent de cache au disque. Chaque bloc de données lu depuis le disque est mémorisé dans un des tampons. A chaque nouvel accès le système teste si la donnée n'est pas déjà présente en mémoire pour éviter une entrée/sortie. Les écritures sont faites directement dans le cache et ne sont reportées sur disque que lorsque les données sont expulsées du cache.

4.3.

Décrivez l'ensemble des actions à effectuer pour chaque demande de lecture d'un bloc de données.

Lorsqu'il n'y a plus de tampons libres, le système réquisitionne un tampon plein en choisissant le plus anciennement utilisé (algorithme LRU).

4.4.

Pourquoi la stratégie LRU est-elle moins coûteuse ici que pour la gestion de la mémoire virtuelle ?

Soit un programme effectuant la suite de références suivante (on représente des triplets <fichier, numéro de bloc de données, mode d'accès (L: Lecture, E: Ecriture)>) :

<F2, 0, E>, <F1,1, E>, <F2, 0, L>, <F3, 1, L>, <F3, 0, L>

4.5.

On prend `m = 4` et on considère qu'au départ tous les tampons sont vides. Représentez l'évolution du contenu de chacun des tampons (nom du fichier, numéro de bloc). Indiquez le nombre d'entrées/sorties engendrées par cette séquence (sans oublier l'accès aux blocs d'index).

5. PARTAGE DE FICHIERS SOUS UNIX

5.1.

Lorsqu'un utilisateur accède à un fichier, quelles sont les informations dont le système a besoin pour pouvoir réaliser cet accès ? Proposez une structure de données permettant au système de gérer ces informations.

5.2.

On considère 2 fichiers fic1 , de taille 50 octets, et fic2, de taille 30 octets, ayant pour inodes respectifs 17 et 21. Deux processus p1 et p2 effectuent les opérations suivantes :

```
Processus p1                        Processus p2
d1 = open("fic1", O_RDONLY) ;       d1 = open("fic1", O_RDONLY) ;
d2 = open("fic2", O_APPEND) ;       read (d1, buf, 10) ;
```

où « `O_APPEND` » désigne une écriture en fin de fichier, `buf` est un tableau de taille 20, et `read(int fd, void *buf, size_t count)` est un appel système qui lit au maximum `count` octets dans le fichier de descripteur `fd` et stocke les octets lus à l'adresse `buf`.

Représentez la structure de données après ces opérations.

On considère le programme C suivant :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
    int no_fich;
    char car_lu;
    int nbcar;

    no_fich = open("Mon_fichier", O_RDONLY);

    if (fork() == -1) {
        perror("fork");
          exit(1);
    }

    while ((nbcar = read(no_fich, &car_lu, 1)) != 0) {
        printf("%c (%d)\n", car_lu, getpid());
          sleep(1) ;
    }

    close(no_fich);
    return 1;

}
```

La valeur renvoyée par `read` est le nombre de caractères effectivement lus, zéro si la fin de fichier est atteinte.

5.3.

On suppose que le fichier « Mon_fichier » existe et contient le texte suivant :

```
Systeme
```

Quel est le résultat de l'exécution de ce programme ? Expliquez.

5.4.

Que se passe-t-il si l'on remplace dans le programme précédent l'appel système `open` par la fonction C `fopen` et l'appel système `read` par la fonction `fgetc` ?