SCIENCES SORBONNE UNIVERSITÉ

UFR 919

Licence d'Informatique 2024 -2025

LU3IN010

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.

Un *processus,* ou une tâche, correspond à un programme en cours d'exécution. Il est caractérisé par son contexte matériel (la valeur du compteur ordinal, les valeurs des registres), et son contexte mémoire (le code et les variables).

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 ou si une tâche plus prioritaire est prête à s'exécuter (réquisition).

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 maximum de temps processeur allouée à la tâche active (de l'ordre de 10 à 200 ms).

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

Une *unité d'échange* (ou contôleur) sert à effectuer des transferts entre la mémoire principale et un périphérique sans utiliser le processeur centrale (CPU). Elle reçoit un ordre du processeur, effectue le transfert direct en mémoire (*DMA : Direct Memory Access*), 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 en supposant que T1 est initialement élue. 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 en supposant que $t \leq (n - 1) c$. Calculez r 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 $t = T$, $t \leq (n-1)c+NC$ et $T \leq nc+(N-1)C$.

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. INTERRUPTION

1.1

Dans quel mode s'exécute-t-on le traitement d'une interruption ?

Le traitement d'interruptions utilise la pile utilisateur ou la pile système ?

1.2

Indiquez les différents types d'interruption et leur priorité

1.3

Soit le scénario suivant avec un seul processus P : on suppose qu'à t=5ms un appel système est appelé par P, cet appel doit s'exécuter pendant 4 ms, à t=7ms une interruption horloge a lieu dont le traitement dure 1ms, à t=9ms une interruption disque est déclenchée dont le traitement dure 1ms.

Dessinez un diagramme temporel où sont indiqués les traitements de l'appel système, l'interruption disque et de l'interruption horloge, aussi que les instants où ils se terminent.

1.4

On considère maintenant que le traitement de l'interruption horloge dure 3ms.

Dessinez le nouveau diagramme temporel.

2. GESTION D'UNE INTERRUPTION HORLOGE

On considère un processeur qui dispose d'un registre horloge 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).

L'interruption horloge est utilisée à la fois pour tenir à jour l'heure universelle dans une variable globale walltime, pour instaurer un tour de rôle avec quantum d'exécution, pour comptabiliser le temps CT utilisé par une tâche et pour gérer les alarmes.

Le système gère une table `proc` des descripteurs de tâches : proc[i] contient l'ensemble des informations relatives à la tâche i. Le système dispose d'une variable *current* qui contient l'indice de la tâche élue.

On suppose dans un premier temps 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). Le quantum a une valeur fixée à QX microsecondes.

2.1.

Quelles informations doit maintenir le système pour chaque tâche de la table proc afin de gérer la commutation et le temps ?

2.2.

Donnez de code de la routine clock() permettant de traiter l'interruption horloge. Cette routine provoque une nouvelle élection.

La fonction int Election() retourne l'indice de la prochaine tâche élue.

La fonction void save(int i) sauvegarde les registres de la tâche i.

La fonction void restore(int i) restaure les registres de la tâche i.

La fonctoin void process_alarm() traite les alarmes.

Sous Unix, un tick correspond au passage à 0 du registre temporisateur. Certaines opérations 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.

En revanche, 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.

2.3.

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.

2.4.

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 - 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 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 avec réquisition (appelée aussi Préemption), l'arrivée d'une tâche prête 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) qui choisit la tâche prête ayant les plus petit d'exécution restant. 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 (UE) travaillant en parallèle avec la CPU.

On suppose que l'UE gère plusieurs disques, et que les E/S des tâches se font sur des disques différents.

On considère les tâches suivantes :

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

A t = 20, la tâche T2 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.

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 forte 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		3
T2	20 ms	toutes les 10 ms	250 ms	0
T3	200 ms	aucune		2

T4	40 ms	toutes les 20 ms	180 ms	0
T5	200 ms	aucune		2

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.

3.4.

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 :

```
int 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 principal se termine après tous les autres processus.

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.

```
int 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.

2. ACCES A UNE VARIABLE PARTAGEE

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);`

2.1.

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);

Pour simplifier, on suppose que les opérations sur **a** sont indivisibles.

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

2.2.

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.

2.3.

- 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.

2.4.

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

2.5.

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.

3. SYNCHRONISATION

Nous considérons N processus P0, P1, P2 …., PN-1 créés dans cet ordre. Chaque processus *Pi* fait le code suivant :

```
Pi :

v = calcul(i);
printf ("id: %d, valeur:%d \n", i, v);
```

3.1.

Expliquez pourquoi les N affichages ne seront pas forcement faits dans l'ordre croissant des identifiants de processus.

3.2.

Modifier le code de Pi pour garantir que les affichages soient faits dans **l'ordre croissant des identifiants des processus,** autrement dit, dans l'ordre P_0, P_1, P_2 …., P_{N-1}. Notez que seuls les affichages doivent se faire dans l'ordre, l'appel à la fonction calcul par les processus fils peut se faire dans n'importe quel ordre

Nous voulons modifier le programme ci-dessus en considérant un processus de plus P_N. Comme dans la question précédente, les processus P_0, P_1, …, P_{N-1} appellent la fonction *int calcul (int i)* en passant comme paramètre leur *id* respectif. Cependant, ils ne font pas

l'affichage qui sera fait par P_N. Par conséquent, le processus P_N, qui n'appelle pas la fonction *calcul,* est dédié à l'affichage qui doit toujours être fait dans l'ordre des créations des autres processus fils (P_0, P_1, puis P_2 …., P_{N-1}).

Pour assurer une telle synchronisation des affichages, votre programme ne peut utiliser que des sémaphores et, si nécessaire, des variables partagées.

3.3.

Donnez le code en C du programme demandé en spécifiant l'initialisation des sémaphores et/ou variables partagées utilisés.

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

```
Retirer(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.

3. LE PROBLEME DU SAS

On souhaite décrire le comportement de clients qui entrent dans une banque par l'intermédiaire d'un sas. Ce sas a deux portes qui ne doivent jamais être ouvertes simultanément.

Chaque client qui veut entrer exécute successivement les deux procédures suivantes :

ENTRER_SAS : bloque les clients dans le sas tant que la première porte est ouverte.

ENTRER_BANQUE : permet au client d'entrer lorsque la deuxième porte est ouverte.

NB : on ne s'intéresse pas à ce qui se passe à la sortie de la banque.

On suppose dans un premier temps que le sas d'entrée a une capacité de N clients. Les clients ne peuvent entrer dans la banque que lorsque le sas est plein.

3.1.

Que pensez-vous de cette gestion du sas ?

3.2.

Donnez la liste et la signification des sémaphores que vous utilisez pour programmer les procédures ENTRER_SAS et ENTRER_BANQUE, ainsi que la valeur d'initialisation de leur compteur.

3.3.

Programmez les procédures ENTRER_SAS et ENTRER_BANQUE.

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. 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** stockées dans des **cases** de la mémoire physique. Quand le processus est exécuté, seules les pages dont il a besoin sont chargées en mémoire centrale.

On dispose d'une machine monoprocesseur avec des pages faisant 512 mots. On considère le programme suivant :

Adresse	Instruction	Signification
E	`Loada 549`	reg accumulateur A = 549
E+1	`Storea N`	N = A
E+2	`Loadx N`	reg d'index X = N
E+3	`Loada T,X`	reg accumulateur A = T[X]
E+4	`Inca`	A = A + 1
E+5	`Storea S,X`	S[X] = A
E+6	`Subx`	X = X – 1
E+7	`Brxpz E + 3`	Si X >= 0 alors aller en E + 3
E+8	`Stop`	

Où:

- X est un registre d'index,
- A est un registre accumulateur,
- E, l'adresse du début du programme est égale à l'adresse virtuelle 1018 (512 * 1 + 506),
- S et T sont des tableaux de taille [0 .. 549], S [0] étant à l'adresse 5800 (512 * 11 + 168) et T[0] se trouvant à l'adresse 6350 (512 * 12 +206),
- N est une variable se trouvant à l'adresse 3572 (512 * 6 + 500)

Ce programme effectue le traitement suivant :

```
N = 549
x = N;
do {S[X] = T[X] + 1;
    x --;
} while (x >= 0);
```

2.1.

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 dans la case 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, et 37.

2.2.

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.

3. 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	xrw
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 `xrw` 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>.

3.1.

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

3.2.

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

3.3.

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

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 ?

5.2.

Dans un format de TLB classique <page,case>, quelles opérations faut-il faire sur la TLB lors d'une commutation ?

Quelles modifications dans le format de la TLB permettraient d'éviter de faire ces opérations ?

10 - REMPLACEMENT DE PAGES

L'exécution d'un programme engendre une longue suite d'accès mémoire : il y a souvent plusieurs accès pour une seule instruction et plusieurs dizaines de milliers d'instructions exécutées. L'efficacité d'une politique de remplacement ne peut donc se juger que sur des suites de taille significative, ce qui n'est pas le cas ici. Les exercices proposés dans ce TD ont pour but de vous aider à assimiler les mécanismes des différentes stratégies de remplacement.

Pourquoi l'exécution d'un programme engendre-t-elle (souvent) des défauts de page ? Durant un délai faible, un programme n'utilise qu'une faible partie de son espace de travail. Il est donc inutile et coûteux de charger la totalité de cet espace. Mais en n'en chargeant qu'une partie, il va arriver un moment où l'on accèdera à un élément non chargé. On peut charger cet élément *à la place* d'un élément déjà en mémoire : la stratégie de remplacement va alors déterminer l'élément victime. Dans ce cas, l'espace alloué à la tâche est de taille fixe. On peut aussi *ajouter* cet élément à l'espace de travail. La taille de celui-ci augmente donc, mais comme on ne peut indéfiniment augmenter l'espace de toutes les tâches, il faut à un moment supprimer certains éléments en mémoire. La taille de l'espace varie en fonction des besoins de la tâche, c'est le fonctionnement des stratégies de type *Working Set*.

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.

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 ?

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.

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.

5.1.

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

1. ACCES AUX DONNEES D'UN FICHIER - INDEXATION

Dans le système de gestion des fichiers d'UNIX, à chaque fichier est associée une structure de données appelée `inode` qui stocke, entre autres, un index des blocs composant le fichier. Dans cette structure, on trouve 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.

1.1.

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

1.2.

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

2. TABLE D'ALLOCATION DE FICHIERS

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.

2.1.

On suppose que chaque numéro de bloc est stocké sur 2 octets. Quelle est la taille de la FAT pour une partition de 1 Go avec des blocs de 512 octets ?

2.2.

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.

2.3.

Si la FAT est chargée en mémoire, quel est le nombre d'accès mémoire et le nombre d'E/S nécessaires pour lire l'octet 2000 du fichier F1 ?

3. ACCES AUX FICHIERS SOUS UNIX

3.1.

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.

3.2.

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
fd1 = open("fic1", O_RDONLY) ;            fd1 = open("fic1", O_RDONLY) ;
fd2=open("fic2",O_RDRW|O_APPEND);         read (fd1, 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.

3.3.

Le processus P2 continue son exécution en faisant un « fork » :

```
Processus P2
fd1 = open("fic1", O_RDONLY) ;
read (fd1, buf, 10) ;
if (fork()== 0) {
  read(fd1, buf, 2)
...
}
else wait(NULL)
```

Représentez la structure de données après le read

4. GESTION D'ESPACE LIBRE

On considère une partition de 1 Go. Les blocs 0 à 13 sont occupés, ainsi que tous les blocs de 20 à 23.

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

4.1.

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

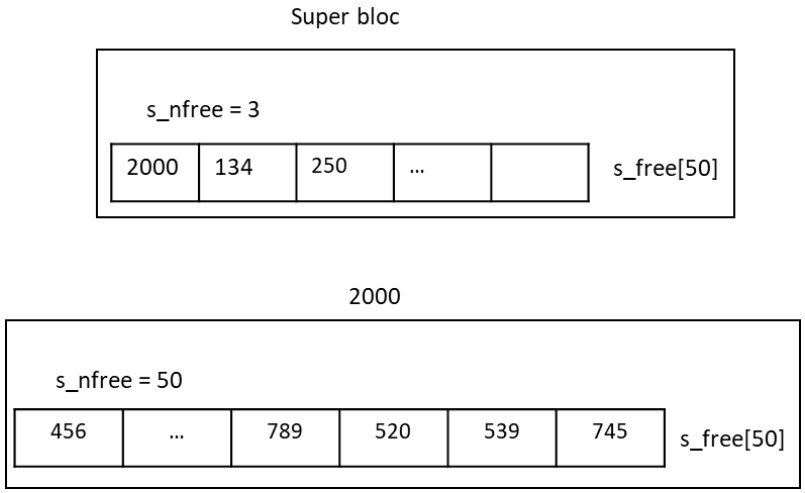
Une autre stratégie consiste à gérer l'espace libre sous forme groupée : un bloc spécial (appelé le superbloc) qui contient (entre autres) :

- Une table pour stocker des numéros de blocs libres (s_free[50]),
- Le nombre de blocs libres contenus dans le superbloc (s_nfree),
- s_free[0] désigne un bloc libre contenant lui-même une liste de blocs libres,

A chaque allocation, on choisit le dernier numéro de blocs libre de la table s_free du superbloc.

Lorsqu'on utilise le dernier numéro de bloc du superbloc (le bloc désigné par s_free[0]), on recopie dans le superbloc le contenu de s_free[0].

Soit la configuration initiale suivante du superbloc et du bloc 2000 :



4.2.

On alloue trois nouveaux blocs pour un fichier.

Quels sont les numéros de blocs alloués ?

4.3.

Donnez la nouvelle configuration du superbloc après ces trois allocations