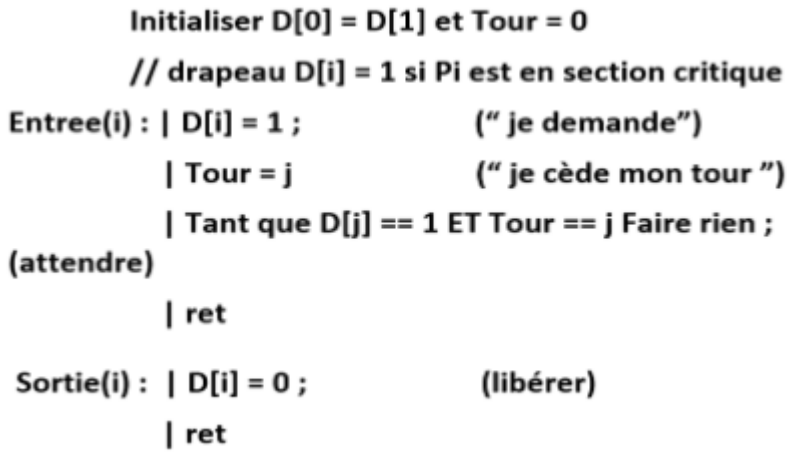
Exclusion mutuelle : Algo de Peterson



- **Vérification de l’algo** : "Tour" ne peut avoir qu'une valeur. Donc, même si les deux instructions ont lieu "en même temps", il n'y a qu'un seul des tests "Tant que D[j]==1 ET Tour==j" qui sera faux, l'autre laissant un processus bloqué.
- **Exclusion mutuelle** : Si les deux processus sont en section critique, alors (D[2]=0 ou Tour vaut 1) et (D[1]=0 ou Tour=2) et (D[1]=1 et D[2]=1) ce qui est impossible.
- **Absence de blocage** : Si les deux processus sont bloqués, alors (D[2]=1 et Tour =2) et (D[1]=1 et Tour =1), ce qui est également impossible.
- **Progression**: La solution étant symétrique, on peut supposer que P2 n'a pas demandé l'entrée en section critique, donc D[2]=0. Si P1 ne peut pas entrer en section critique, alors D[2]=1 et Tour=2, ce qui est une contradiction.
- Attente bornée
  - Si P1 et P2 ont demandé l'entrée en section critique, supposons que P1 ait pu y accéder. Donc D[1]=1 et D[2]=1, Tour =1 et P2 est en attente
  - Si P1 demande une nouvelle fois l'entrée en SC, après avoir exécuté sa section de sortie, il affecte à D[1] la valeur 1 et à Tour la valeur 2, ce qui lui interdit l'entrée en SC, il attend au plus un passage de l'autre en section critique pour y entrer à son tour

Segments de mémoire partagée (SMP)

- Création avec la commande `int shmget(key_t cle, int taille, int flag);` où :
  - clé = `IPC_PRIVATE` car on ne souhaite pas acquérir l'identificateur
  - taille donne le nombre d'octets du segment, ici = `sizeof(int)` car on veut qu'il puisse contenir un entier E
  - flag = `IPC_CREAT | 0666` pour créer un segment et donner tous les droits d'accès. On aurait mis uniquement *0666* pour l'acquisition
  - retourne la clé ou -1 en cas d'échec
- Attacher le SMP à l'espace virtuel du processus avec `char* shmat(int shmid, char* shmaddr, int shmflag) ;`
  - si shmaddr = 0, alors le système choisit l'emplacement dans l'espace virtuel
  - si shmaddr ≠ 0
    - si shmflag=0, le segment est attaché à l'adresse shmaddr
    - si shmflag=SHM_RND (round) le segment est attaché à l'adresse spécifiée dans shmaddr, arrondie vers le bas par la constante (SHMLBA)
    - si shmflag contient SHM_RDONLY le segment est attaché en lecture seule, sinon en lecture + écriture retourne un pointeur _char*_ vers l'addresse shmaddr
- Après utilisation du SMP, il faut faire un détachement avec la primitive `int shmdt(char* adr);`. Ensuite, il faut détruire le SMP avec l'appel à la primitive `int shmctl(int shmid, int cmde, struct shmid_ds * buf) ;` où *cmde* vaut `IPC_RMID`. Par exemple : `shmctl(shmid, IPC_RMID, 0) ;` ou `shmctl(shmid, IPC_RMID, NULL) ;`

▼ TD6 Exercice 1 - Création d'une bibliothèque P(), V()

```
#include "sem_pv.h"

static int semid = -1;
struct sembuf op_P = {-1,-1,0},
```

```
op_V = {
    -1, //sem_num : numero du semaphore cible de l'opération
    1,  //sem_op : si > 0, alors V(n), si < 0 alors P(n), sinon Z (processus bloqué)
    0   //sem_flg option : IPC_NOWAIT, SEM_UNDO; initialisé à 0 car pas utile ici
};


union semun {
    int val;                // Valeur pour SETVAL
    struct semid_ds *buf;   // Tampon pour IPC_STAT, IPC_SET
    unsigned short *array;  // Tableau pour GETALL, SETALL
    struct seminfo *__buf;  // Tampon pour IPC_INFO (spécifique à Linux)
};

int init_semaphore() {
    union semun arg0; //un union va prendre un type différent en
                      //fonction de la variable utilisé.
                      //arg0 est le nom de la variable de type union semun
    arg0.val = 0; //  semun représente un int car val est un int
    if(semid != -1) {
        fprintf(stderr,"fonction deja appelee");
        return 0;
    }
    semid = semget(IPC_PRIVATE, N_SEM, 0666);
    //IPC_PRIVATE permet d'avoir un IPC sans clé
    //N_SEM correspond au nombre de sémaphores
    //0666 flag qui correspond aux droits d'accès
    if(semid == -1) {
        fprintf(stderr,"Echec de creation");
        return -2;
    }
    for(int i=0; i < N_SEM; i++) {
        semctl(semid, i, SETVAL, arg0); //on initialise touq les sémaphores
                                        //du groupe semid
    }
}

int detruire_semaphore() {
    if(semid == -1) {
        fprintf(stderr, "semaphore non cree");
        return -1;
    }
    int ret = semctl(semid, 0, IPC_RMID, 0); //les deux '0' servent à remplir
                                             //des champs dont on n'a pas besoin
    semid = -1;
    return ret;
}

int val_sem(int sem, int val){
    if(semid == -1) {
        fprintf(stderr, "Le semaphore n'existe pas");
        return -1;
    }
    union semun semun_val;
    semun_val.val = val;
    int ret = semctl(semid, sem, SETVAL, semun_val);
    return ret;
}
```

```
int P(int sem) {
    if(semid == -1) {
        fprintf(stderr, "Les semaphores n'existent pas");
        return -1;
    }
    if(sem<0 || sem>=N_SEM) {
        fprintf(stderr, "Numero de semaphore incorrect\n");
        return(-2);
    }
    op_P.sem_num=sem;
    return(semop(semid, &op_P, 1));
}

int V(int sem) {
    if(semid == -1) {
        fprintf(stderr, "Les semaphores n'existent pas");
        return(-1);
    }
    if(sem<0 || sem>=N_SEM) {
        fprintf(stderr, "Numero de semaphore incorrect\n");
        return(-2);
    }
    op_V.sem_num=sem;
    return(semop(semid, &op_V, 1));
}
```

▼ TD6 : Exercice 2 - Section critique

```
#include "sem_pv.h"
#include <time.h>
#include <sys/wait.h>

int main()
{
  init_semaphore(); // on créé un groupe de sémahpores
  val_sem(0, 1);    // on met la valeur 1 dans le sémaphore 0 du groupe

  // 1. Creation d'un segment de memoire partagé
  int shmid = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | 0666);
  printf("shmid : %d\n", shmid);
  if (shmid == -1) {
    fprintf(stderr, "Echec de creation du segment de memeoire partage");
    return -1;
  }

  // 2. Initialisation à 0 de l'entier E en mémoire partagée.
  int* shmaddr = (int*)shmat(shmid, NULL, 0); // on attache le SMP à
                                              //l'espace virtuel du processus
  if (shmaddr == (void*)-1) {
    fprintf(stderr, "Echec du shmat");
    return -1;
  }
```

```
  *shmaddr = 0; // on initialise à 0 l'entier E en mémoire partagée

  // 3. Création d'un process "fils" partageant le segment
  //précédemment créé avec son "père"
  pid_t pid = fork();
  if (pid == -1) {
    fprintf(stderr, "Echec de la création du processus fils");
    return -1;
  }

  srand(time(NULL));
  for (int i = 0; i < 100; ++i)
  {
    P(0);   // le processus ppredn l'accès à la section critique
            // pour le sémpahore 0
    // 1.
    int A = *shmaddr;   // on met la val E du semaphore dans une var A

    // 2. On attend entre 20 et 100 ms
    usleep((20+(rand()%80)));   // rand()%80 génère un nombre entre 0 et 79
                                // 20+(rand()%80) génère un nombre entre 20 et 99
    // 3. Incrémenter A
    ++A;

    // 4. On affecte A à E
    *shmaddr = A;

    V(0); // On libère la section critique pour le sémaphore 0

    // 5. On attend entre 20 et 100 ms
    usleep((20+(rand()%80)));

    // 6. Affichage dans le processus père de la valeur de E.
    if (pid > 0) // si on est dans le père
      printf("%d\n", *shmaddr);
  }

  if (pid > 0)  // si on est dans le père
  {
    wait(NULL); //on attend la fin des processus fils
    detruire_semaphore();   //on détruit le groupe de sémaphores
  }
}
```

▼ Notes final blanc

- Les sémaphores, les verrous et les variables de condition prennent en charge deux opérations distinctes : Synchronisation et notification d'événements

Processus fils/père

- `fork()` crée un nouveau processus (le fils) par duplication du processus appelant (le père) : le fils aura le même espace adressable que son père. Afin de distinguer le père du fils, la fonction retourne :
  - 0 au processus fils
  - l'identifiant (ou pid) du processus fils créé au processus père
  - 1 en cas d'erreur
- Processus **zombie** (!= processus orphelin) :
  - Désigne un processus qui s'est achevé, mais qui dispose toujours d'un identifiant de processus (PID) et reste donc encore visible dans la table des processus.
  - Au moment de la terminaison d'un processus, le système désalloue les ressources que possède encore le processus mais ne détruit pas son bloc de contrôle.
  - Au moment de la terminaison d'un processus, le système désalloue les ressources que possède encore le processus mais ne détruit pas son bloc de contrôle
  - La commande `ps` affiche Z comme état d'un processus Zombie et 0 pour sa taille
- `pid_t wait(int* etat)` permet au père de récupérer la terminaison des ses fils et donc d'éviter les processus zombi
  - si le père possède au moins un fils zombi, alors la primitive renvoie le *pid* de l'un d'eux et la valeur `etat` fournit des infos sur la terminaison de ce processus
  - si aucun des processus fils n'est zombi, alors le processus père est bloqué jusqu'à ce qu'un fils devienne zombi
  - si le père n'a pas de fils, `wait()` retourne -1
  - états retournés :
    - `WIFEXITED (etat)` : renvoie vrai si le statut provient d'un processus fils qui s'est terminé normalement ;

        - `WEXITSTATUS (etat)` : (si `WIFEXITED (etat)` renvoie vrai) renvoie le code de retour du processus fils passé à _exit() ou exit() ou la valeur retournée par la fonction main() ;
        - `WIFSIGNALED (etat)` : renvoie vrai si le statut provient d'un processus fils qui s'est terminé à cause de la réception d'un signal ;
        - `WTERMSIG (etat)` : (si `WIFSIGNALED (etat)` renvoie vrai) renvoie la valeur du signal qui a provoqué la terminaison du processus fils.
    - si l'on ne souhaite pas récupérer l'état, on peut utiliser `wait(NULL);`
- `pid_t waitpid(pid_t pid, int* etat, int option);` permet de tester la terminaison du processus qui a le pid passé en paramètre
    - paramètre *pid* :
        - < -1 : Tout processus fils dans le groupe |pid|
        - 1 : Tout processus fils
        - 0 : Tout processus fils du même groupe que l'appelant
        - 0 : Le processus fils d'identité *pid*

▼ Final Blanc

```
main(int argc, char ** argv) {
  int child = fork();
  int c = 5;

  if(child == 0) {
    c += 5;
  }
  else {
    child = fork();
    c += 10;
    if(child) c += 5;
  }
}
// Combien de copies différentes de la variable c existe-t-il ? => 3
```

Interblocage

Exemple :

P1	P2
1. Demander(Périph1)	1. Demander(Périph2)
2. Demander(Périph2)	2. Demander(Périph1)
<Traitement>	<Traitement>
3. Liberer(Périph1)	3. Liberer(Périph2)
4. Liberer(Périph2)	4. Liberer(Périph1)

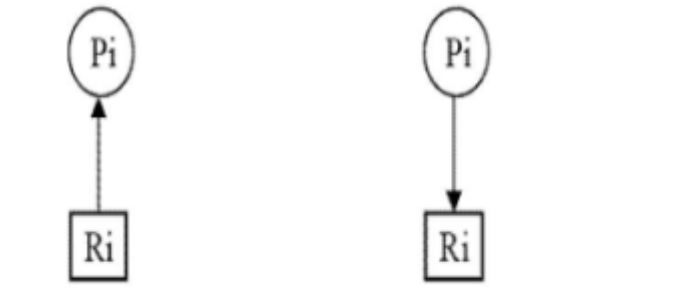
Exemple d'interblocage

Si P1 d'abord puis P2 ou P2 puis P1 alors pas d'interblocage. Mais s'ils s'exécutent en parallèle lors interblocage (chacun souhaite accéder à une ressource déjà utilisée par l'autre processus).

▼ Conditions nécessaires à l'interblocage : (conditions doivent être en simultané)

- Exclusion mutuelle : Une ressource au moins doit se trouver dans un mode non partageable et nécessite une EM pour son utilisation.
- Occupation et attente : Il peut exister un processus occupant au moins une ressource et qui attend d'acquérir des ressources détenues par d'autres processus (allocation partielle)
- Pas de réquisition : Les ressources déjà détenues ne peuvent être retirées de force à un processus. Elles doivent être explicitement libérées par le processus qui les détient.
- Attente circulaire

▼ Modélisation : ressources avec un carré et processus avec un cercle :

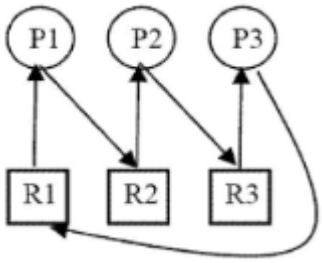


Pi détient la ressource Ri Pi demande la ressource Ri

Représentation de l'allocation de ressources

Théorème : Si chaque type de ressource possède exactement un seul exemplaire alors : Il y a
situation d'interblocage si et seulement si le graphe d'allocation possède un circuit.

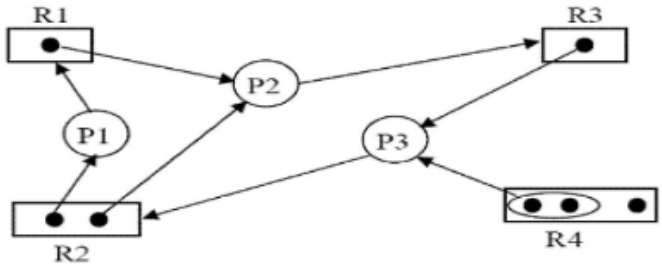
Exemple :



Théorème : Dans le cas de plusieurs exemplaires par type de ressource : Si le graphe d'allocation est sans circuit alors aucun processus n'est dans une situation d'inter-blocage.

Cycle1 : R2, P2, R3, P3, R2
==> **interblocage**

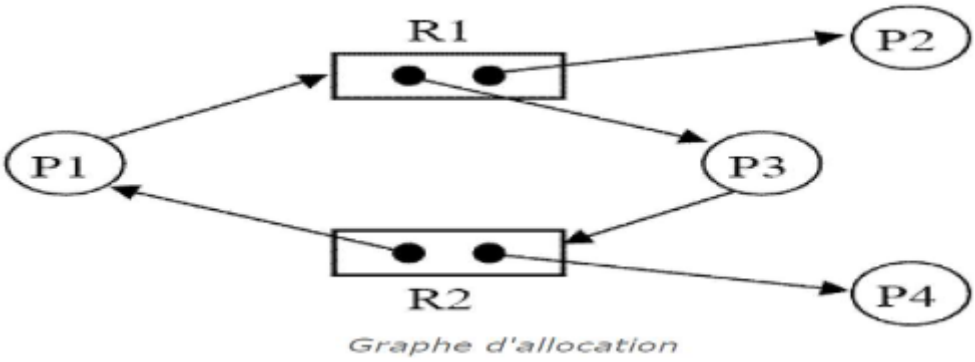
Cycle2 : R2, P1,R1,P2,R3,P3,R2
==> **interblocage.**



Représentation des exemplaires d'une ressource

▼ Allocation avec des matrices

Exemple avec les matrices :



Graphe d'allocation

DISPONIBLE[]

R1	R2
0	0

DEMANDE[], D={ P1, P3 }

	R1	R2
P1	1	0
P2	0	0
P3	0	1
P4	0	0

ALLOCATION[]

	R1	R2
P1	0	1
P2	1	0
P3	1	0
P4	0	1

Formule : Demande[i] ≤ DISPONIBLE + Σ Allocation[j]

⇔ DEMANDE[1] = (1, 0) ≤ (0,0) + (1,0) + (0,1)

Formule : Demande[i] ≤ DISPONIBLE + Σ Allocation[j] ⇔ DEMANDE[1] = (1, 0) ≤ (0,0) + (1,0) + (0,1)

▼ Eviter les interblocage

- Séquence Saine : Une séquence P1, P2, ..., Pn est saine si pour chaque processus Pi, les demandes de ressources de Pi peuvent être satisfaite par les ressources disponibles plus les ressources détenues par tous les Pj avec i>j.
- Algorithme du banquier : évaluer le risque d'interblocage pouvant être provoqué par une demande de ressource ⇒ Si une demande présente ce risque, le système doit la mettre en attente même si elle peut être satisfaite avec les ressources dispo
    - Algo de détermination d'état sain :

Travail : Tableau [1..*M*] d'entiers ;
Fini : Tableau [1.. *N*| de Booléen ;
1. *Travail* := DISPONIBLE ;
   Pour *i* = 1 jusqu'à *N* faire *Fini [i]* = faux ;
2. Trouver *i* tel que *Fini [i]* = faux et *BESOINi* <= *Travail*
   Si *i* n'existe pas aller à 4 ;
3. *Travail* = *Travail* + *Allocationi* ;
   Fini [i] = vrai ;
   Aller à 2 ;
4. Si *Fini [i]* = vrai pour tout *i* alors le système est dans un état sain ;

- Algo de requête :

1. Si *DEMANDEi* ≤ *BESOINi* Alors Aller à 2
   Sinon *Erreur* : excéder sa demande maximale
2. Si *DEMANDEi* ≤ *DISPONIBLE* Alors Aller à 3
   Sinon *Attente*
3. /* On suppose que le système a alloué les ressources demandées par le processus Pi */
   DISPONIBLE = *DISPONIBLE* - *DEMANDEi*
   ALLOCATIONi = *ALLOCATIONi* + *DEMANDEi*
   BESOINi = *BESOINi* - *DEMANDEi*
4. Si cet état est sain Alors allocation avec succès
   Sinon /* Restaurer l'état initial */
   DISPONIBLE = *DISPONIBLE* + *DEMANDEi*
   ALLOCATIONi = *ALLOCATIONi* - *DEMANDEi*
   BESOINi = *BESOINi* + *DEMANDEi*

▼ TD10 EX1

Soit un système qui ne contient qu'une seule ressource de chaque type. L'état du système est comme suivi :

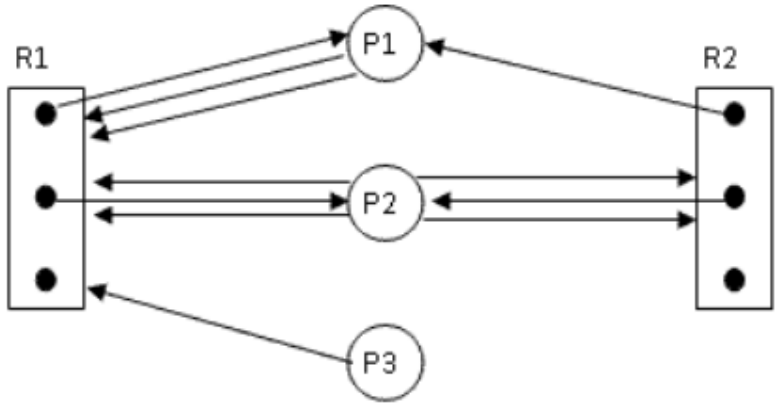
- Le processus A détient la ressource R et veut la ressource S.
- Le processus B ne détient aucune ressource mais veut la ressource T.
- Le processus C ne détient aucune ressource mais veut la ressource S.
- Le processus D détient la ressource U et veut les ressources S et T
- Le processus E détient la ressource T et veut la ressource V.
- Le processus F détient la ressource W et veut la ressource S.
- Le processus G détient la ressource V et veut la ressource U.

- Ce système est-il en situation d'interblocage ?

⇒ On construit le graphe ⇒ On trouve un circuit et les ressources ne sont disponibles qu'en un seul exemplaire ⇒ Interblocage

▼ TD10 EX2

L'état d'un système est représenté par le graphe suivant :



Demande de ressources

- Donner la représentation par matrice
- Y a t-il interblocage ?

DISPONIBLE[] :

R1	R2
1	1

ALLOCATION[] :

	R1	R2
P1	1	1

P2	1	1
P3	0	0

DEMANDE [] :

	R1	R2
P1	2	0
P2	2	2
P3	1	0

D = { P1, P2 } ⇒ Car ils forment le circuit

Note : (x, y) < (w, z) ssi x<w ET y<z. Si on ne peux pas vérifier pour aucun processus (exemple : (2,0) < (1,1)), alors il y a interblocage

DEMANDE[i] ≤ DISPONIBLE[] + ALLOCATION[P3] car P3 ∉ D

DEMANDE[1] = (2,0) ≤ (1, 1) + (0, 0) ⇒ FAUSSE

DEMANDE[2] = (2,2) ≤ (1,1) + (0,0) ⇒ FAUSSE

⇒ Interblocage

▼ TD10 EX4

L'état des allocations de ressources d'un système est donné par la représentation suivante :

	ALLOCATION					DEMANDE					DISPONIBLE			
	R0	R1	R2	R3		R0	R1	R2	R3	R0	R1	R2	R3
P0	1	0	0	0	P0	0	0	1	0	0	0	0	0
P1	0	0	0	0	P1	1	1	1	0				
P2	0	0	1	1	P2	0	1	0	0				
P3	0	0	0	0	P3	0	1	0	1				
P4	0	1	0	0	P4	0	0	0	1				

Matrice d'allocation et de demande

- Représenter le graphe d'allocation.
- Représenter le graphe des attentes

Voir notes papier

⇒ Le graphe d’attente montre qu’il y a un interblocage

Gestion de la mémoire

💡 Deux grandes catégories de mémoire :

- Mémoire centrale (ou mémoire interne)
- Mémoire de masse : stockage magnétique (disques durs), optique (CD-ROM, DVD-ROM) et mémoire mortes

▼ Propriétés de la mémoire :

- mode de localisation et d'accès à l'info / la mémoire
  - Accès séquentiel (e. g. bande magnétique) : le + lent de tous
  - Accès direct ou aléatoire (e. g. mémoire centrale, registres) : chaque information a une adresse
  - Accès semi-séquentiel
  - Accès par le contenu (e. g. mémoire cache)
- capacité du volume
- temps d'accès
- le temps de cycle
- le débit
- la volatilité

▼ Espace d'adressage logique ou physique

- L'unité centrale manipule des adresses logiques (emplacement relatif). L'espace d'adressage logique (virtuel) est un ensemble d'adresses pouvant être générées par un programme
- L'unité mémoire manipule des adresses physique (emplacement mémoire)
  Les adresses physiques ne sont jamais vues par les programmes utilisateurs
  L'espace d'adressage physique est un ensemble d'adresses physiques correspondant à un espace d'adresses logiques

▼ Allocation de partitions contiguës

▼ Partitions contiguës fixes

- Mémoire partagée en n partitions de tailles fixes (inégales) au démarrage du système.
- Le syst. d'exploitation maintient une table de description des partitions

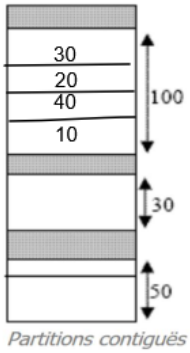
Adresse	Taille	Etat
312k	8K	allouée
320k	32K	allouée
352k	32K	libre
384k	120K	libre
504k	520K	allouée

- L'allocation de la mémoire physique est simple, puisqu'un processus reçoit une portion de mémoire physique de la même taille que son espace d'adressage
- Lorsqu'une partition est libre, le système recherche un processus dans la file d'attente qui a le plus gros besoin d'espace tout en étant inférieur à la taille de la partition ⇒ Peut entraîner la famine d'un petit processus
- Possibilité d'une file d'attente commune à toutes les partitions ou de files d'attentes séparées pour chaque partition.

▼ Partition contiguë dynamique

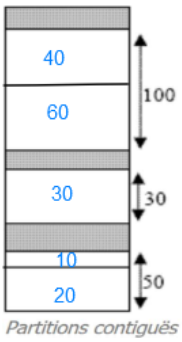
- Mémoire partitionnée dynamiquement selon la demande, chaque processus est alloué exactement la taille de mémoire requise

- Stratégies d'allocation :
  - **First Fit** (le premier qui convient)
    Exemple : Soit une mémoire allouée à la demande par zones de tailles variables. Cette mémoire contient 3 zones libres de tailles 100, 30 et 50 chainées par ordre croissant des adresses. Traiter les suites de demande d'allocation suivantes : 30, 20, 40, 60, 10



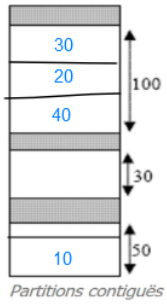
Partitions contiguës

  - **Best Fit** (meilleur qui convient) : On alloue la plus petite partition dont la taille est au moins égale à celle du processus en attente.
    Exemple :



Partitions contiguës

  - **Worst Fit** (pire qui convient) : On alloue au processus la partition de plus grande taille



Partitions contiguës

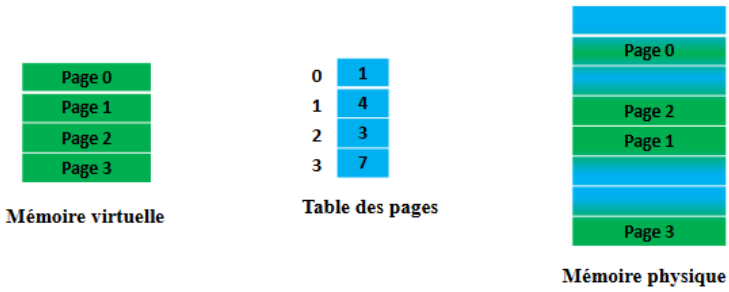
▼ Partitions contiguës siamoises (Buddy System)

Au départ	1M								
A=70K	A		128			256			512 K
B=35K	A	B		64		256			512
C=80K	A	B		64	C		128		512
A se termine	128	B		64	C		128		512
D=60K	128	B		D	C		128		512
B se termine	128	64		D	C		128		512
D se termine	256				C		128		512
C se termine	512								512
Fin	1024K								

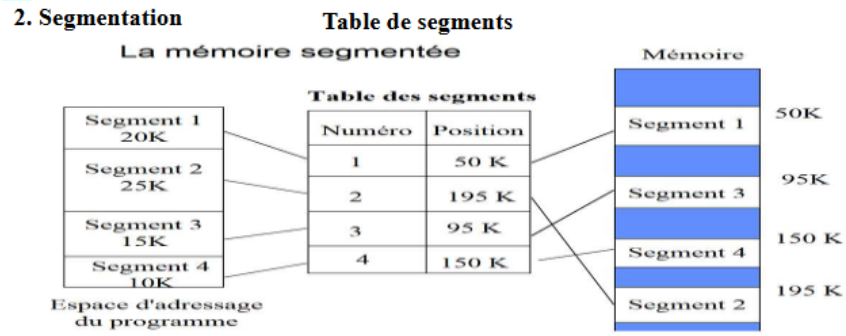
▼ Allocation de partition non contiguës

▼ Pagination

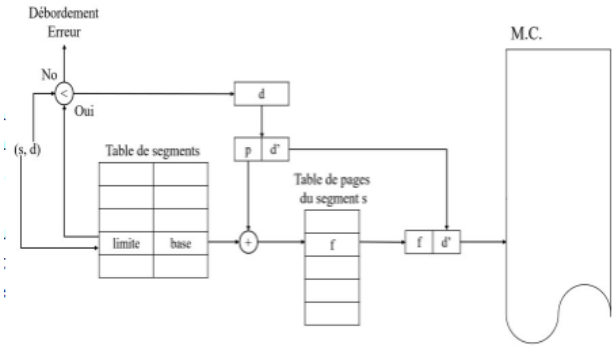
- La solution apportée par les mécanismes de pagination consiste à découper l'espace adressable, ou espace virtuel, en zones de taille fixe appelée pages.
- La mémoire réelle est également découpée en cases ayant la taille d'une page de sorte que chaque page peut être implantée dans n'importe quelle case de
  la mémoire réelle
- Une table de correspondance (une par processus), appelée table des pages est gérée par le système d'exploitation. En général, l'adresse de cette table est une adresse physique.
- Une adresse est divisée en 2 parties : un numéro de page p et un déplacement à l'intérieur de la page d. La taille de la page (et donc de la case) est une puissance de 2 entre 512 et 8192 octets
- Traduction d'adresses virtuelles en adresses réelles



▼ Segmentation

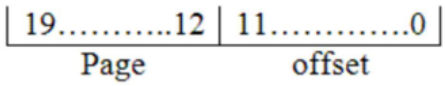


▼ Segmentation paginée



▼ Exercice 2 TD7 (pagination et adressage)

Supposons qu'un adresse, en mémoire virtuelle paginée, nécessite 20 bits organisés comme suit :



Adresse paginée

Quelle est la taille de cette mémoire virtuelle exprimée en nombre de mots et de pages ?

⇒ Le nombre de bits de l'offset détermine le nombre max d'adresses différentes qu'un seul mot peut référencer. Dans ce cas, avec 12 bits, nous avons 2^12 = 4096 adresses ≠ pour chaque page.

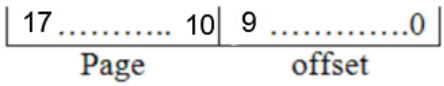
⇒ Le nombre de bits pour la page détermine le nombre maximum de pages différentes que nous pouvons avoir. Avec 8 bits, nous avons 2^8 = 256 pages différentes.

⇒ Taille de la mémoire virtuelle en mots = 2^12 adresses par page * 2^8 pages = 2^(8+12) = 2^20 mots.

▼ Exercice 3 TD7 (Traduction d'adresse logique)

Sur un système de pagination simple de 2^14 octets de mémoire physique, 256 pages d'espace d'adressage logique et une taille de page de 2^10 octets

1. Combien de bits se trouvent dans une adresse logique ?
   ⇒ Étant donné qu'il y a 256 pages d'espace d'adressage logique, cela signifie que nous avons besoin de 8 bits pour représenter chaque page (2^8 = 256)
   ⇒ Chaque page a une taille de 2^10 = 1024 octets, cela nécessite 10 bits pour représenter l'offset dans chaque page.
   ⇒ La taille totale de l'adresse logique est de 8 bits (pour la page) + 10 bits (pour l'offset) = 18 bits



Adresse paginée

2. Quelle est la taille d'une case ?
   ⇒ Dans un système de pagination simple, chaque case dans la mémoire physique correspond à une taille de page. Dans ce cas, la taille d'une case est de 2^10 octets
3. Combien de bits de l'adresse physique spécifient la case ?
   ⇒ Étant donné que la mémoire physique est de 2^14 octets, cela signifie que nous avons besoin de 14 bits pour représenter chaque adresse physique. Cependant, dans un système de pagination simple, les bits de poids faible de l'adresse physique correspondent à l'offset dans la page et ne spécifient pas la case. Par conséquent, la partie des bits de l'adresse physique qui spécifie la case est égale à la taille de la mémoire physique moins le nombre de bits nécessaires pour représenter l'offset.
   ⇒ Donc, la taille des bits de l'adresse physique spécifiant la case est de 14 bits (pour l'adresse physique) - 10 bits (pour l'offset) = 4 bits.

4. Combien d'entrées se trouvent dans la table de pages ?
   ⇒ Le nombre d'entrées dans la table de pages est égal au nombre total de pages d'espace d'adressage logique. Dans ce cas, il y a 256 pages, donc il y a également 256 entrées dans la table de pages.
5. Quelle est la largeur de la table de pages (taille d'une entrée) ?
   ⇒ La largeur de la table de pages dépend de la quantité d'informations stockées dans chaque entrée. Dans ce cas, chaque entrée de la table de pages doit stocker l'adresse physique de la page correspondante. Comme nous avons déterminé précédemment que nous avons besoin de 14 bits pour représenter chaque adresse physique, la largeur de la table de pages (taille d'une entrée) serait de 14 bits pour stocker l'adresse physique de chaque page.

▼ Pagination à la demande - mémoire virtuelle

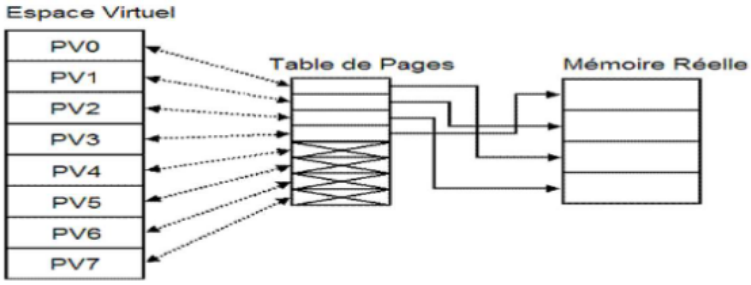
Lorsqu'un processus accède à une page qui n'est pas en mémoire physique, l'instruction est interrompue, un déroutement au système est exécutée et le processeur est restitué au SE (Syst. d'exploitation). On dit qu'il y a défaut de page.

Le défaut de page est un déroutement qui oblige le processeur à suspendre l'exécution du programme en cours pour lancer une entrée/sortie qui charge la page manquante en mémoire centrale dans une case libre. Le déroutement d'un défaut de page est réalisé de la manière suivante :

1. Le SE sélectionne une case peu utilisée (ou vide) en mémoire centrale et écrit son contenu sur le disque (mémoire secondaire) ⇒ cf Algorithme de remplacement.
2. Il transfère ensuite la page qui vient d'être référencée dans la case libérée et modifie la correspondance.
3. Il recommence l'instruction déroutée

Cette méthode qui consiste à charger une page uniquement lorsqu'elle est demandée s'appelle pagination à la demande (demand paging).

L'unité de gestion mémoire (MMU, Memory Management Unit) fait correspondre les adresses virtuelles à des adresses physiques, en se basant sur une table de page.



Puisqu'il n'y que 4 cases physiques, seuls les 4 des pages virtuelles de la figure sont mises en correspondance avec la mémoire physique. Les autres, illustrées par des croix dans la figure, ne sont pas mises en correspondance.

En terme de matériel, un bit de présence/absence (1: page en mémoire (RAM), 0: sur disque) conserve la trace des pages qui se trouvent physiquement en mémoire.

On appelle chaîne de références la séquence des numéros de pages référencées par un programme durant une exécution. Un bon algorithme de remplacement doit diminuer le nombre de défauts de pages engendrés par une chaîne de référence définie par l'exécution d'un processus.

▼ Algorithmes de remplacement de page

- Algorithme optimal :

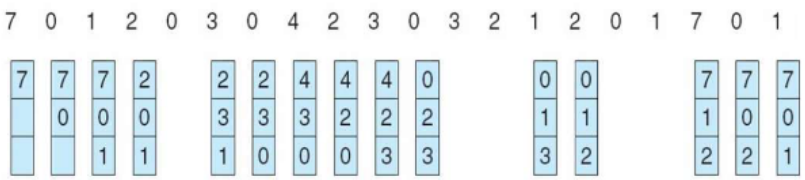
  Lors d'un défaut de page, choisir comme victime une page qui ne fera l'objet d'aucune référence ultérieure, ou, à défaut, la page qui fera l'objet de la référence la plus tardive.

  Cet algorithme suppose une connaissance de l'ensemble de la chaîne de référence; il est donc irréalisable en temps réel. Il permet d'évaluer, par comparaison, les autres algorithmes

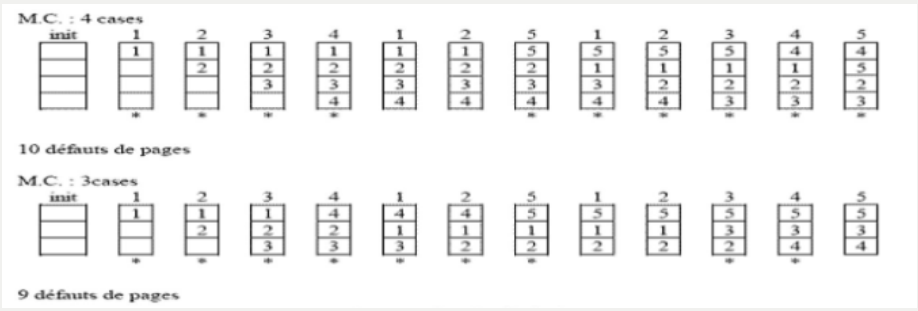
- Tirage aléatoire : La victime est choisie au hasard (loi uniforme) parmi l'ensemble des pages présentes en mémoire

- Ordre Chronologique de Chargement (FIFO) : Choisit comme victime la page la plus anciennement chargée.
  ⇒ Entretenir dans une file FIFO les numéros des cases où sont chargées les pages successives
  ⇒ Performances pas toujours bonnes

  Exemple : On considère notre séquence de référence 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 et on exécute l'algo de remplacement FIFO avec 3 cadres de pages :



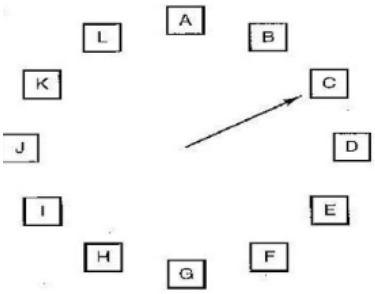
⚠ Anomalie de Belady : ce n'est pas parce qu'on a plus de cases qu'on aura moins de défauts !



- Ordre Chronologique d'utilisation (LRU : Least Recently Used) : puisque les pages récemment utilisées ont une probabilité plus élevée que d'autres d'être utilisées dans un futur proche, une page non utilisée depuis un temps élevé a une probabilité faible d'être utilisée prochainement.
  ⇒ L'algorithme choisit donc comme victime la page ayant fait l'objet de la référence la plus ancienne. Pour cela, une information doit être associée à chaque case et mise à jour à chaque référence.
- L'algorithme NRU (Not Recently Used) : consiste à éliminer une page qui n'a pas été utilisée récemment. Pour cela, il repose sur le bit R mis à jour par le matériel. Régulièrement, le bit est mis a 0 sur toutes les pages actives. Lorsqu'une page doit être libérée, une page ayant encore son bit a 0 est choisie (et si aucune n'est disponible, une page ayant son bit a 1 est alors choisie).

  On distingue alors 4 classes de pages (par ordre de priorité) :

  1. Non référencées, non modifiées (choisies en priorité 1 car ne nécessite pas d'écriture sur le disque)
  2. Non référencées, modifiées
  3. Référencées, non modifiées
  4. Référencées, modifiées
- Algorithme de la seconde chance : modification de l'algo FIFO pour éviter la suppression d'une page à laquelle on a régulièrement recours en inspectant le bit R de la page la plus ancienne. S'il est à 0, la page est à la fois ancienne et inutilisée, elle est donc remplacée immédiatement. Si le bit R est à 1, le bit est effacé, la page est placée à la fin de la liste des pages, et son temps de chargement est mis à jour comme si elle venait d'arriver en mémoire. La recherche continue alors.
  ⇒ Manque d'efficacité car il déplace constamment des pages dans sa liste
- Algorithme de l'horloge : consiste à garder tous les cadres de pages dans une liste circulaire formant une horloge, comme l'illustre la figure ci-dessous :



Quand un défaut de page se produit, la page pointée est testée. L'action entreprise dépend de la valeur du bit R :
R=0: retirer la page
R=1: mettre R à 0 et avancer le pointeur

On appelle Ecroulement une haute activité de pagination. Un processus s'écroule lorsqu'il passe plus de temps à paginer qu'à s'exécuter

▼ Exercice 4 TD7 (Pagination à la demande)

Considérons la séquence de références mémoires suivante d'un programme de 460 mots : 10, 11, 104, 170, 73, 309, 185, 245, 246, 434, 408, 364.

1. Donner la chaîne de références correspondante si on suppose que la taille d'une page est de 100 mots
   ⇒ taille d'une page : 100 mots → page 0 : mots 1 à 99 ; page 1 : mots 100 à 199…
   ⇒ chaîne de références : 0, 0, 1, 1, 0, 3, 1, 2, 2, 4, 4, 3
2. Calculer le nombre de défauts de pages générés pour une mémoire centrale de 200 mots en utilisant les algorithmes de remplacement FIFO, LRU.
   ⇒ 200 mots = 2 cases de 100 mots

   FIFO : 6 défauts

chaîne	0	0	1	1	0	3	1	2	2	4	4	3
case 1	0	0	0	0	0	3	3	3	3	4	4	4
case 2			1	1	1	1	1	2	2	2	2	3
défaut	*		*			*		*		*		*

   LRU : 7 défauts

chaîne	0	0	1	1	0	3	1	2	2	4	4	3
case 1	0	0	0	0	0	0	1	1	1	4	4	4
case 2			1	1	1	3	3	2	2	2	2	3
défaut	*		*			*	*	*		*		*

3. On suppose maintenant que la taille d'une page est de 200 mots et la mémoire principale est de 400 mots. Refaire les questions précédentes. Que peut-on en déduire ?

   ⇒ page 0 : mots 1 à 199 ; page 1 : mots 200 à 399 …

   ⇒ 400 mots = 2 cases de 200 mots

   ⇒ chaîne de références : 0, 0, 0, 0, 0, 1, 0, 1, 1, 2, 2, 1

   FIFO : 3 défauts

chaîne	0	0	0	0	0	1	0	1	1	2	2	1
case 1	0	0	0	0	0	0	0	0	0	2	2	2
case 2						1	1	1	1	1	1	1
défaut	*					*				*		

LRU : 3 défauts

chaîne	0	0	0	0	0	1	0	1	1	2	2	1
case 1	0	0	0	0	0	0	0	0	0	2	2	2
case 2						1	1	1	1	1	1	1
défaut	*					*				*		

▼ Exercice 5 TD7 (Placement mémoire et performances)

Considérons la matrice suivante : *char A[10][100]* ;
On suppose que la matrice sera rangée en mémoire centrale ligne par ligne.
On considère un système paginé, avec une taille de page de 200 octets, et une mémoire physique de 600 octets.
On suppose que le programme qui manipule la matrice est chargé dans une mémoire cache et son référencement ne provoque aucun défaut de page. Seul le référencement de la matrice est géré par pagination à la demande.

⇒ 10 lignes de 100 colonnes
⇒ 600 octets = 3 cases de 200 octets
⇒ On a donc 5 (=10*100/200) pages
⇒ Page 0 : de A[0][0] à A [1][99] (=200 cases de la matrice)

Algo 1 :

```
for i:= 0 to 9 do
    for j:= 0 to 99 do
        A[i][j] := 0;
```

Algo 2 :

```
for j:= 0 to 99 do
    for i:= 0 to 9 do
        A[i][j] := 0;
```

1. Donner la chaîne de référence induite par l'exécution de l'algo 1 :
   ⇒ Chaque référence est répétée 200 fois : 0, 1, 2, 3, 4
2. Donner la chaîne de référence induite par l'exécution de l'algo 2 :
   ⇒ Chaque référence est répétée 100 fois : 0, 1, 2, 3, 4
3. Calculer le nombre de défauts de page générés par l'exécution de chacune des deux boucles d'initialisation ci-dessus en utilisant l'algorithme de remplacement de page :

   ⇒ OPTIMAL : algo 1 : 5 défauts et algo 2 : 252 défauts

   ⇒ LRU : algo 1 : 5 défauts et algo 2 : 500 défauts

   ⇒ Après les 4 premiers défauts, on remarque un pattern de 2 défauts toutes les 4 références (pattern valide sur les 496 (500 - 4) références restantes)

   ⇒ 4 + 496 / 2 = 252 défauts

▼ Exercice 6 TD7 (Partitions siamoises)

Sur un système de 1Mo de mémoire et qui utilise le système de zones siamoises.

- Elaborer un diagramme d'allocations de la mémoire après chacun des événements suivants :

Au départ	**1M**								
A=70K	**A**		**128**			**256**			**512 K**
B=35K	**A**	**B**	**64**			**256**			**512**
C=80K	**A**	**B**	**64**	**C**	**128**				**512**
A se termine	**128**	**B**	**64**	**C**	**128**				**512**
D=60K	**128**	**B**	**D**	**C**	**128**				**512**
B se termine	**128**	**64**	**D**	**C**	**128**				**512**
D se termine	**256**			**C**	**128**				**512**
C se termine	**512**								**512**
Fin	**1024K**								

▼ Ordonnancement de processus

Ordonnanceur = partie du SE qui se base sur une stratégie pour choisir le prochain programme à exécuter. ⇒ “L’ordonnanceur (scheduleur) est un module du SE qui attribue le contr$ole du CPU à tour de rôle aux différents processus en compétition suivant une politique définie à l’avance par les concepteurs du système”

⇒ Il faut éviter le problème de famine, que la répartition des ressources soit équitable

Deux startégies d’ordonnancement :

▼ sans réquisition du CPU (startégie non préemptive)

- FCFS : First Come First Served (FIFO)

  ⇒ Les tâches sont ordonnancées dans l'ordre où elles sont reçues.

  ⇒ Le processus qui sollicite le CPU le premier sera servi le premier. On utilise une structure de file

  ⇒ Exemple : Temps de traitement moyen = 2.6

PID	Heure d'arrivée	Durée d'exécution	Début d'exécution	Heure de fin d'exécution	Temps de traitement
1	10	2	10	12	2
2	10,1	1	12	13	2,9
3	10,25	0,25	13	13,25	3

Tableau 11 : FCFS

- Le Plus Court Job D’abord (SJF: Shortest Job First)

  ⇒ Cet algorithme nécessite la connaissance du temps d'exécution estimé de chaque processus. Le CPU est attribué au processus dont le temps d'exécution estimé est minimal.

  ⇒ Exemple : Temps de traitement moyen de 2.38

PID	Heure d'arrivée	Durée d'exécution	Début d'exécution	Fin d'exécution	Temps de traitement
1	10	2	10	12	2
2	10,1	1	12,25	13,25	3,15
3	10,25	0,25	12	12,25	2

▼ avec réquisition du CPU (startégie préemptive)

- Le temps restant le plus court (SRT : Shortest remaining Time)

  ⇒ C'est la version préemptive de l'algorithme SJF. Au début de chaque quantum, le CPU est attribué au processus qui a le plus petit temps d'exécution restant.

⇒ Exemple : On prend un quantum = 1, temps de traitement moyen = 7, les travaux longs peuvent être victime de famine

PID	Heure d'arrivée	Durée d'exécution	Début d'exécution	Fin d'exécution	Temps de traitement
A	0	5	0	10	10
B	1	2	1	3	2
C	2	5	10	15	13
D	3	3	3	6	3

Tableau 13 : SRT

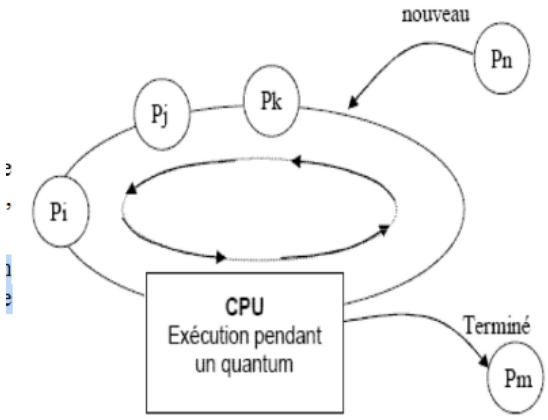
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	B	D	D	D	A	A	A	A	C	C	C	C	C

Tableau 14 : Déroulement de SRT

- Algo à Tourniquet (RR : Round Robin)

  Le contrôle du CPU est attribué à chaque processus pendant une tranche de temps Q, appelée quantum, à tour de rôle.

  Lorsqu'un processus s'exécute pendant un quantum son contexte est sauvegardé et le CPU est attribué au processus prêt suivant



Algorithme à tourniquet (Round Robin)

⇒ Exemple : Q=1, temps de traitement moyen = (37+11.8+5.5) /3= 18.1

PID	Heure d'arrivée	Durée d'exécution	Début d'exécution	Fin d'exécution	Temps de traitement
A	0	30			
B	0,2	5			
C	0,5	2			

Tableau 15 : RR

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	B	C	A	B	C	A	B	A	B	A	B	A	A	A	A	A	A	A

19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

Tableau 16 : RR avec q=1

Avec un quantum = 10 : temps de traitement moyen = (37+14.8+16.5) /3= 22.76

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	A	A	A	A	A	A	A	A	A	B	B	B	B	B	C	C	A	A

19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

Tableau 17 : RR avec q=10

- Algo avec priorité :

  Le système associe à chaque processus une priorité : nombre mesurant l'importance du processus dans le système.
  ⇒ Exemple :

PID	Temps d'arrivée	Durée d'exécution	Priorité
A	0	10	3
B	0	1	1
C	10	2	2
D	0	1	4
E	0	5	2

Tableau 18 : Algorithme avec priorité

Sans réquisition:

1	6	16	18	19
B(1)	E(2)	A(3)	C(2)	D(4)

Avec réquisition:

1	6	10	12	18	19
B(1)	E(2)	A(3)	C(2)	A(3)	D(4)

Tableau 19 : Déroulement algorithme avec priorité

▼ TD9 Exercice 1

Processus	Date d'arrivée	Temps d'exécution
A	0	3
B	1,001	6
C	4,001	4
D	6,001	2

Elaborer le diagramme d'exécution des processus ci-dessus selon l'algorithme d'ordonnancement donné :

1. FCFS :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	A	A	B	B	B	B	B	B	C	C	C	C	D	D

   Temps de traitement moyen : [3 + (9-1,001) + (13-4,001) + (15-6,001)] / 4

2. SJF :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	A	A	B	B	B	B	B	B	D	D	C	C	C	C

   Temps de traitement moyen : [3 + (9-1,001) + (15-4,001) + (11-6,001)] / 4

3. SRT

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	A	A	B	B	B	B	B	B	D	D	C	C	C	C

4. Tourniquet (Q=2)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	A	B	B	A	B	B	C	C	D	D	B	B	C	C

Temps de traitement moyen : [5 + (13-1,001) + (15-4,001) + (11-6,001)] / 4

5. Tourniquer (Q=1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	A	B	A	B	C	B	C	D	B	C	D	B	C	B

Temps de traitement moyen : [4 + (15-1,001) + (14-4,001) + (12-6,001)] / 4

▼ TD9 Exercice 2

Processus	Durée	Arrivée	Priorité
P1	7	0	2
P2	4	0	3
P3	6	1	1
P4	1	1	2
P5	2	1	3
P6	4	2	1
P7	1	2	2

File 3 plus prioritaire que la file 2 qui est plus prioritaire que la file 1

File 3	P2	P5	
File 2	P1	P4	P7
File 1	P3	P6	

1. Un processus peut-il être victime d'un phénomène de famine ?
   ⇒ Oui il peu y avoir un phénomène de famine si des processus + prioritaires continuent d'arriver
2. Quel est le temps de traitement moyen (à un centième prêt) si on utilisait l'algorithme d'ordonnancement "Plus Court d'abord : SJF (Shortest Job First)" au niveau de chacune des trois files ?

0	1	2	3	4	5	6	7	8	...	14	15	...	18	19	...	24
P2	P2	P2	P2	P5	P5	P4	P7	P1	P1	P1	P6	P6	P6	P3	P3	P3

3. Quel est le temps de traitement moyen (à un centième prêt) si on utilisait l'algorithme d'ordonnancement "Tourniquet : RR (Round Robin)" avec quantum=2, au niveau de chacune des trois files ?

0	1	2	3	4	5	6	7	8	9	10	...	14	15	16	17	18
P2	P2	P5	P5	P2	P2	P1	P1	P4	P7	P1	P1	P1	P3	P3	P6	P6

19	20	21	22	23	24
P3	P3	P6	P6	P3	P3

4. Quel est le temps de traitement moyen TTM (à un centième prêt) si on utilisait l'algorithme d'ordonnancement "Temps Restant Plus court d'abord : SRT (Shortest Remaining Time)" avec quantum=1, au niveau de chacune des trois fils ?

0	1	2	3	4	5	6	7	8	...	14	15	...	18	19	...	24
P2	P5	P5	P2	P2	P2	P4	P7	P1	P1	P1	P6	P6	P6	P3	P3	P3

TTM = [(finP2 - arrivéeP2) + (finP5 - arrivéeP5) + … + (finP4 - arrivéeP4)] / 7

TTM = (finP5 + finP2 + finP4 + finP7 + finP1 + finP6 + finP3 - arrivéeP5 - arrivéeP2 - arrivéeP4 - arrivéeP7 - arrivée P1 - arrivéeP6 - arrivéeP3) / 7

TTM = [3 + 6 + 7 + 8 + 15 + 19 + 25 - (0+0+1+1+1+2+2)] / 7

TTM = 10.86

5. Calculer le **temps d'attente moyen (TAM)** en se basant sur la figure suivante :

Processus	Heure d'arrivée	Durée d'exécution	Début d'exécution	Fin d'exécution
A	0	6	0	6
B	2	9	6	22
C	10	4	10	14
D	14	3	14	17

TAM = Σ(TAi) / n Avec TAi = TRi - temps d'exécution Avec TRi = temps fin d'exécution - date d'arrivée

TAM = Σ(temps fin d'exécution - date d'arrivée - temps d'exécution) / n

TAM = `[ (6-0-6) + (22-2-9) + (14-10-4) + (17-3-14) ] / 4`

▼ TD9 Exercice 3

Processus	Date d'arrivée	Temps d'exécution	Priorité de base
P1	0	4	2
P2	1	4	3
P3	1	3	1
P4	4	2	5
P5	5	2	1
P6	6	2	1

Processus

Donner le schéma d'exécution correspondant au système suivant :

Temps	P1 (4)	P2 (4)	P3 (3)	P4 (2)	P5 (2)	P6 (1)
0	2					
1	2.5	3	1			
2	2.5	3.5	1			
3	2.5	4	1			
4	2.5	4.5	1	5		
5	2.5	4.5	1	5.5	1	
6	2.5	4.5	1	X	1	1
7	2.5	X	1	X	1	1
8	3	X	1	X	1	1
9	3.5	X	1	X	1	1
10	X	X	1	X	1	1
11	X	X	1.5	X	1	1

Temps	P1 (4)	P2 (4)	P3 (3)	P4 (2)	P5 (2)	P6 (1)
12	X	X	2	X	1	1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P1	P2	P2	P2	P4	P4	P2	P1	P1	P1	P3	P3	P3	P5	P5	P6	P6

Signaux

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

struct sigaction act_pere, act_fils;
pid_t pid_fils, pid_pere;
int cpt_pere=1, cpt_fils=1; /* nombre de caractères à afficher*/
int idx_pere=0, idx_fils=0; /* indice du caractère à afficher*/

char lettres_fils[26]="abcdefghijklmnopqrstuvwxyz";
char lettres_pere[26]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";

void handler_pere(int x){
    int i;
    for(i=0;i<cpt_pere;i++){
        if(idx_pere<=25){
            printf("%c",lettres_pere[idx_pere]);
            fflush(stdout);
            idx_pere++;
        }else{
            printf("\n");
            exit(0);
        }
    }
    cpt_pere++;
    kill(pid_fils,SIGUSR1);
}
void handler_fils(int x){
    int i;
    for(i=0;i<cpt_fils;i++){
        if(idx_fils<=25){
            printf("%c",lettres_fils[idx_fils]);
            fflush(stdout);
            idx_fils++;
        }else{
            kill(pid_pere,SIGUSR1);
            exit(0);
        }
    }
    cpt_fils++;
    kill(pid_pere,SIGUSR1);
}
int main(){
    pid_fils=fork();
    if(pid_fils==0){
        /*processus fils: minuscules*/
        pid_pere=getppid();
        act_fils.sa_handler=handler_fils;
        sigaction(SIGUSR1,&act_fils,NULL);
```

```
        while(1){
            pause();
        }
    }else{      /*processus pere: majuscules*/
        act_pere.sa_handler=handler_pere;
        sigaction(SIGUSR1,&act_pere,NULL);
        sleep(1);
        kill(pid_fils,SIGUSR1);
        while(1){
            pause();
        }
    }
}
```

Final Blanc

Soit le programme suivant :

```
char a[100];
main(int argc, char ** argv)
{
   int d;
   static double b;
   char * s = "boo", * p;
   p = malloc(300);
   return 0;
}
```

Cochez l'affirmations correcte

Veuillez choisir une réponse.

- a. On peut pas savoir l'emplacement exacte du tableau a, la variable statique b et la constante "boo"
- b. Le tableau a, la variable statique b et la constante "boo" sont tous dans le segment pile.
- c. Le tableau a, la variable statique b et la constante "boo" sont tous dans le segment de données.
- d. Le tableau a, la variable statique b et la constante "boo" sont tous dans le segment code.
- e. Aucune de ces réponses

Votre réponse est incorrecte.

La réponse correcte est : Le tableau a, la variable statique b et la constante "boo" sont tous dans le segment de données.

Cette question utilise les notations suivantes pour décrire l'allocation des ressources dans un système informatique :

- Ri: vecteur de requête pour le processus Pi
- Ai: vecteur d'allocation actuel pour le processus Pi
- U: vecteur de ressource non alloué (disponible)

U = (0,1,0,0) R1 = (1,0,0,0), R2 = (0,1,0,0), R3 = (0,0,1,0) , R4 = (1,0,0,0), A1 = (0,2,0,0), A2 = (1,0,0,1), A3 = (1,0,0,0), A4 = (0,0,1,0).

Ce système est dans une situation d'interblocage? Répondez par vrai (si interblocage) ou faux (si pas d'interblocage).

Veuillez choisir une réponse.

- a. On ne peut pas savoir
- b. Vrai
- c. Faux

Votre réponse est incorrecte.

La réponse correcte est : Faux

Note : Ordre = P2, P4, P3, P1

Soit le programme suivant :

```
char a[100];
main(int argc, char ** argv)
{
   int d;
   static double b;
   char * s = "boo", * p;
   p = malloc(300);
   printf("%s",s);
   return 0;
}
```

Cochez l'affirmation correcte.

Veuillez choisir une réponse.

- a. Les variables automatiques d, s et p se trouvent dans le segment de données
- b. Les variables automatiques d, s et p se trouvent dans le segment de pile
- c. On peut pas savoir l'emplacement exacte des des variables automatiques d, s et p
- d. Les variables automatiques d, s et p se trouvent dans le segment code
- e. Aucune de ces réponses

Votre réponse est incorrecte.

La réponse correcte est : Les variables automatiques d, s et p se trouvent dans le segment de pile

---

Soit un système qui utilise l'algorithme du banquier possédant 5 processus (P0, P1, P2, P3, P4) et 3 types de ressources (A, B, C). Le type A possède 10 instances, le type B possède 5 instances et le type C possède 7 instances. Initialement, l'état des ressources du système est décrit par les matrices Allocation, Max et Available suivantes :

Allocation

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Max

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Available

A	B	C
3	3	2

Le système est-il dans un état sain ? Répondez par vrai (il est dans un état sain) ou faux (il est dans un état malsain)

Veuillez choisir une réponse.

- Vrai
- Faux

La réponse correcte est « Vrai ».

Soient les trois processus P1, P2, P3 suivants :
init(mutex, 1) ; init(synchro,0);
Processus P1
Debut
Répéter
P(mutex)
A1
V(synchro)
Jusqu'à Faux

Debut
Répéter
P(syncho)
A2
V(mutex)
Jusqu'à Faux
Fin.
Processus P3
Debut
Répéter
P(synchro)
A3
V(mutex)
Jusqu'à Faux
Fin.
Ce schéma de synchronisation entre les processus P1, P2 et P3 permet de faire en sorte que :

Veuillez choisir une réponse.

- a. Les actions Ai peuvent être exécutées simultanément et peuvent se dérouler dans un ordre aléatoire.
- b. Les actions Ai se déroulent dans un ordre aléatoire mais jamais d'une façon simultanée.
- c. Aucune de ces réponses.
- d. Les actions Ai ne s'exécutent jamais simultanément et se déroulent toujours dans l'ordre A1A2A3A1A2A3…
- e. Les actions Ai ne s'exécutent jamais simultanément et se déroulent toujours dans l'ordre A1(A2 ou A3)A1(A2 ou A3)…

Votre réponse est incorrecte.
La réponse correcte est : Les actions Ai ne s'exécutent jamais simultanément et se déroulent toujours dans l'ordre A1(A2 ou A3)A1(A2 ou A3)…

On considère le problème des lecteurs/rédacteurs avec l'hypothèse que le nombre de lectures simultanées est borné (soit K le nombre de lectures max simultanées). On suppose que les rédacteurs sont prioritaires que les lecteurs. On propose les algorithmes ci-dessous permettant de résoudre ce problème :

nL,nR : integer init 0 ; // nombre lecteurs et nombre rédacteurs//
mutex1,mutex2 : semaphore init 1 ; // garantir l'exclusion mutuelle pour les variables nL et nR//
lecteur, redacteur : semaphore init 1 ; // priorité des redacteurs par rapport aux lecteurs//
nb_lect: semaphore init K ; // K lecteurs au plus peuvent occuper le fichier en même temps//

Lecteur

Début
….(1)….
P(lecteur) ;
P(mutex1) ;
nL:=nL+1;
if nL=1 then P(redacteur) ;
V(mutex1) ;
V(lecteur) ;
<Lire le fichier>
P(mutex1) ;
nL :=nL-1 ;
if nL=0 then V(redacteur) ;
V(mutex1) ;
….(2)…. ;
Fin.

Redacteur

Début
P(mutex2) ;
nR :=nR+1 ;
if nR=1 then P(lecteur) ;
V(mutex2) ;
….(3)…. ;
<Ecrire dans le fichier>
….(4)…. ;

Tableau de bord Événements Mes cours Cours actuel Aide

V(mutex2) ;
Fin.

Choisir la bonne réponse permettant de corriger les algorithmes ci-dessus :

Veuillez choisir une réponse.

- a. Les programmes sont incorrects, il faut les changer complètement
- b. (1) = P(nb_lect) ; (2) = V(nb_lect) ; (3)= P(redacteur) ; (4)= V(redacteur)
- c. Les programmes sont corrects, pas besoin de rajouter des instructions supplémentaires
- d. (1) = rien ; (2) = V(nb_lect) ; (3) = rien; (4)= V(nb_lect) ;
- e. (1) = rien ; (2) = rien ; (3)= P(nb_lect) ; (4)= V(nb_lect) ;
- f. (1) =rien; (2) = V(nb_lect) ; (3)= P(redacteur) ; (4)= V(redacteur) ; V(nb_lect) ;

Votre réponse est incorrecte.
La réponse correcte est : (1) = P(nb_lect) ; (2) = V(nb_lect) ; (3)= P(redacteur) ; (4)= V(redacteur)

Si on augmente le quantum dans l'algorithme d'ordonnancement Round Robin alors :

Veuillez choisir au moins une réponse.

- ☐ a. Aucune de ces réponses
- ☐ b. Le temps de rotation augmentera
- ☐ c. Le temps de rotation baissera
- ☐ d. Le temps de réponse augmentera
- ☐ e. Le nombre de commutation des contextes augmentera

Votre réponse est incorrecte.
Les réponses correctes sont : Le temps de réponse augmentera, Le temps de rotation augmentera