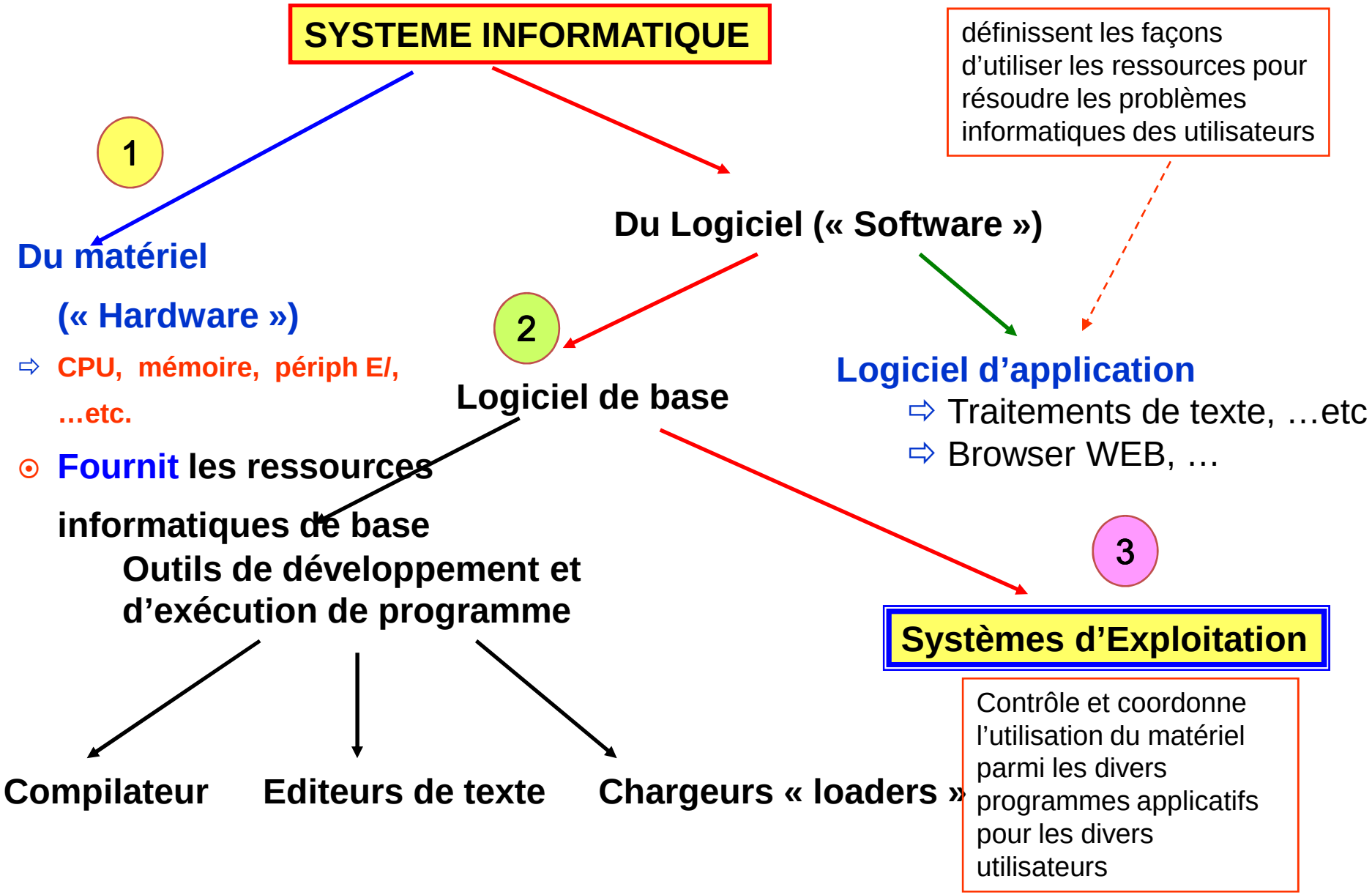
Le système d'exploitation

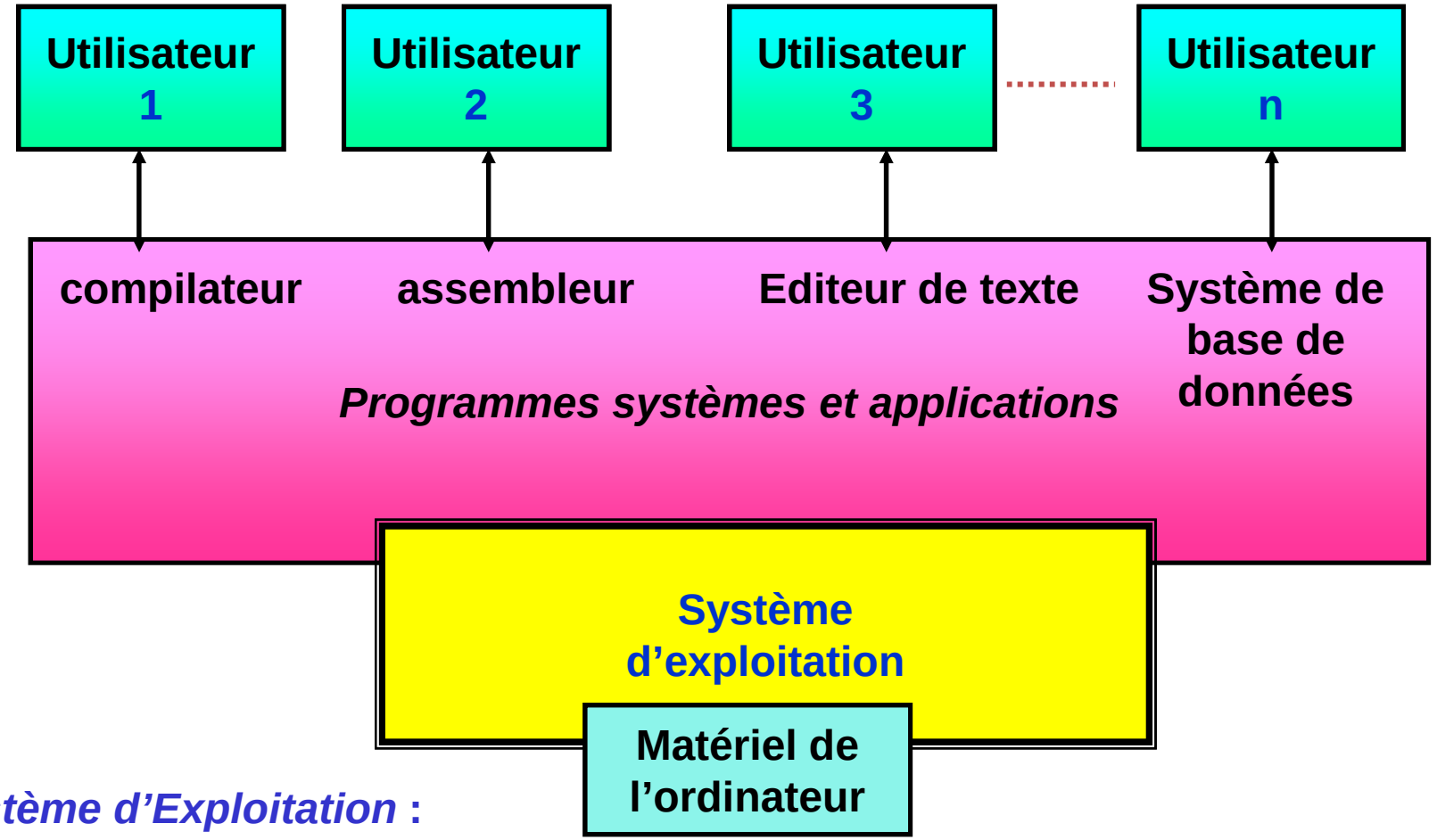
Rappel : Système Informatique : Les composantes d'un système informatique

- **Un SE est une partie importante de presque tout système informatique**



Rappel : Système Informatique « Place du Système d'exploitation »

Vue abstraite d'un système informatique



***Le Système d'Exploitation* :**

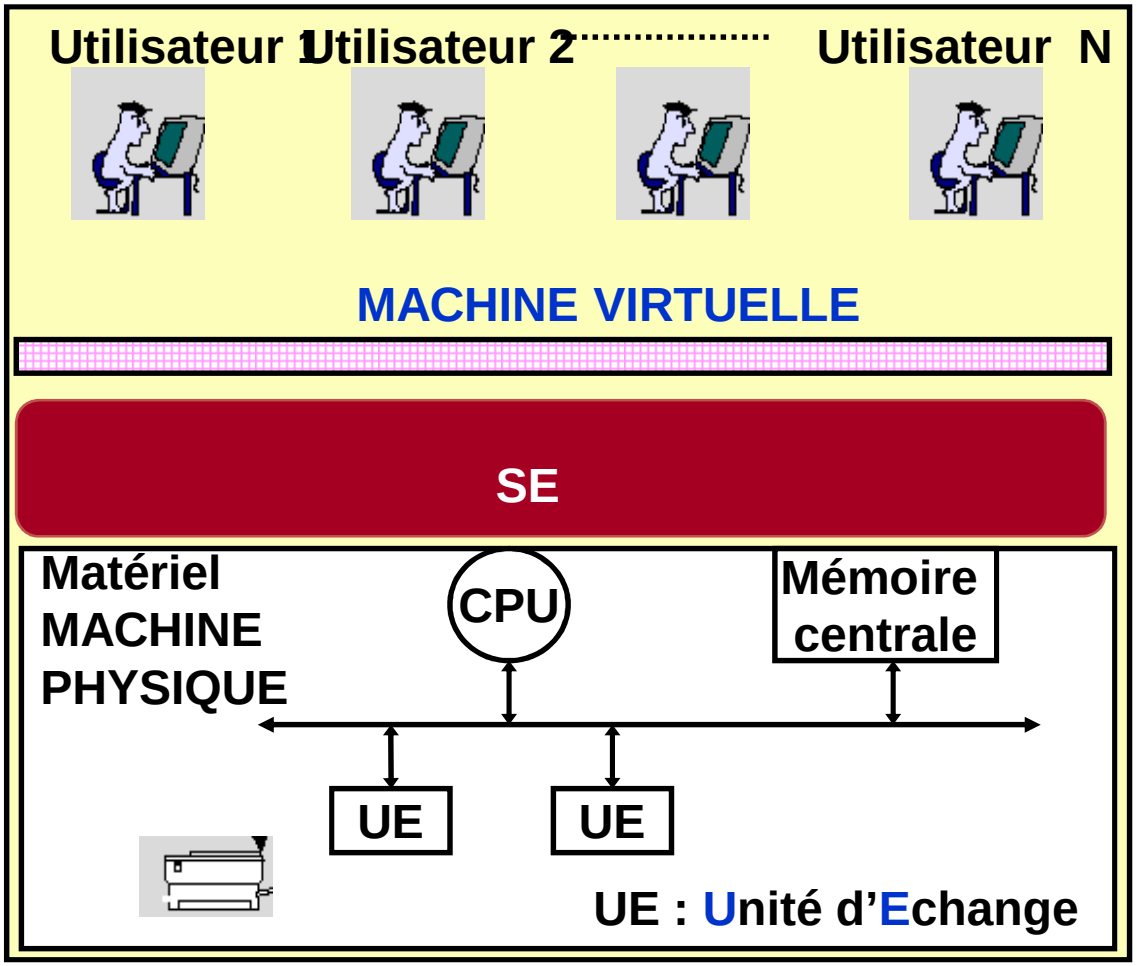
- **Contrôle et coordonne l'utilisation du matériel par les divers programmes applicatifs pour les divers utilisateurs**
  - ➢ **Fournit, donc, les moyens d'utiliser correctement les ressources matériels**

DEFINITION D'UN SYSTEME D'EXPLOITATION

- Le Système d'exploitation est un ensemble de programmes qui réalisent l'interface entre le matériel de l'ordinateur et les utilisateurs.

Deux objectifs principaux

1. Construction d'une machine virtuelle, plus facile d'emploi et plus conviviale
2. gestion des ressources physiques et leur partage



RÔLES DU SYSTÈME D'EXPLOITATION

- Rôles du système d 'exploitation :

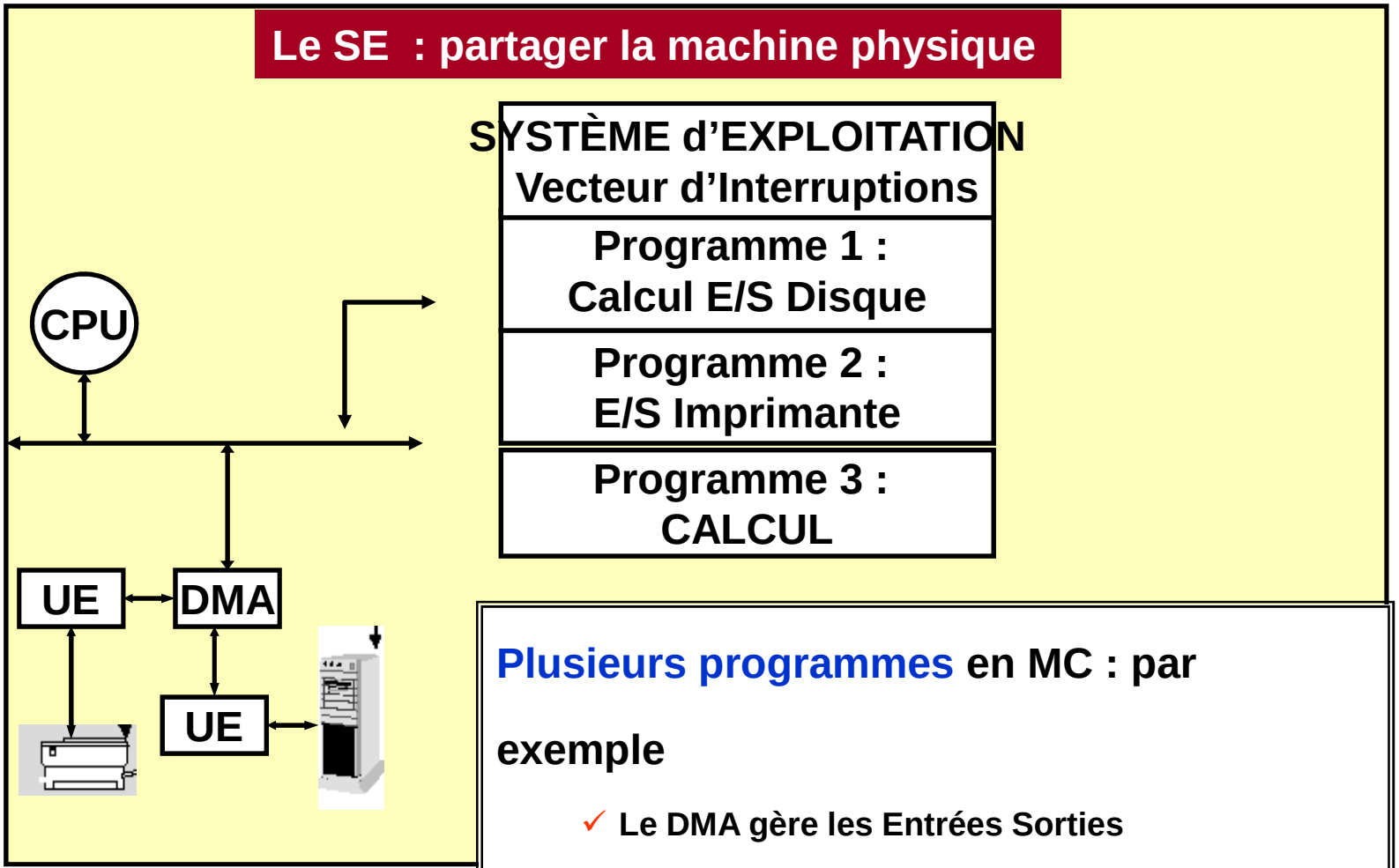
1. Gérer le partage de la machine physique et des ressources matérielles entre les différents programmes (<=> dans un environnement multiprogrammé)

2. Faciliter l'accès à la machine physique

LE SYSTÈME D'EXPLOITATION : PARTAGER LA MACHINE PHYSIQUE (1)

Il doit optimiser l`utilisation des ressources pour maximiser les performances du système

- Cette gestion doit assurer l' équité d'accès aux ressources matérielles
- Assurer également que les accès des programmes à ces ressources s'effectuent correctement, c'est-à-dire que les opérations réalisées par les programmes sont licites (<=> *protection des ressources*).



Plusieurs programmes en MC : par exemple

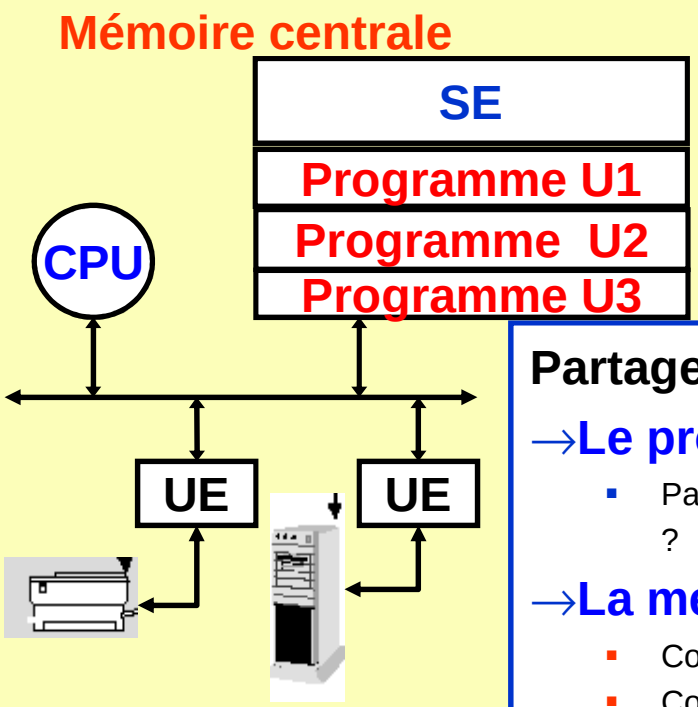
- ✓ **Le DMA gère les Entrées Sorties**
- ✓ **Parallèlement, le CPU effectue les « calculs »**

LE SYSTÈME D'EXPLOITATION : PARTAGER LA MACHINE PHYSIQUE (2)

Le SE « Contrôle » l'exécution des applications

- ✓ Le SE doit céder le contrôle aux programmes utilisateurs et le reprendre correctement
- ✓ « Dit » à l'UCT quand exécuter tel programme

Le SE partage la machine physique



- **Le partage des ressources va concerner principalement**
  - ⇨ Le processeur,
  - ⇨ La mémoire centrale
  - ⇨ Les périphériques d ' entrées-sorties.

Partager les ressources ⇔

→Le processeur :

- Parmi tous les programmes chargés en mémoire centrale, lequel doit s'exécuter ?

→La mémoire centrale :

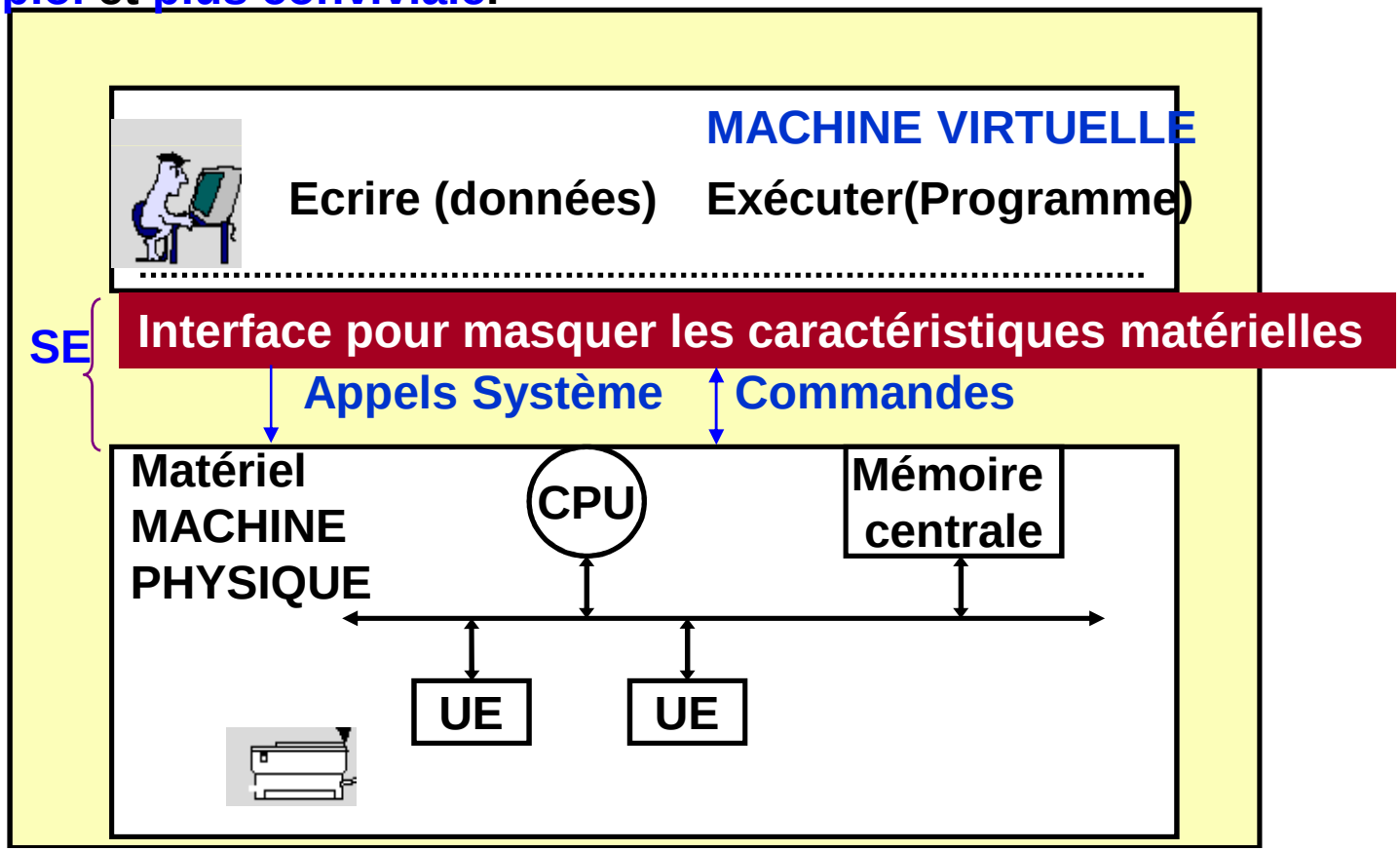
- Comment **allouer la mémoire centrale** aux différents programmes ?
- Comment **disposer d'une quantité suffisante de mémoire** pour y placer tous les programmes nécessaires à un bon taux d'utilisation du processeur ?
- Comment assurer **la protection entre ces différents programmes utilisateurs** ? (⇔ veiller à ce qu'un programme donné n'accède pas à une plage mémoire allouée à un autre programme).

→Les Périphériques :

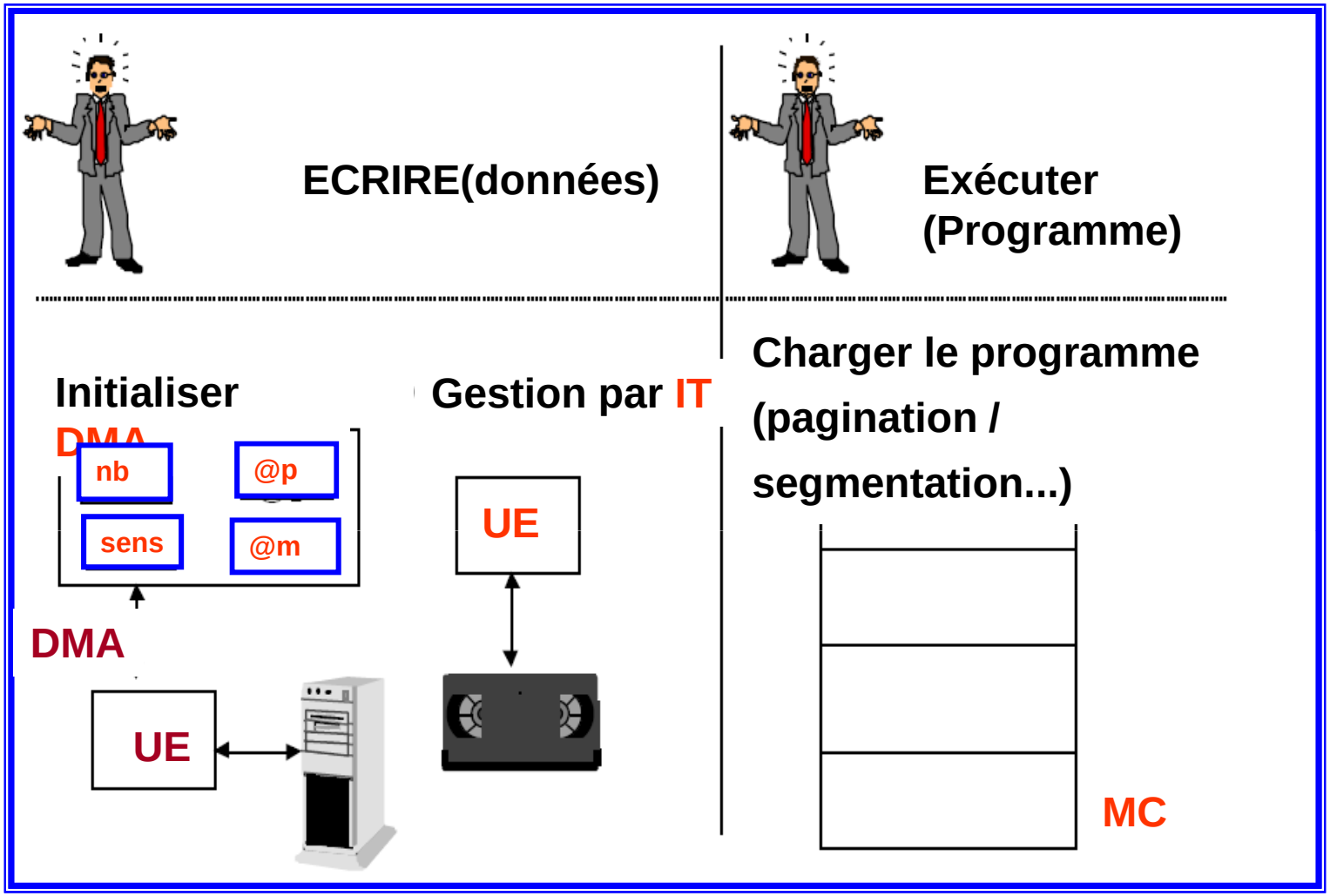
- Dans quel ordre traiter les requêtes d'E/S pour optimiser les transferts ?

LE SYSTÈME D'EXPLOITATION : le SE pour faciliter l'accès à la machine physique (1)

- **Fournit une interface de haut niveau, composée d'un ensemble de primitives attachées à des fonctionnalités**
  - Ce sont ces primitives qui gèrent les caractéristiques matérielles sous-jacentes et offrent un service à l'utilisateur
- **Le SE construit au-dessus de la machine physique, une machine virtuelle plus simple d'emploi et plus conviviale.**



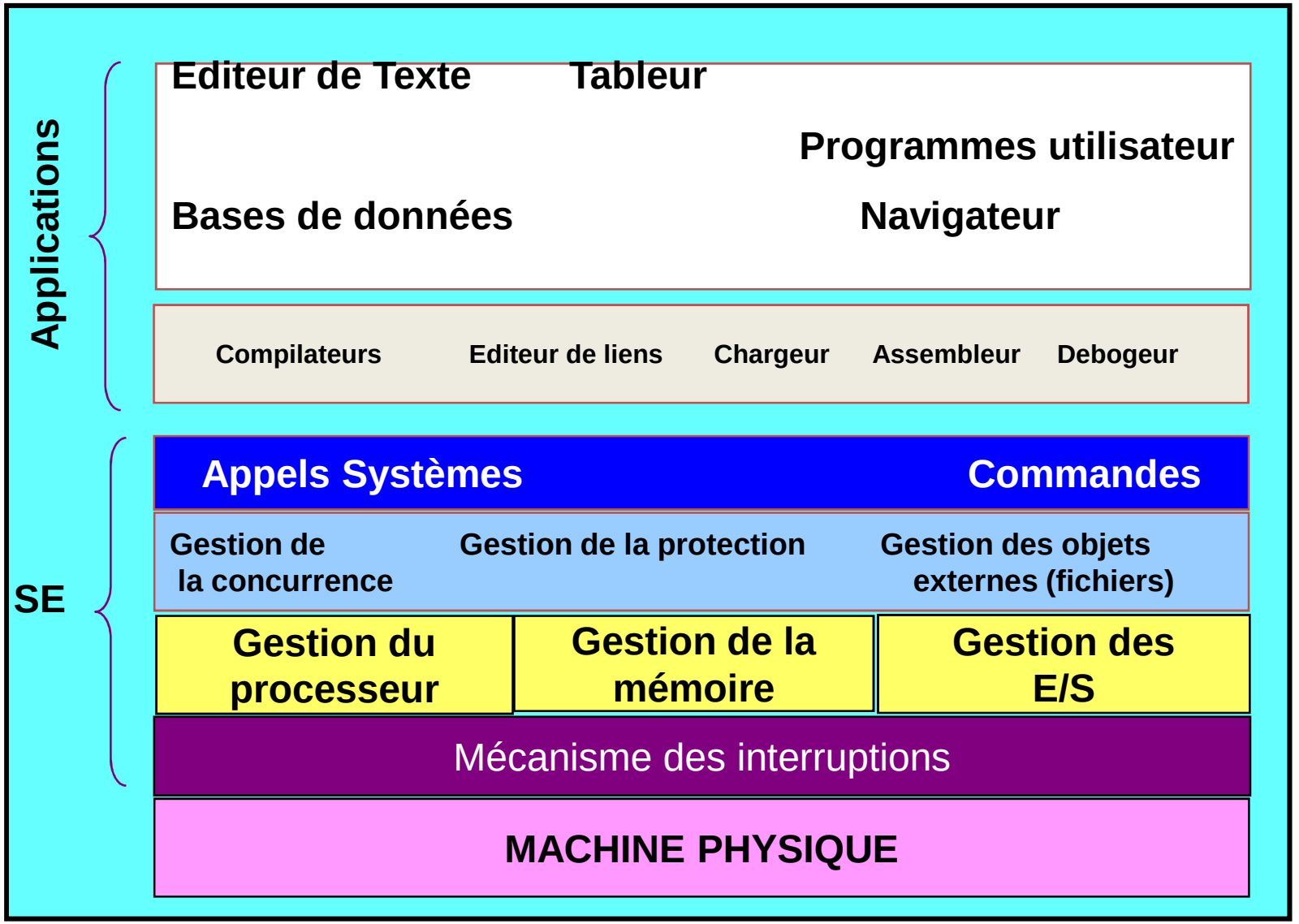
LE SYSTÈME D'EXPLOITATION : le SE pour faciliter l'accès à la machine physique (2)



- **Exemple :** Pour réaliser une opération d'entrées- sorties, l'utilisateur fait appel à **une même primitive** **ECRIRE**(données) quel que soit le périphérique concerné.
  - C'est la primitive **ECRIR**E et la **fonction de gestion des entrées-sorties** du système d'exploitation à laquelle cette primitive est rattachée qui feront la liaison avec les caractéristiques matérielles.

FONCTIONS D'UN SYSTEME D'EXPLOITATION

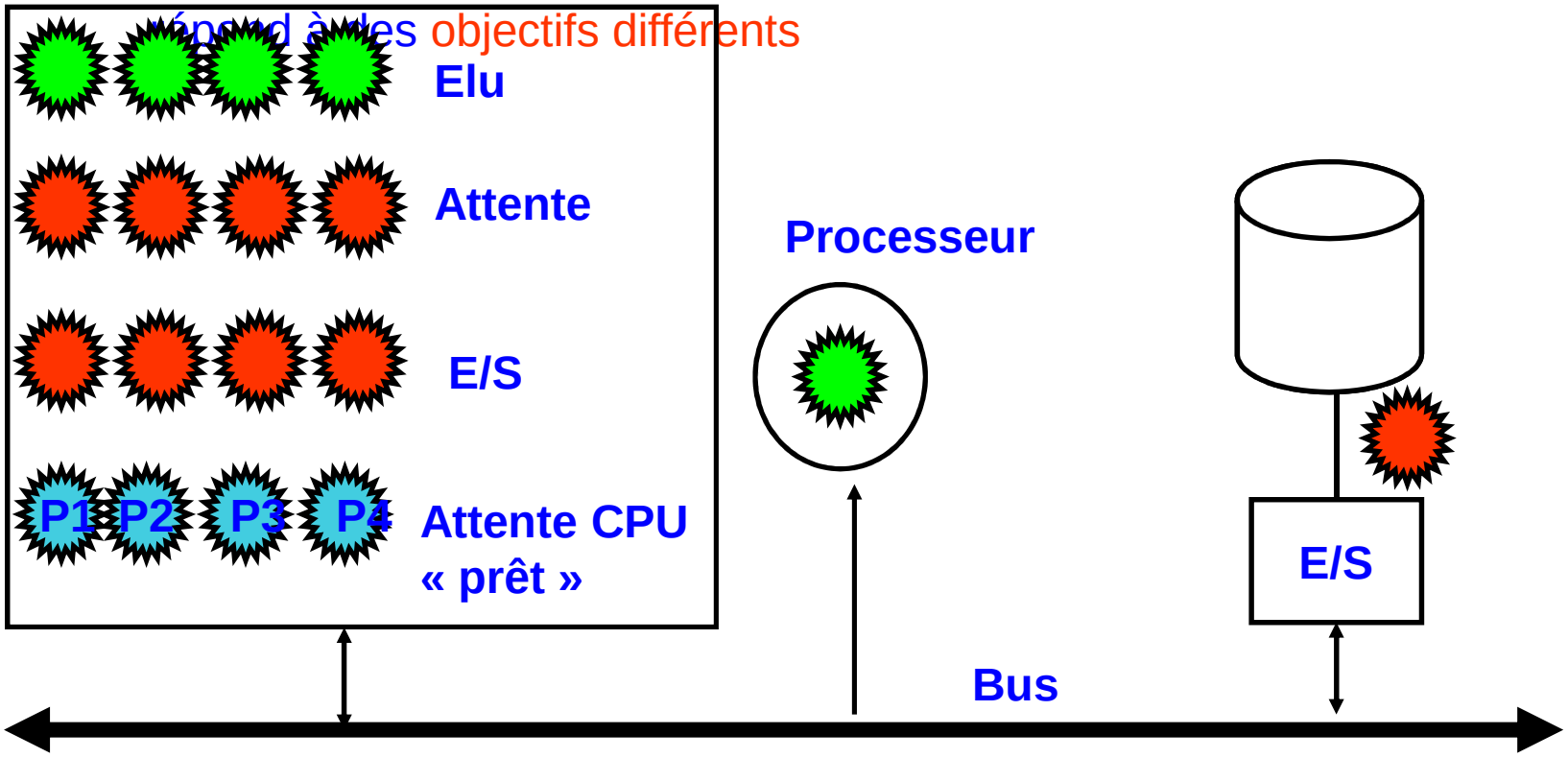
FONCTIONS D'UN SYSTEME D'EXPLOITATION (1)



FONCTIONS D'UN SYSTEME D'EXPLOITATION (2)

1. Gestion du processeur :

- le système doit gérer l'allocation du processeur aux différents programmes pouvant s'exécuter. Cette allocation se fait par le biais d'un algorithme d'ordonnancement qui planifie l'exécution des programmes
  - ➢ Selon le type de système d'exploitation, l'algorithme d'ordonnancement répond à des objectifs différents

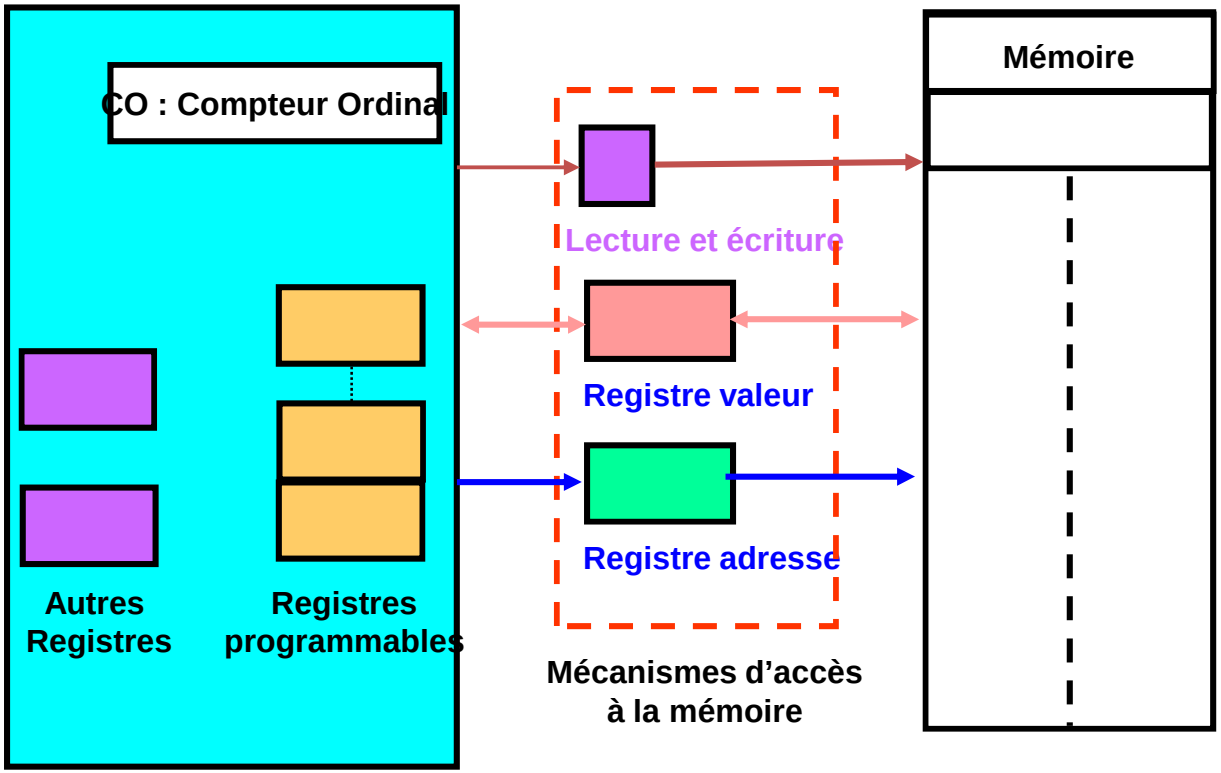


FONCTIONS D'UN SYSTEME D'EXPLOITATION (2)

2. Gestion de la mémoire :

- le système doit gérer l'allocation de la mémoire centrale entre les différents programmes pouvant s'exécuter (pagination/segmentation).
- Comme la mémoire physique est souvent trop petite pour contenir la totalité des programmes, la gestion de la mémoire se fait selon le principe de la mémoire virtuelle.
  - *à un instant donné, seules sont chargées en MC, les parties de code et données utiles à l'exécution*

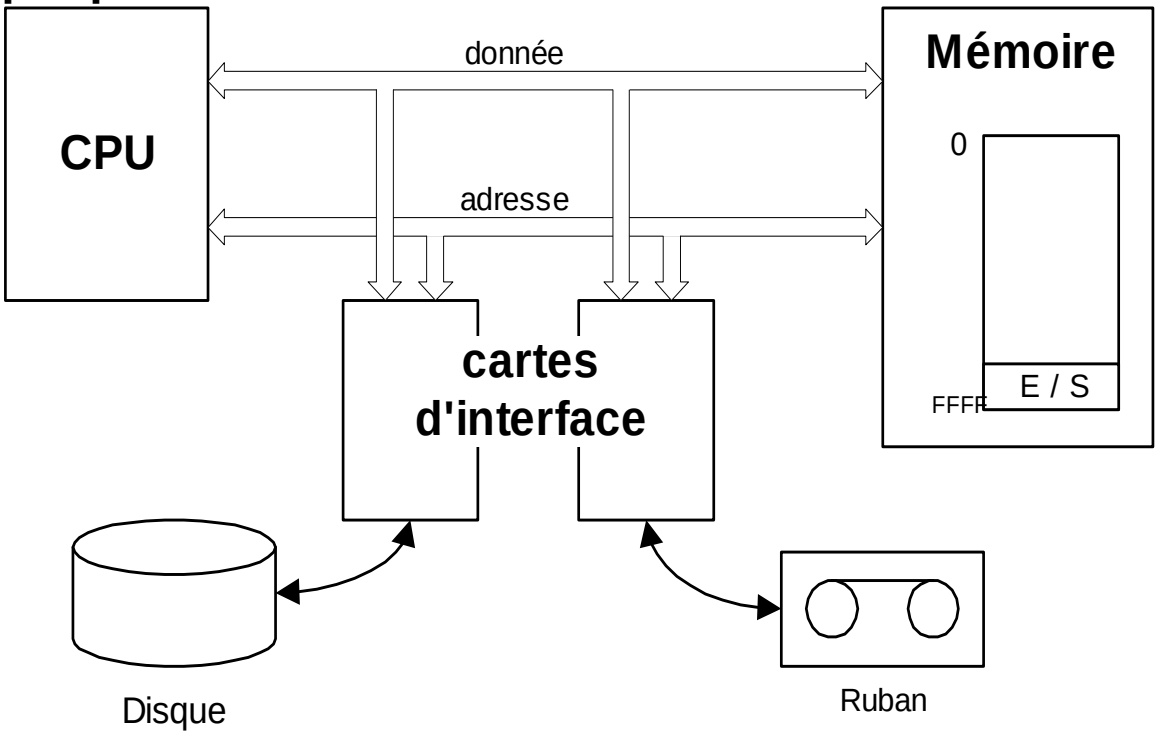
- **Problèmes de l'allocation mémoire ?**
  - correspondance entre adresses virtuelles et adresses physiques ;
  - gestion de la mémoire physique ;
- **Et si multiprocessus :**
  - partage de l'information ;
  - protection mutuelle.



FONCTIONS D'UN SYSTEME D'EXPLOITATION (3)

3. Gestion des entrées/sorties :

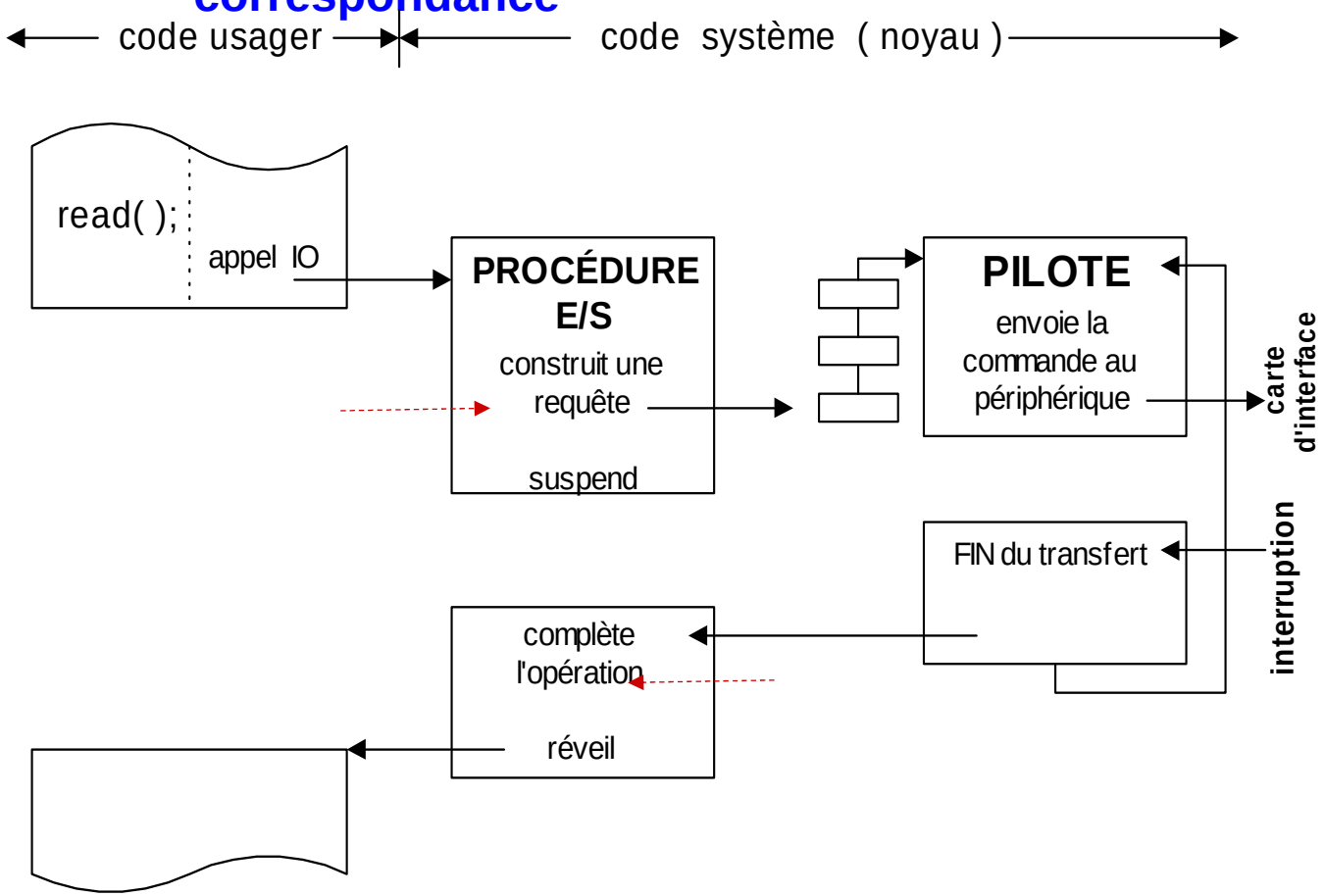
- **le système doit gérer l'accès aux périphériques,**
  - **Plusieurs mécanismes de communication entre UCT et contrôleurs périphériques**
    - → **Interruptions**
    - → **Scrutation**
    - → **DMA**
  - **Pilotes et contrôleurs de périphs**



FONCTIONS D'UN SYSTEME D'EXPLOITATION (4)

⇔ **la liaison entre les appels de haut niveau des programmes utilisateurs [exemple getchar( )] et les opérations de bas niveau de l'UE responsable du périphérique (UE clavier)**

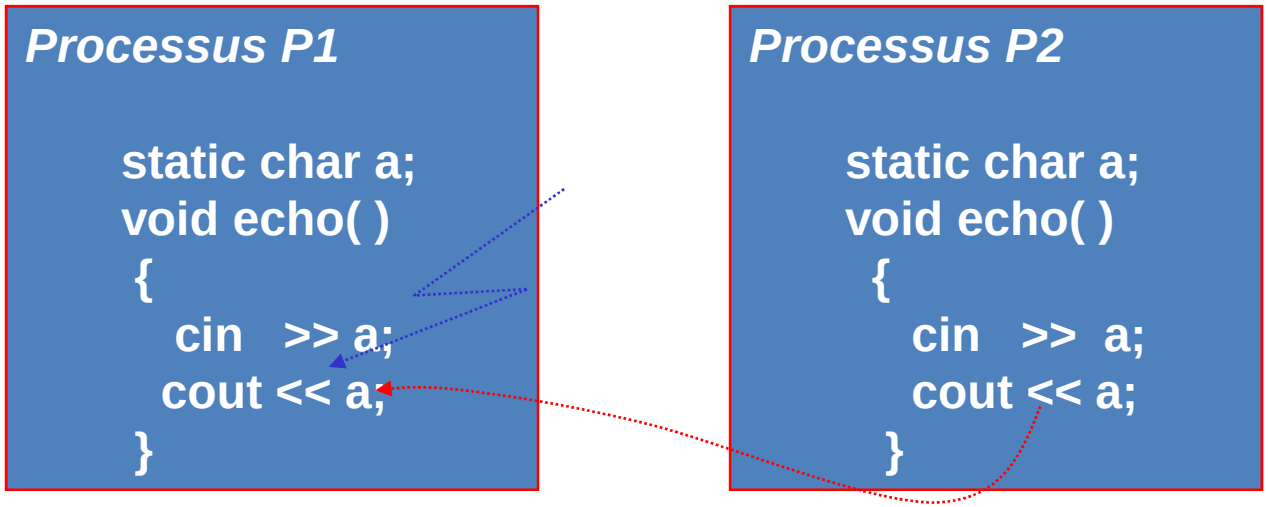
→ **c'est le pilote d'entrées/sorties (driver) qui assure cette correspondance**



FONCTIONS D'UN SYSTEME D'EXPLOITATION (5)

4. Gestion de la concurrence

- **Comme plusieurs programmes coexistent en mémoire centrale, ceux-ci peuvent vouloir communiquer pour échanger des données.**
- **Par ailleurs, il faut synchroniser l'accès aux données partagées afin de maintenir leur cohérence**
  - Le système offre des outils de communication et de synchronisation entre programmes
- **Exemple :**



FONCTIONS D'UN SYSTEME D'EXPLOITATION (6)

5. Gestion des objets externes

- **La mémoire centrale est une mémoire volatile. Aussi, toutes les données devant être conservés au delà de l'arrêt de de la machine, doivent être stockées sur une mémoire de masse (disquette, Cédérom...)**
  - ⇨ la gestion de l'allocation des mémoires de masse ainsi que l'accès aux données stockées s'appuient sur la notion de fichiers et de système de gestion de fichiers (SGF).

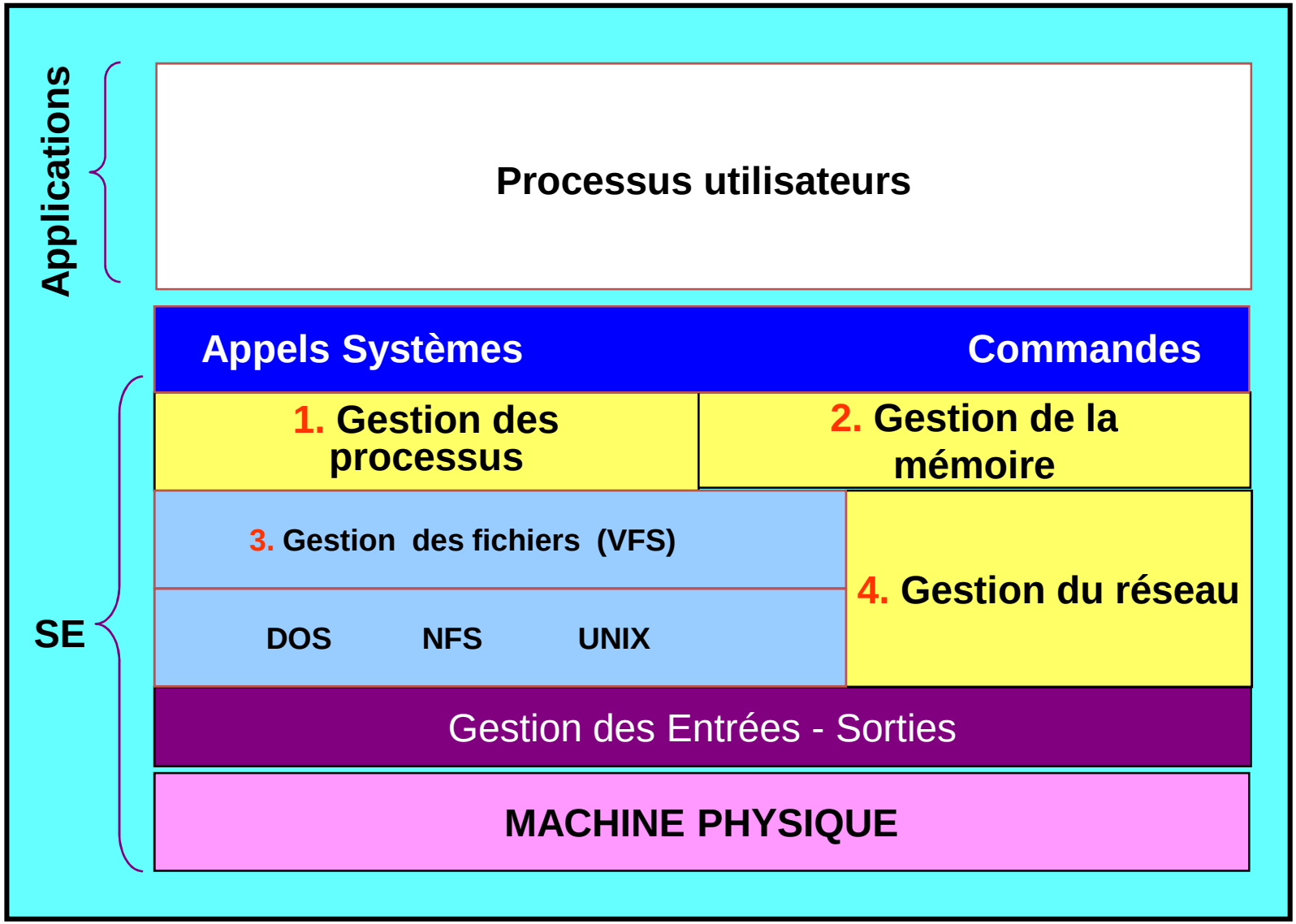
FONCTIONS D'UN SYSTEME D'EXPLOITATION (7)

6. Gestion de la protection

- le système doit fournir des mécanismes garantissant que ses ressources (CPU, mémoire, fichiers) ne peuvent être utilisées que par les programmes auxquels *les droits nécessaires ont été accordés*.
  - il faut notamment protéger le système et la machine des programmes utilisateurs (mode d'exécution utilisateur et superviseur)

Linux : Structure de base

Structure :



Linux : Structure de base

- **Le système Linux est structuré comme un noyau monolithique qui peut être découpé en quatre grandes fonctions :**

1. une fonctionnalité de gestion des processus qui administre les exécutions de programmes, le *processus* étant l'image dynamique de l'exécution d'un programme
2. *une fonctionnalité de gestion de la mémoire ;*
3. *une fonctionnalité de gestion des fichiers de haut niveau, le VFS (Virtual File System) qui s'interface avec des gestionnaires de fichiers plus spécifiques de type Unix, DOS, etc., lesquels s'interfacent eux-mêmes avec les contrôleurs de disques, disquettes, CD-Rom, etc. ;*
4. *une fonctionnalité de gestion du réseau qui s'interface avec le gestionnaire de protocole puis le contrôleur de la carte réseau.*

Linux – services fournis

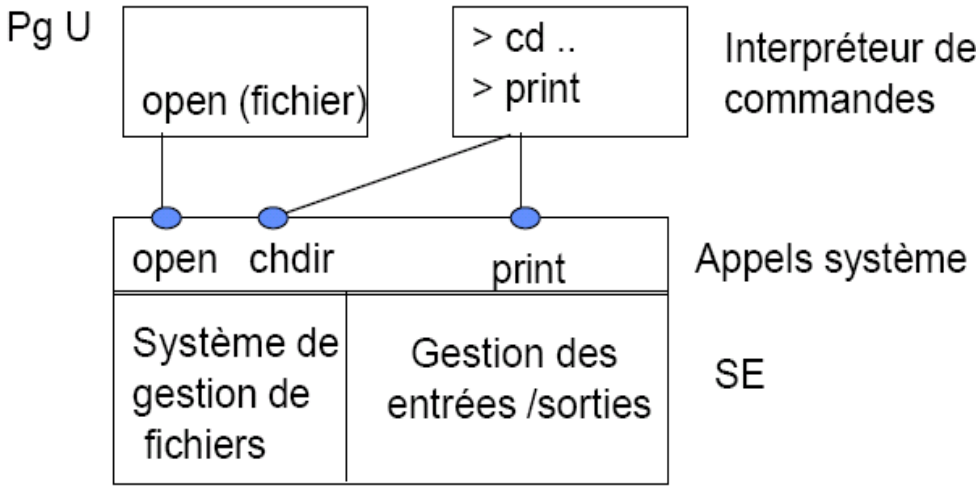
- **Le SE s'interface avec les Prog U par le biais des routines systèmes**

  ⇨ **constituent les points d 'entrées des fonctionnalités du système.**

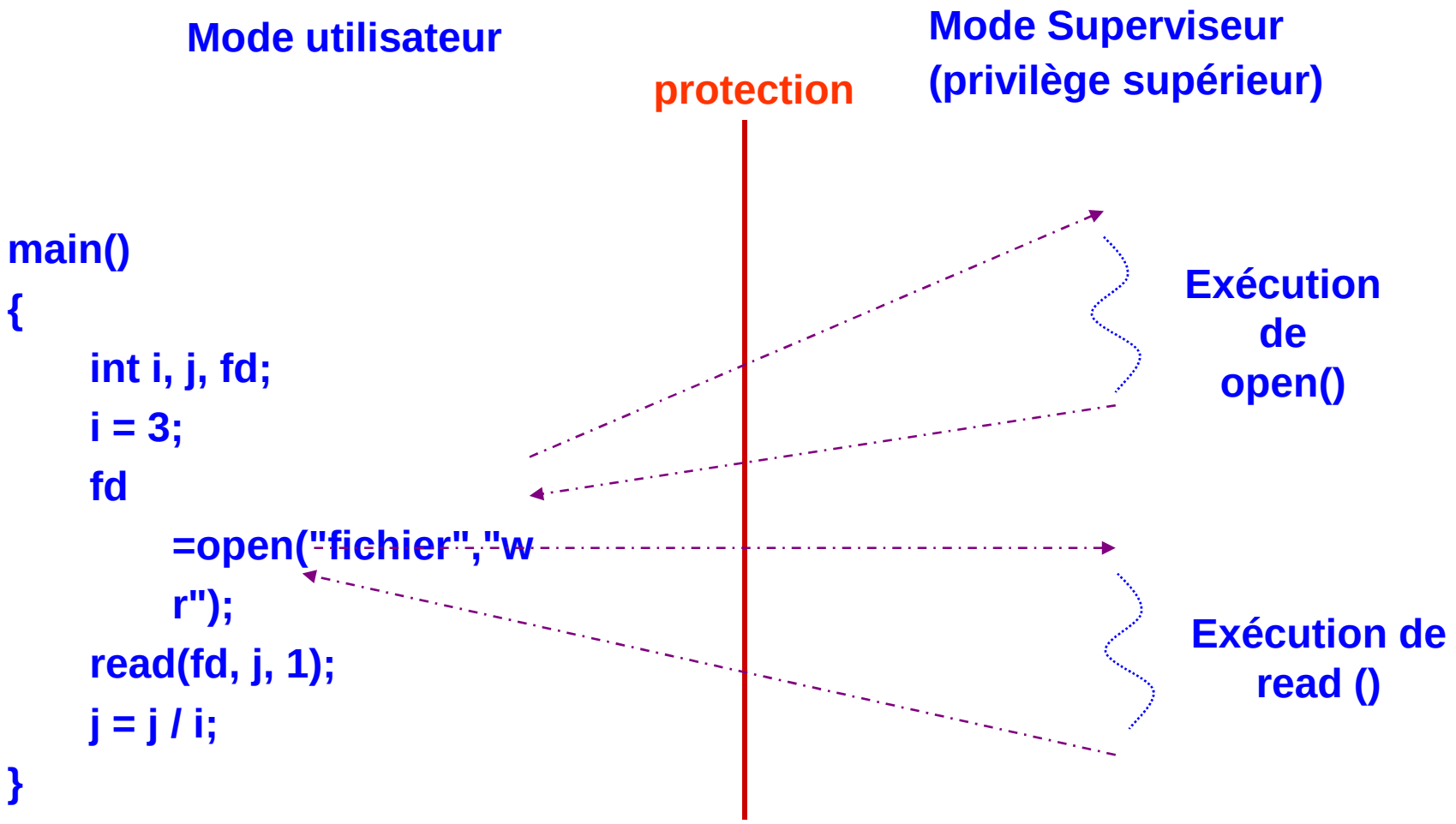
- **Les services fournis par Linux aux routines systèmes peuvent être appelés de deux façons par les processus utilisateurs,**

  1. soit par le biais d'un ***appel système***,
  2. soit par le biais d'une ***commande du langage de commandes.***

- **L'exécution des routines systèmes s'effectue dans le mode privilégié (*mode superviseur*)**



Linux : modes d'exécution



Linux : Modes d'exécutions

Linux : modes d'exécution

⊙Linux : Modes d'exécutions

- **Un processus utilisateur s'exécute par défaut en mode utilisateur (User Mode) :**
  - **Dans ce mode, les actions du programme sont volontairement restreintes afin de protéger la machine des actions parfois malencontreuses du programmeur.**
  - **Exemple : Les instructions permettant la manipulation des interruptions est interdite.**
- **Le système d'exploitation, quant à lui, s'exécute dans un mode privilégié (mode superviseur (Kernel Mode)),**
  - **pour lequel aucune restriction de droits n'existe.**
- **Le codage du mode utilisateur « esclave » et du mode superviseur « maître » est réalisé au niveau du processeur, dans le registre d'état de celui-ci.**

Linux : Commutation de Contexte – principe -

⊙ Linux : Commutation de Contexte :

- **Le passage d'un processus du mode utilisateur au mode superviseur constitue une *commutation de contexte:***

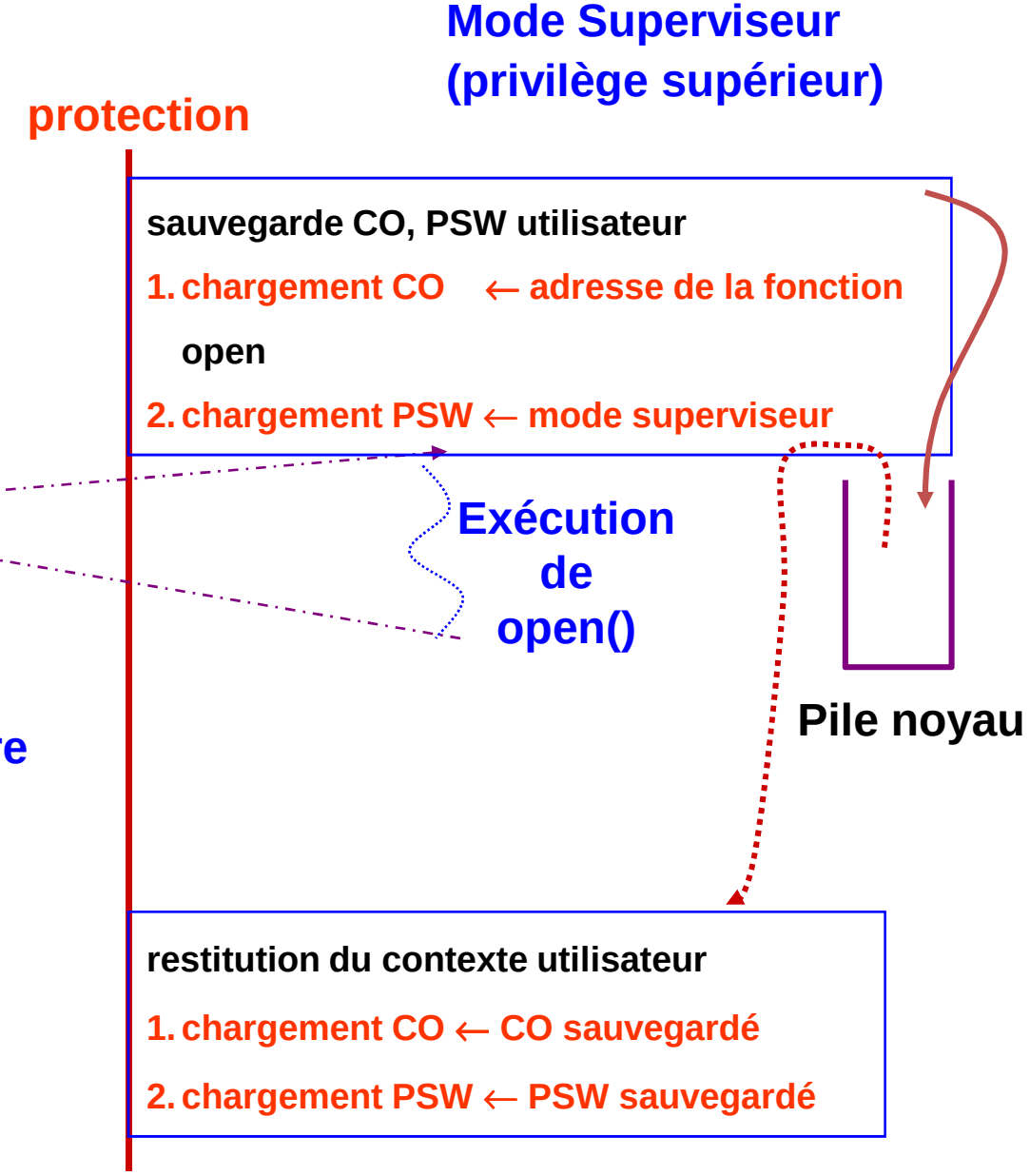
1. **Sauvegarde du contexte utilisateur**
   - **La valeur des registres du processeur (compteur ordinal, registre d'état), sur la pile noyau.**
2. **Un contexte noyau est chargé :**
   - **Valeur de compteur ordinal correspondant à l'adresse de la fonction à exécuter dans le noyau,**
   - **Et d'un registre d'état en mode superviseur.**
3. **A la fin de l'exécution de la fonction système, le processus repasse en mode utilisateur.**
   - **Il y a de nouveau une opération de commutation de contexte :**
   - ***Restauration du contexte utilisateur* ,**
   - **Reprise de l'exécution du programme utilisateur juste après l'appel**

Linux : Commutation de Contexte –exemple -

Linux : Commutation de Contexte

Mode utilisateur

```
main( )
{
    int i, j, fd;
    Char tab[10];
    i = 3;
    fd =open("fichier","wr");
    If(fd== -1)
    {
        perror("Problème ouverture
        fichier\n");
        exi t ( 1) ;
    }
    Else
    {
        read (fd, tab, 10) ;
        printf("caractères lus: %s \n" ,
```



Linux : Commutation de Contexte – Différentes causes

⊙ **Trois causes majeures provoquent le passage du mode utilisateur au mode superviseur :**

1. Lorsqu'un processus utilisateur effectue un **appelle d'une fonction du système.**

   → C'est une demande explicite de passage en mode superviseur

2. L'exécution par le processus utilisateur **d'une opération illicite** (division par 0, instruction machine interdite, violation mémoire ...) :

   → C'est la *trappe* ou *l'exception.*

   → L'exécution du processus utilisateur est alors arrêtée ;

3. La prise en compte ***d'une interruption par le matériel*** (IRQ *Interrupt Request)* par le système d'exploitation.

   → Le processus utilisateur est alors stoppé et l'exécution de la routine d'interruption associée à l'interruption survenue est exécutée en mode superviseur.

Linux : Commutation de Contexte – causes – exemple -

Mode utilisateur

```
main( )
{
    int i, j, fd;
    Char tab[10];
    i = 3;
    fd =open("fichier","wr");
    If(fd== -1)
    {
        perror("Problème ouverture fichier\n");
        exi t ( 1) ;
    }
    Else
    {
        read (fd, tab, 10) ;
        printf("caractères lus: %s \n" , tab) ;
        i=i/j;
        Close (fd);
    }
}
```

protection

Mode Superviseur (privilège supérieur)

Exécution de open()

APPELS SYSTEME

IRQ
Exécution du traitant d'it Horloge

TRAPPE
erreur irrecouvrable arrêt du programme

Trois causes de commutation de contexte

Notion de processus

Notion de processus

QU'ETS-CE QUE C'EST UN PROCESSUS ?

- Définition : Un processus est un programme en cours d'exécution auquel est associé un environnement processeur (CO, PSW, registres généraux, …etc.) et un environnement mémoire appelés contexte du processus

- Processus ≠ Programme :
  - ☞ **Un programme**
    - ✓ peut être exécuté plusieurs fois
    - ✓ Peut se trouver dans plusieurs unités d'exécution en même temps (il constitue un objet inerte)
  - ☞ **Le processus doit connaître à chaque instant**
    - » le code du programme
    - » le pointeur d'instruction
    - » l'état de la pile
    - » les variables, …etc.

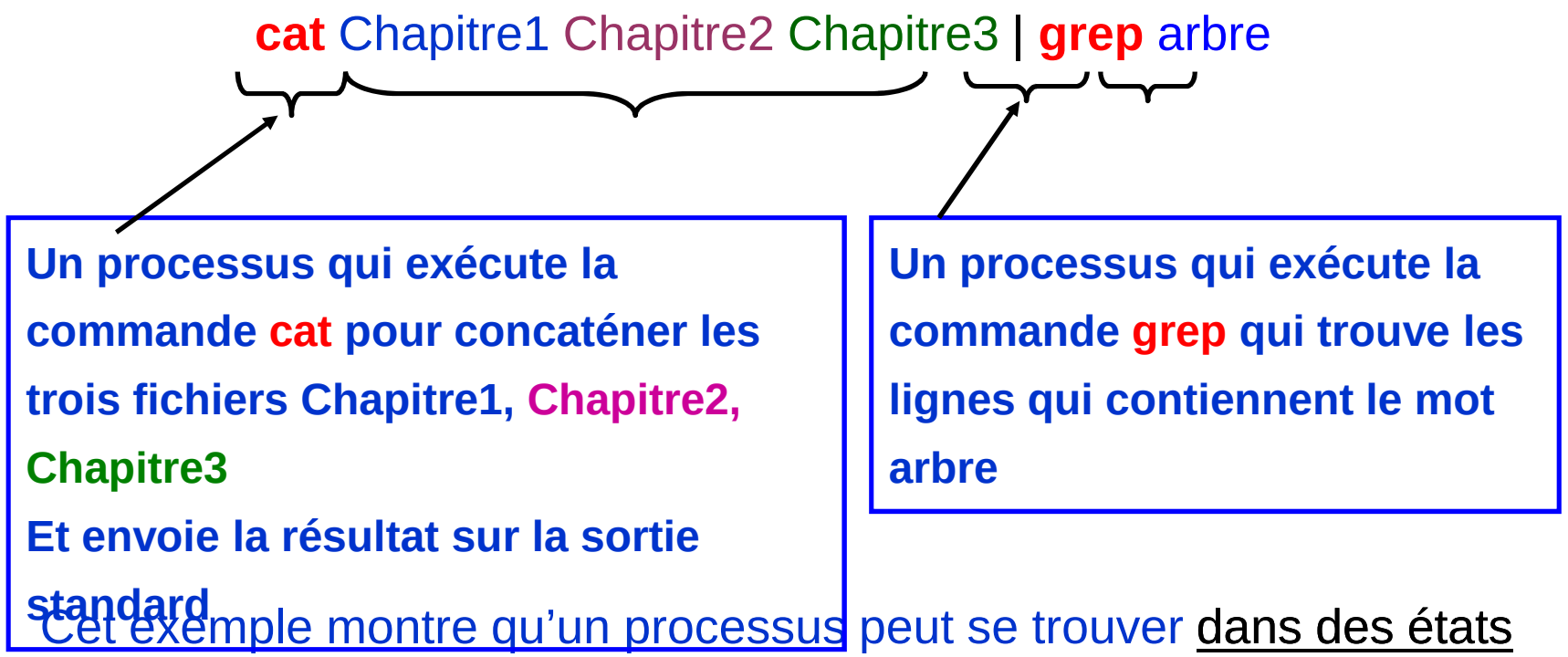
- Donc : Un processus est l'instance dynamique d'un programme et incarne le fil d'exécution de celui-ci dans un espace d'adressage protégé (objets propres : ensemble des instructions et données accessibles)

⇔ Un processus est un programme (⇔code) en exécution (⇔ environnement)

Etat des processus

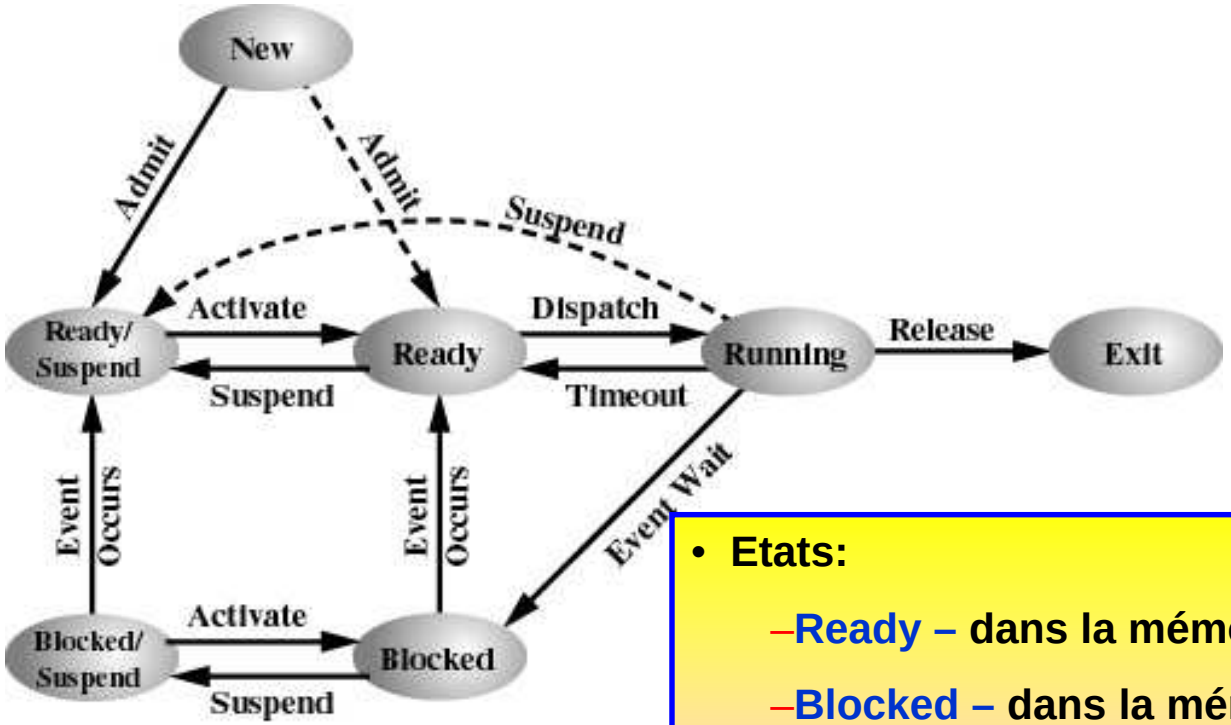
Système multiprocessus : Etats des processus

- Les processus, bien qu'étant indépendants, doivent parfois interagir avec d'autres processus
- Exemple : Les résultats d'un proc. Peuvent parfois être les données d'un autre Proc.



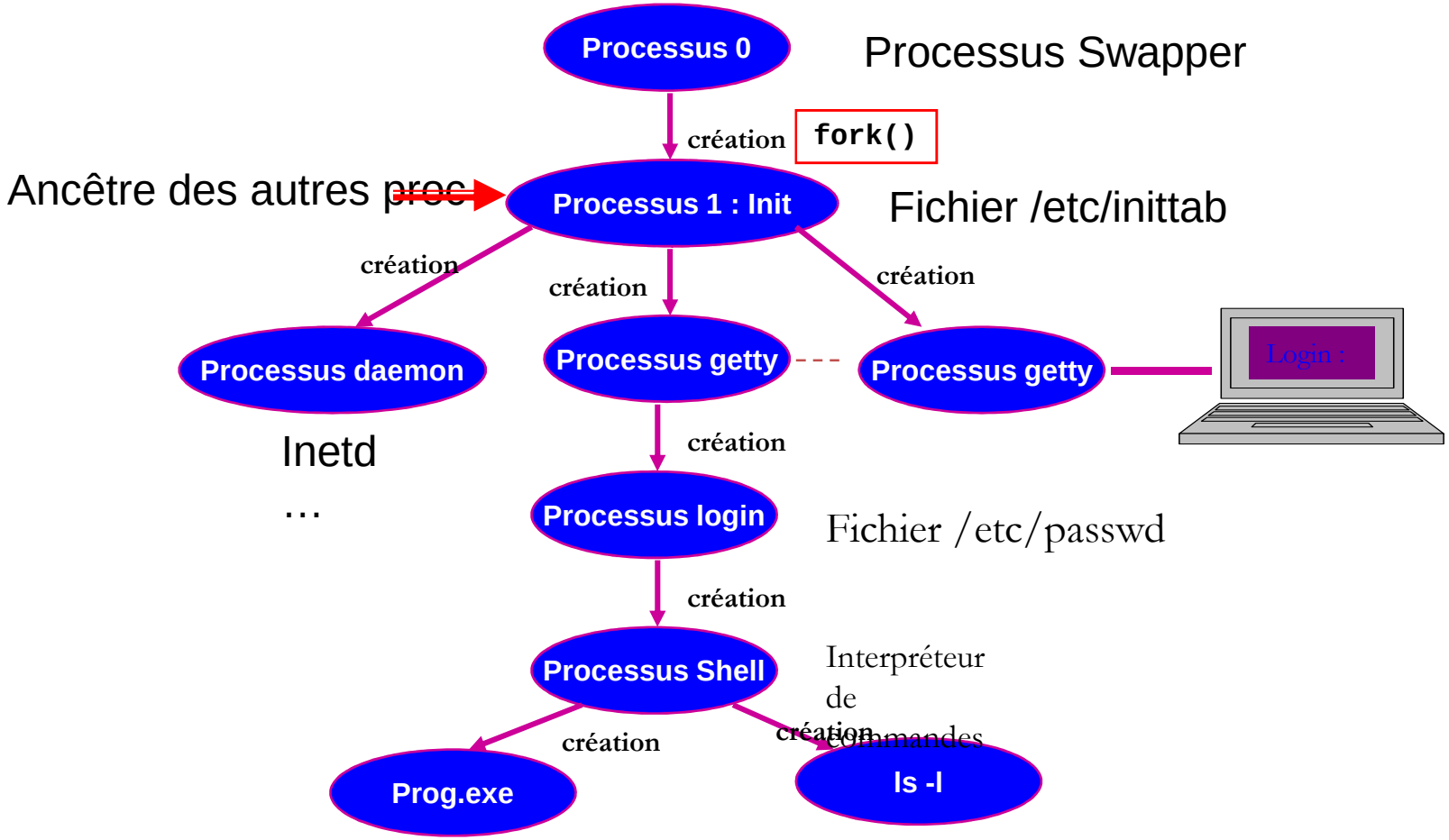
- Cet exemple montre qu'un processus peut se trouver dans des états différents
  - selon le contexte et la disponibilité des ressources nécessaires

Diagramme état du processus



- **Etats:**
  - **Ready – dans la mémoire, prêt à être exécuté**
  - **Blocked – dans la mémoire, en attente d'un événement**
  - **Blocked/Suspend – sur le disque, en attente d'un événement**
  - **Ready/Suspend – sur le disque, prêt à être exécuté après son transfert en mémoire**

Hiérarchie des processus dans UNIX



- **Exemple, Init crée un processus fils, appelé getty pour chaque terminal reconnu par la machine.**

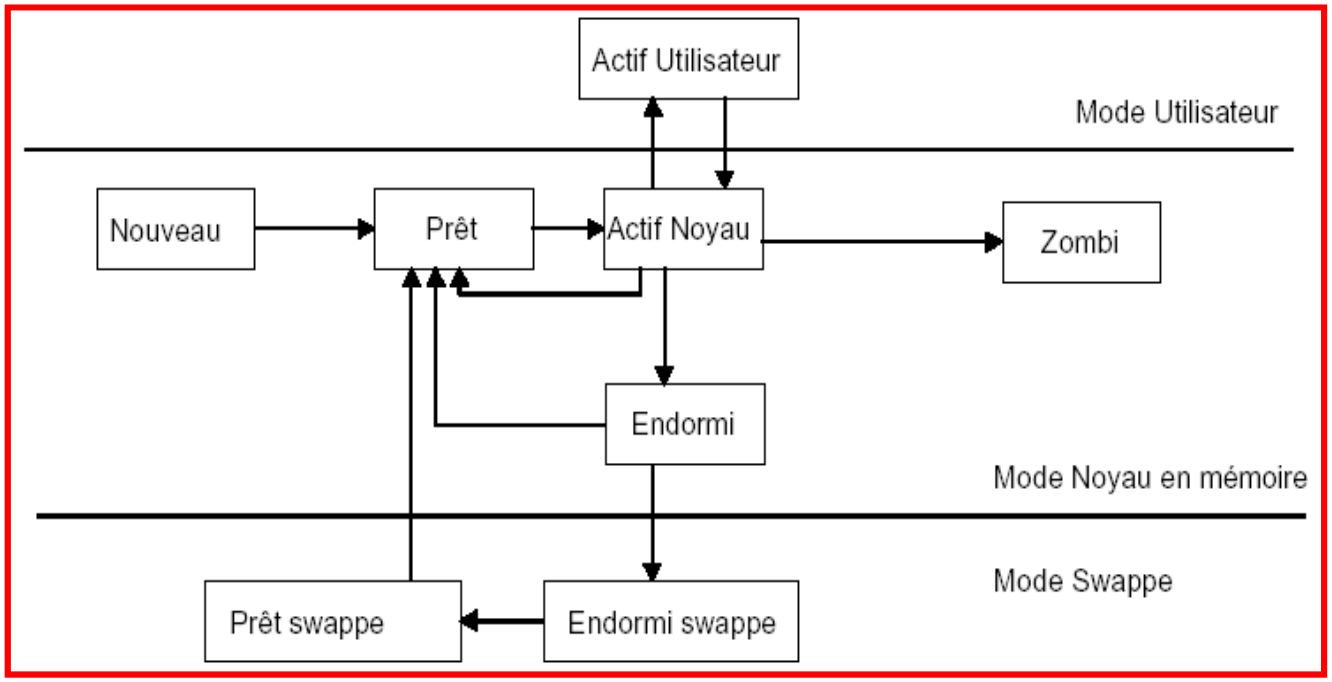
  →Ce processus attend les tentatives de connexion (*login* et *mot de passe*) des utilisateurs.

  →Lorsqu'un utilisateur est admis, le processus **getty** est remplacé par un processus **shell.**

  →Celui-ci appartient à l'utilisateur, il sera l'ancêtre de tous les processus créés par l'utilisateur durant la session

Processus UNIX – Modes d'exécution

- **Graphe d'états simplifié pour un processus Unix : Un processus évolue entre trois modes au cours de son exécution :**
  1. Le **mode utilisateur** qui est le mode normal d'exécution,
  2. Le **mode noyau** en mémoire qui est le mode dans lequel se trouve un processus prêt ou bloqué (endormi)
  3. Et le **mode swappé** qui est le mode dans lequel se trouve un processus bloqué (endormi swappé) déchargé de la mémoire centrale.



Politique d'ordonnancement

- Définition d'une politique d'ordonnancement :

  ⇔ Répondre aux trois questions suivantes :

  1. Qui va être exécuté ?
  2. Pour combien de temps?
  3. Quand appeler l'Ordonnanceur?

Politique d'ordonnancement : Qui?

- Algorithmes non préemptifs :

  (on ne peut pas interrompre un processus):

  1. Premier arrivé , premier servi (FCFS : First-Come, First-Served)
  2. Plus court d'abord (SJF : Shortest Job First)

- Algorithmes préemptifs : (on peut interrompre un processus):

  1. Plus court temps restant (SRT : Shortest Remaining Time)
  2. Chacun son tour (Tourniquet)
  3. Avec Priorités

- Quelques principes aussi
  - **Les processus temps réel devraient avoir la plus haute priorité**
  - **On veut favoriser les processus qui utilisent peu le processeur (en général, ils sont du types interactifs)**

Politique d'ordonnancement : Quand?

- Voici les principales possibilités:
  - Quand le processus en exécution se termine
  - Quand le processus en exécution bloque
  - Quand le processus en exécution a écoulé son quantum
  - Quand un nouveau processus est passé à l'état *prêt* (surtout s'il est prioritaire)
  - À chaque interruption d'horloge

METHODES D’ORDONNANCEMENT

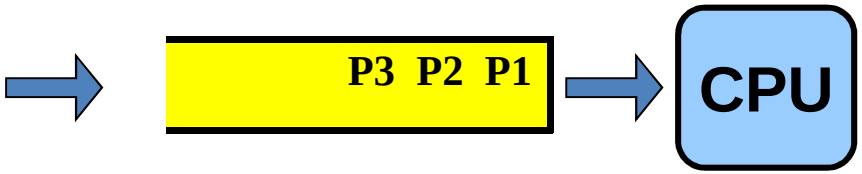
PREMIER ARRIVÉ, PREMIER SERVI (PAPS) : First-Come, First-Served (FCFS)

❑ ***Principes et caractéristiques :***

- ⇨ C'est l'algorithme le plus simple
- ⇨ Dans ce schéma, le premier processus qui demande le processeur le reçoit en premier (⇔ Premier Arrivé Premier Servi).
- ⇨ L'implémentation de la politique FCFS est facilement gérée avec une file d'attente FIFO.
- ⇨ Lorsqu'un processus entre dans la liste d'attente des processus prêts, son PCB est chaîné à la fin de la file d'attente.
- ⇨ Lorsque le processeur est libérée, on lui alloue le processus en tête de file.
  - ➢ Le processus en exécution est alors retiré de la file.

- Exemple d'arrivée des processus : P1, P2 et P3

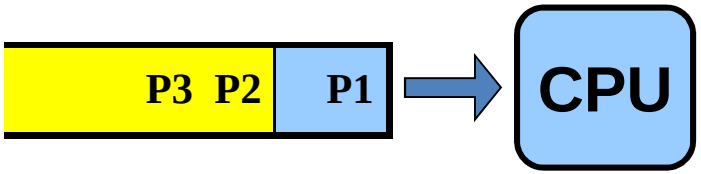
processus	temps d'arrivée	temps CPU
P1	0	24
P2	0	3
P3	0	3

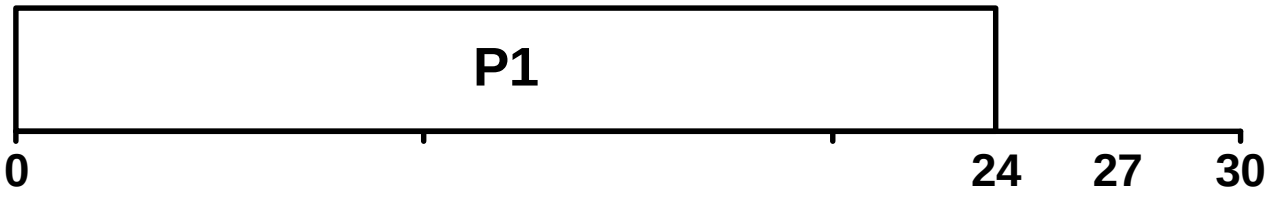


FCFS

processus	temps d'arrivée	temps CPU
P1	0	24
P2	0	3
P3	0	3

temps = 0

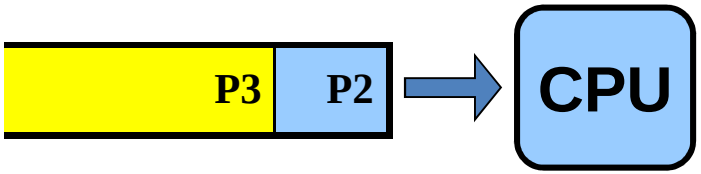


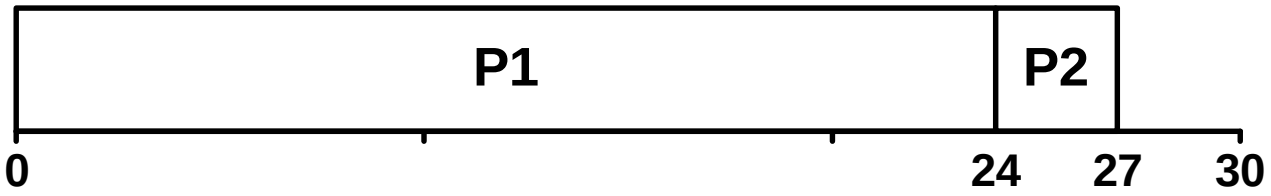


FCFS

processus	temps d'arrivée	temps CPU
P1	0	24
P2	0	3
P3	0	3

temps = 24

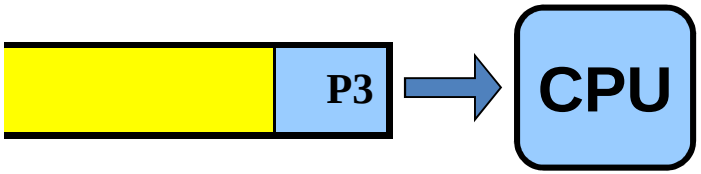


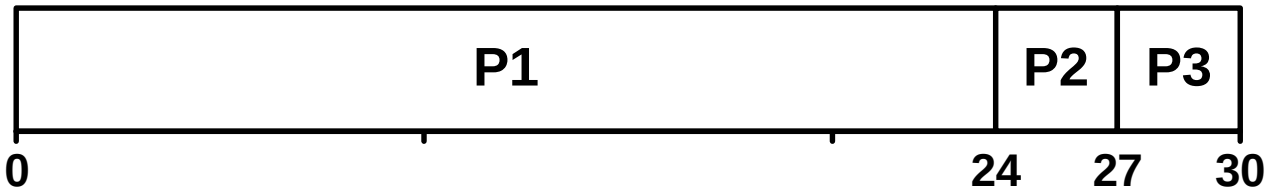


FCFS

processus	temps d'arrivée	temps CPU
P1	0	24
P2	0	3
P3	0	3

temps = 27

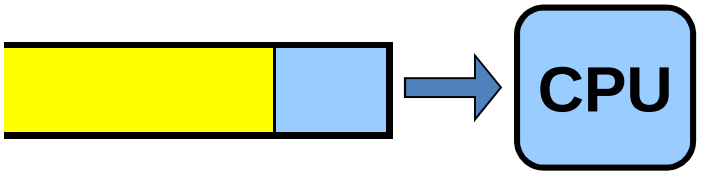


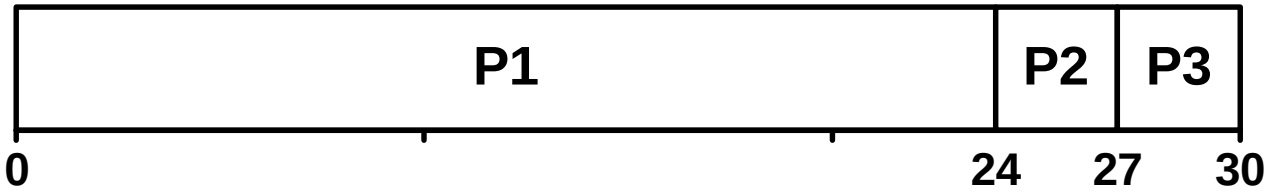


FCFS

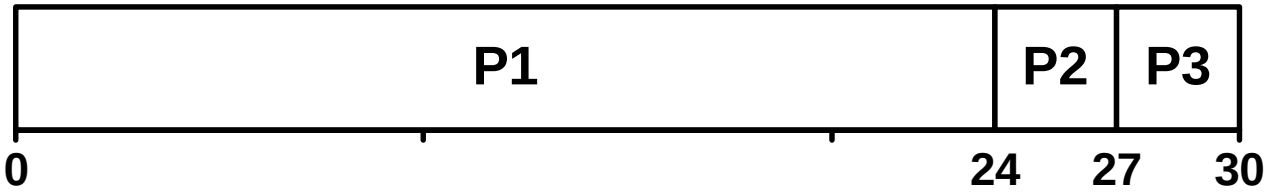
processus	temps d'arrivée	temps CPU
P1	0	24
P2	0	3
P3	0	3

temps = 30





FCFS – Bilan



processus	temps de traitement	temps CPU (d'exécution)	temps d'attente
P1	24	- 24	= 0
P2	27	- 3	= 24
P3	30	- 3	= 27
total	81	- 30	= 51
moyenne(÷3)	27	- 10	= 17

Tourniquet = Round-Robin (RR)

ROUND ROBIN OU TOURNIQUET

- Algorithme ancien, simple, fiable et très utilisé

- Principes de base :

1. On gère la file d'attente des processus prêts comme une file FIFO de processus.
   - ⇨ Les nouveaux processus sont ajoutés en queue de file.
   - ⇨ L'Ordonnanceur d'exécution prend le processus dans la tête de la file,
   - ⇨ Démarre une minuterie qui se déclenchera après un quantum et lance le processus.

2. Le processeur passe d' un processus à l'autre en exécutant chacun pendant quelques dizaines ou centaines de ms (⇔ valeur du quantum)

ROUND ROBIN OU TOURNIQUET

- Le processeur à un instant donné n'exécute qu' un seul processus
  - ⇨ La commutation de contexte entre processus doit être rapide « inférieur au quantum »
- Remarques :
  1. Si le processus a un cycle d'unité centrale inférieur au quantum, alors le processus lui-même libérera l'unité centrale
     - ⇨ L'Ordonnanceur n'aura qu'à traiter le processus suivant dans la file d'attente des processus prêts.
  2. Si le cycle d'unité centrale du processus en exécution courant est supérieur à un quantum, la minuterie se déclenchera et causera une interruption du système d'exploitation.
     - ⇨ Une commutation de contexte sera exécutée et le processus sera placé à la queue de la file d'attente des processus prêts.
     - ⇨ L'Ordonnanceur d'exécution sélectionnera alors le processus suivant dans la file d'attente.

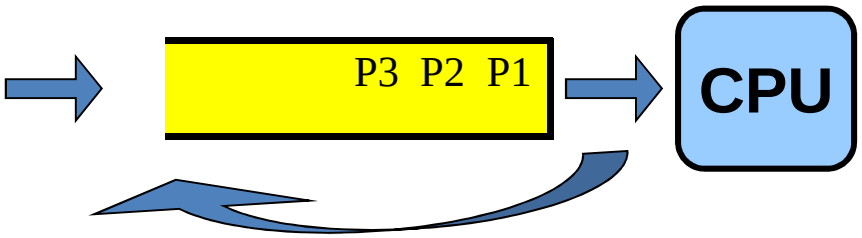
ROUND ROBIN OU TOURNIQUET

Exemple

pour système à temps partagé ou interactif

processus	temps d'arrivée	temps CPU
P1	0	20
P2	0	3
P3	0	7

quantum = 4 ut



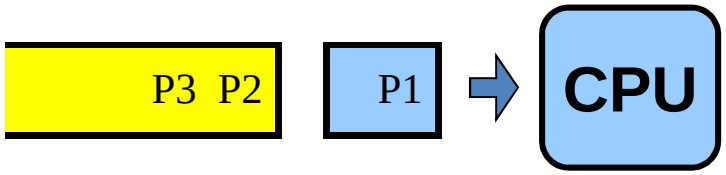
retourne à la fin de la file après un quantum

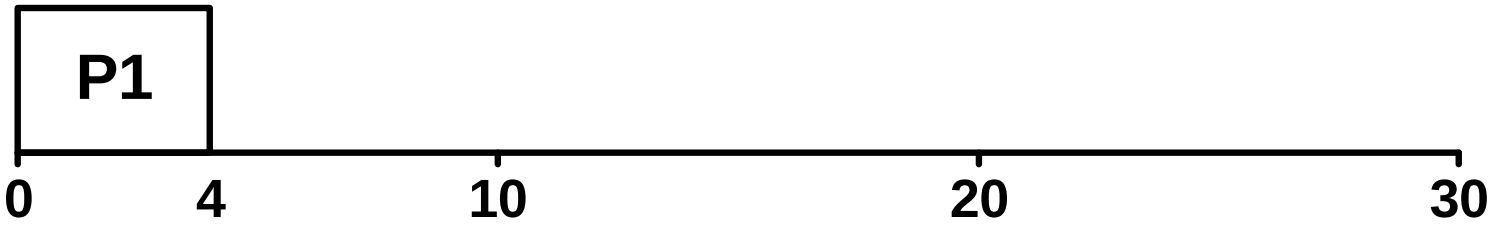
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	20
P2	0	3	3
P3	0	7	7

quantum = 4 ut

temps = 0

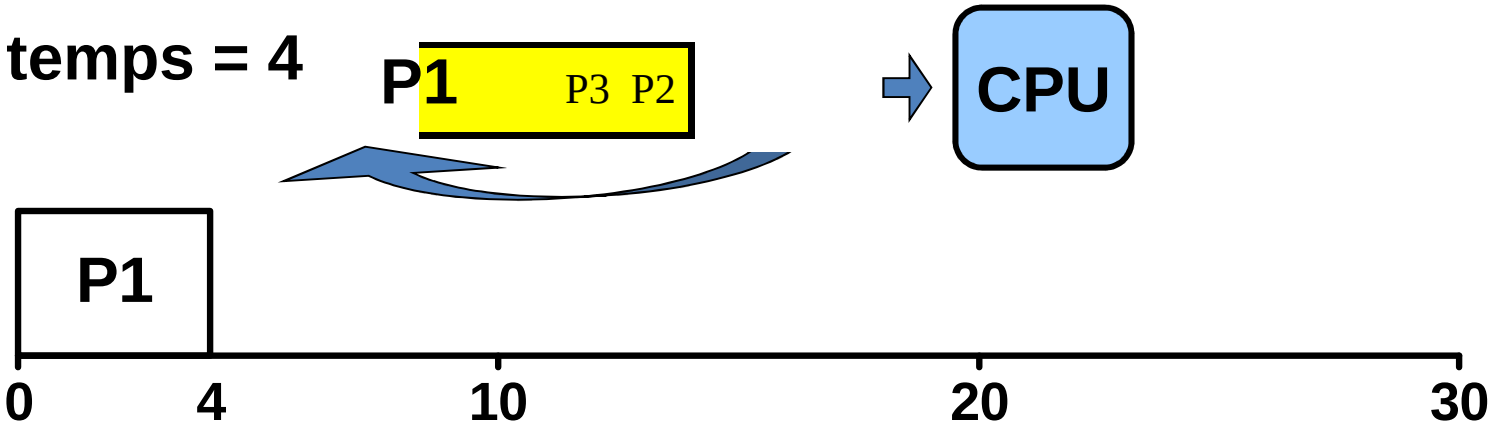




ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	16
P2	0	3	3
P3	0	7	7

quantum = 4 ut

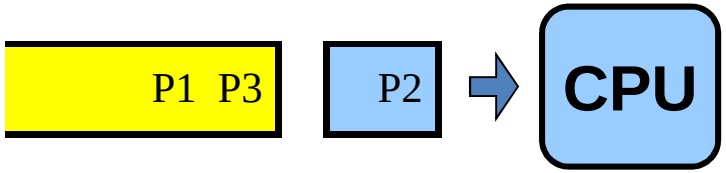


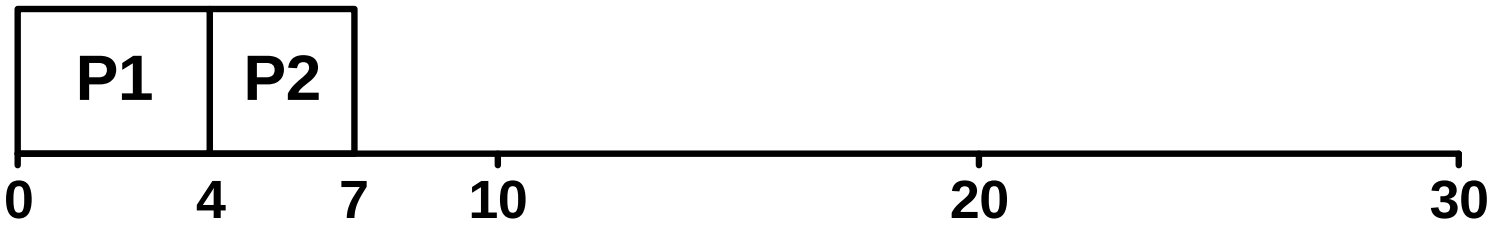
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	16
P2	0	3	3
P3	0	7	7

quantum = 4 ut

temps = 4



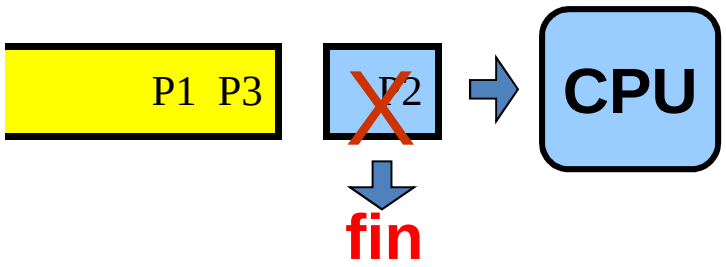


ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	16
P2	0	3	0
P3	0	7	7

quantum = 4 ut

temps = 7

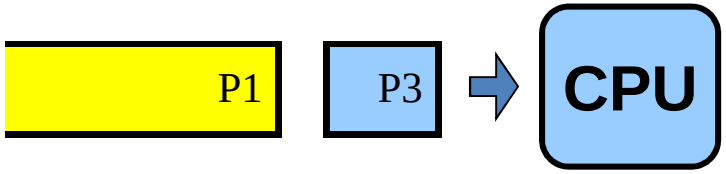


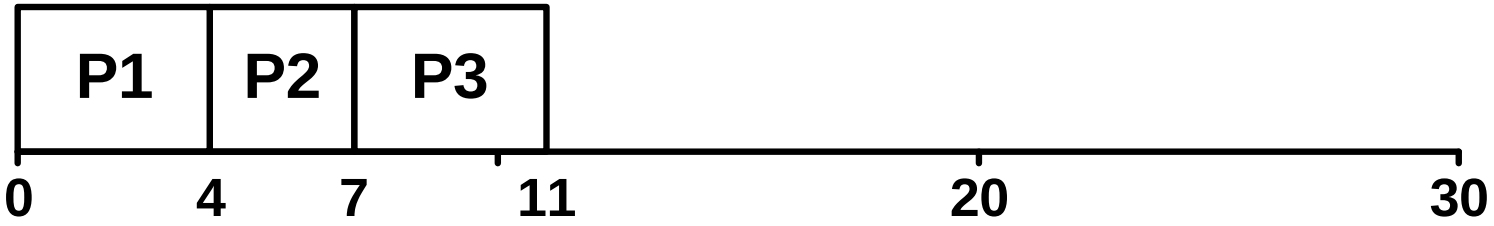
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	16
P2	0	3	0
P3	0	7	7

quantum = 4 ut

temps = 7



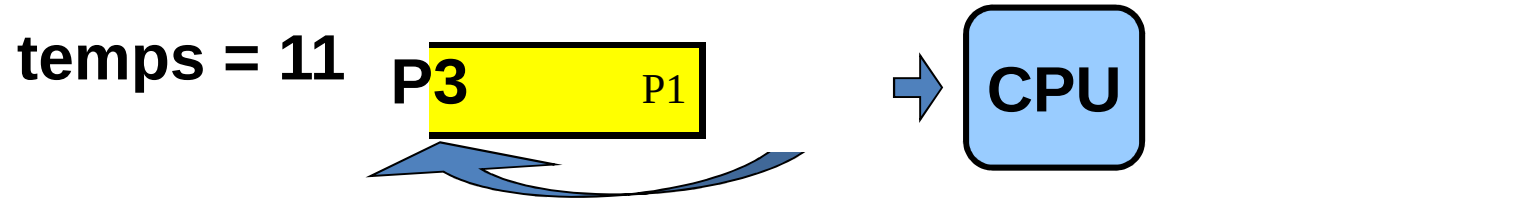


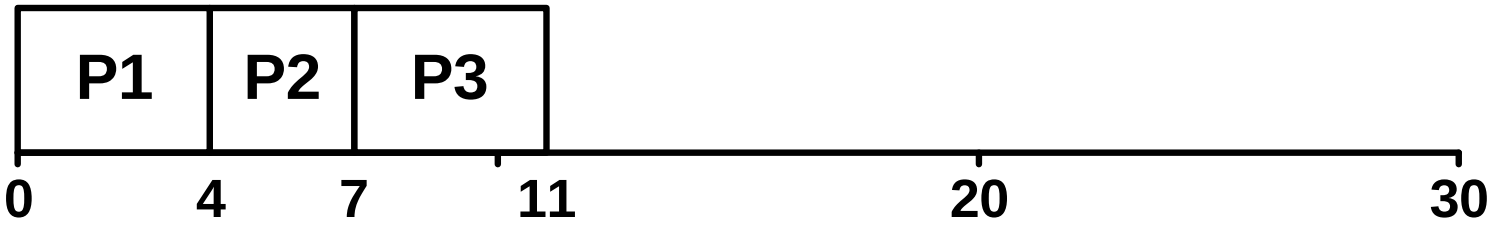
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	16
P2	0	3	0
P3	0	7	3

quantum = 4 ut

temps = 11



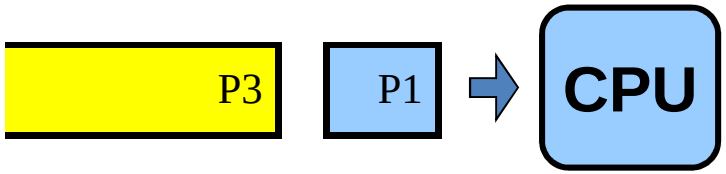


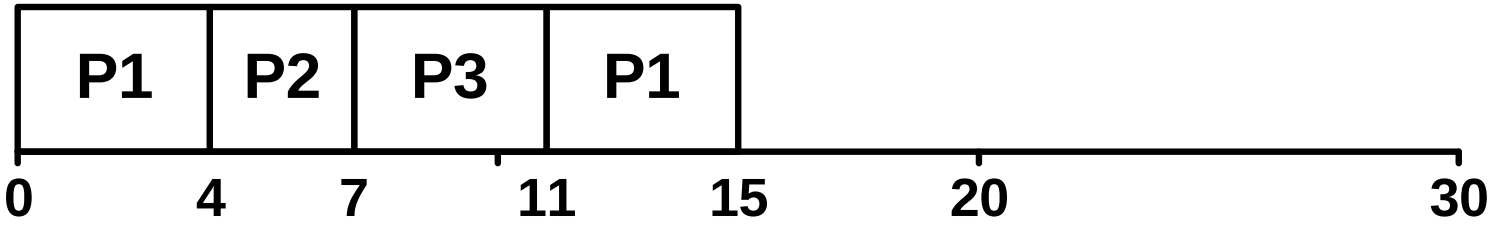
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	16
P2	0	3	0
P3	0	7	3

quantum = 4 ut

temps = 11



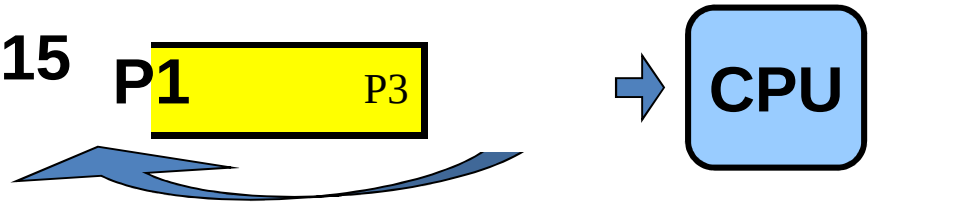


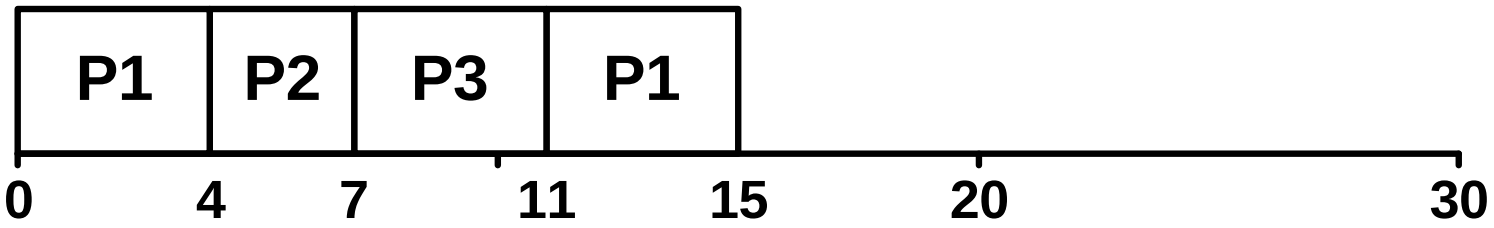
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	12
P2	0	3	0
P3	0	7	3

quantum = 4 ut

temps = 15



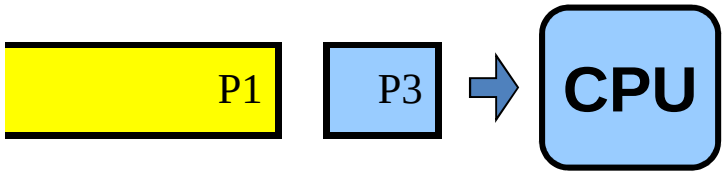


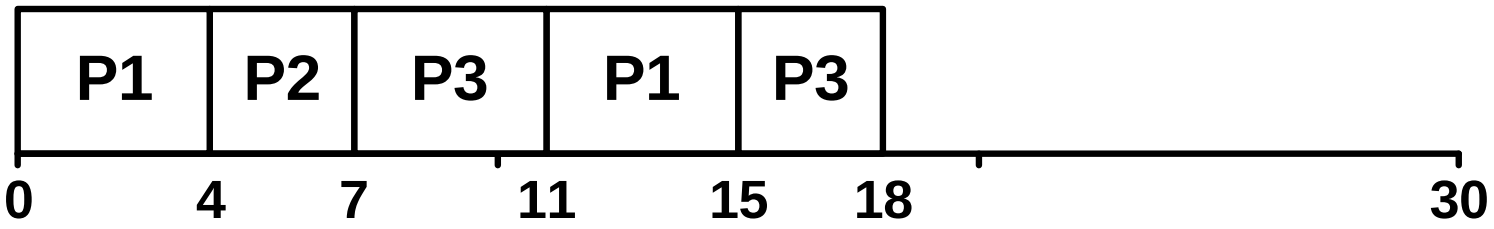
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	12
P2	0	3	0
P3	0	7	3

quantum = 4 ut

temps = 15



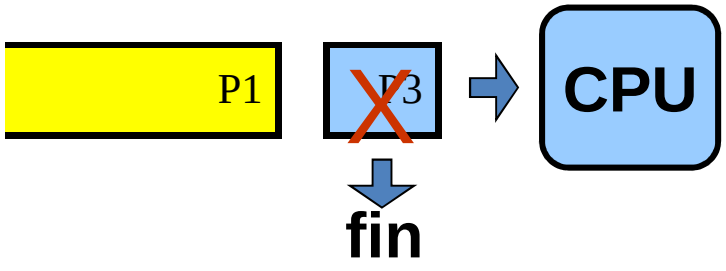


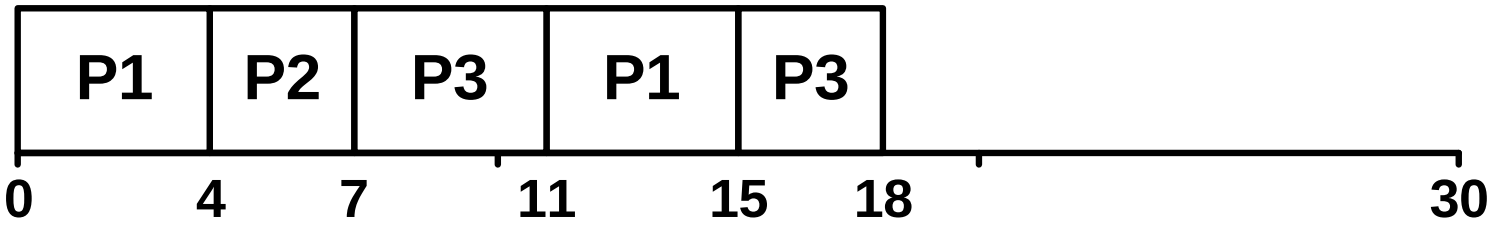
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	12
P2	0	3	0
P3	0	7	0

quantum = 4 ut

temps = 18



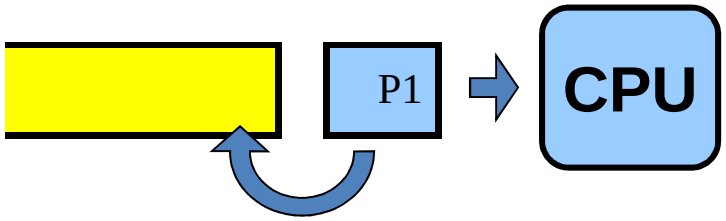


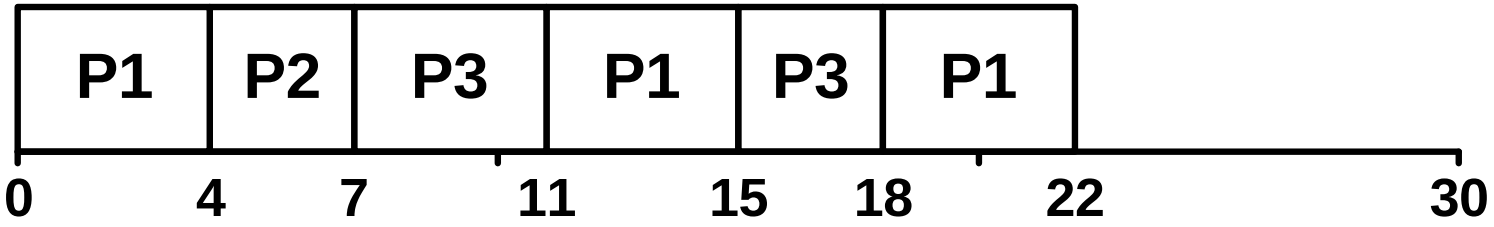
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	12
P2	0	3	0
P3	0	7	0

quantum = 4 ut

temps = 18



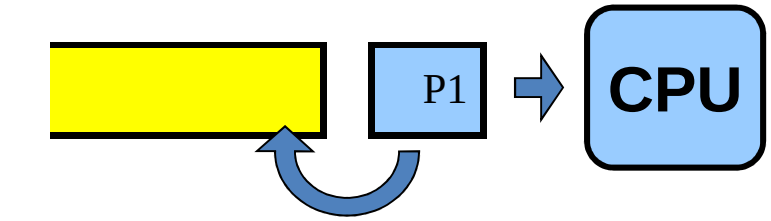


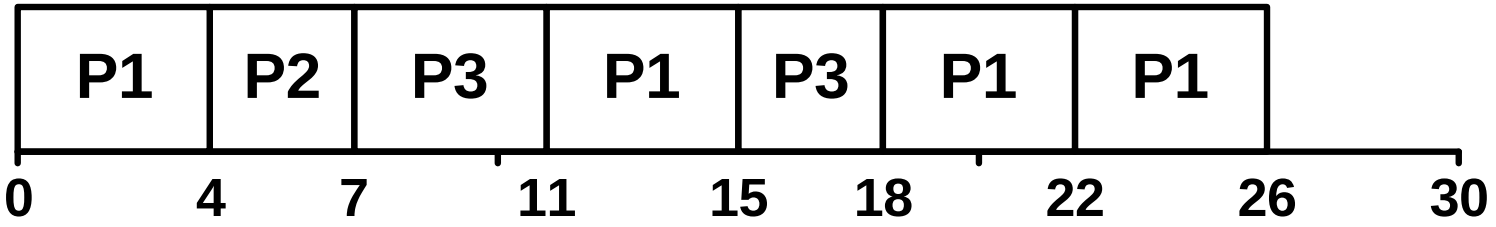
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	8
P2	0	3	0
P3	0	7	0

quantum = 4 ut

temps = 22



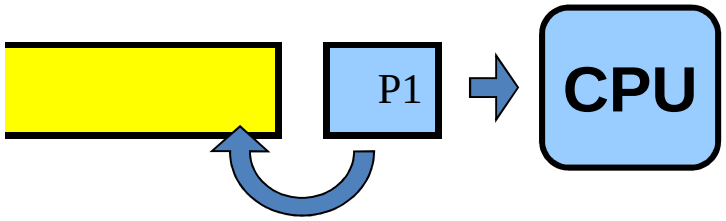


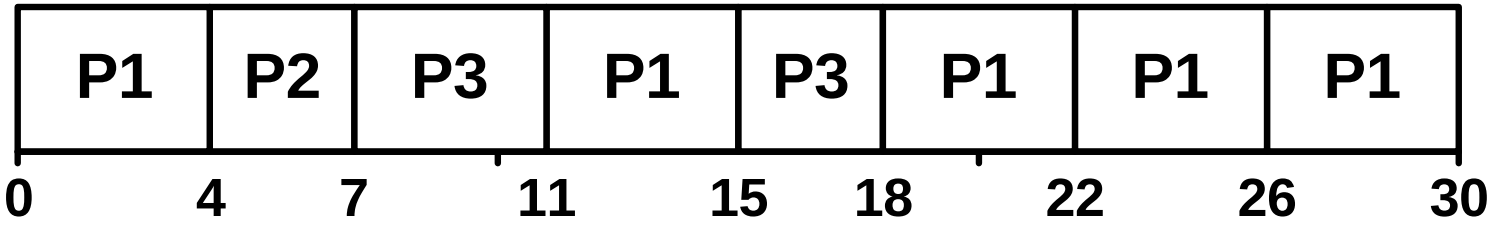
ROUND ROBIN OU TOURNIQUET

processus	temps d'arrivée	temps CPU	temps restant
P1	0	20	4
P2	0	3	0
P3	0	7	0

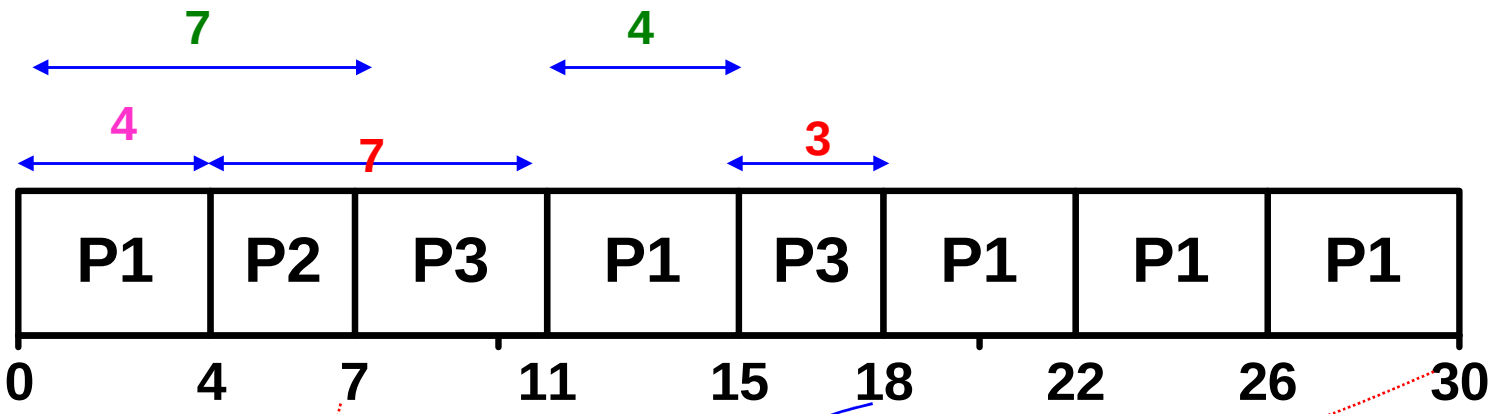
quantum = 4 ut

temps = 26





ROUND ROBIN OU TOURNIQUET

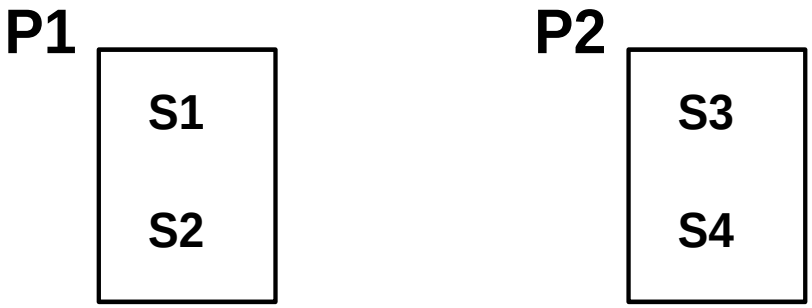


processus	temps de traitement	temps CPU (d'exécution)	temps d'attente
P1	30	- 20	= 10 (7+3)
P2	7	- 3	= 4
P3	18	- 7	= 11 (7+4)
total	55	- 30	= 25
moyenne(÷3)	18.33	- 10	= 8.33

Synchronisation et exclusion mutuelle entre processus

PROCESSUS CONCURRENTS : INTERRELATION (⇔ INTRELACEMENT)

→ Considérer toutes les séquences (états) possibles - aspect temporel



→ Interrelation → considérer toutes les séquences possibles

→ séquences possibles

→ S_1 → S_2 → S_3 → S_4
→ S_1 → S_3 → S_2 → S_4
→ S_1 → S_3 → S_4 → S_2
→ S_3 → S_1 → S_2 → S_4
→ S_3 → S_1 → S_4 → S_2
→ S_3 → S_4 → S_1 → S_2

Bilan

1. Pour des programmes indépendants :
   - ➢ Tous ces entrelacements sont acceptables.
2. Pour des programmes dépendants,
   - ➢ Seulement **`quelques uns de ces`** cas sont acceptables.
   - ➢ Les autres cas **`doivent être évités`**, car ils conduisent à des résultats erronés ou à la catastrophe.
3. Dans la programmation de programmes concurrents (et, sous-entendu, dépendants) :
   - ⇒ Il faut introduire des mécanismes qui permettent de synchroniser les processus et d'éliminer les ***séquences catastrophiques*.**
   - ⇒ La synchronisation implique que la progression d'un processus est éventuellement **contrôlée par un autre**. .

Un exemple

- Considérons la procédure suivante de lecture d'un caractère a depuis le clavier et son affichage.

```
static char a;
void echo( )
      {
        cin     >> a ;   /* saisie du caractère       a
      */
        cout  <<  a  ;  /* affichage du caractère  a
      */
      }
```

Vue globale: exécution concurrente *possible*

- Supposons maintenant deux processus P1 et P2 exécutent cette *même procédure* et partagent la même variable "a"
  - **P1** et **P2** peuvent être *interrompus n'importe* où /* en cours d'exécution */
  - Donc : Si **P1** est *interrompu* après "cin" et que **P2** s'exécute au complet
  - Le caractère affiché par **P1** « après reprise » sera alors celui lu par **P2** !!

❖ **Conclusion :**

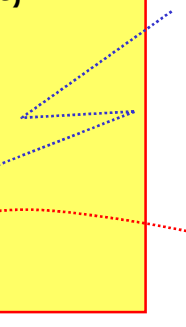
- Le résultat de l'exécution concurrente de P1 et P2 dépend de l`ordre de leur *entrelacement*

Processus P1

```
static char a;
void echo()
 {
    cin >> a;
   cout << a;
 }
```

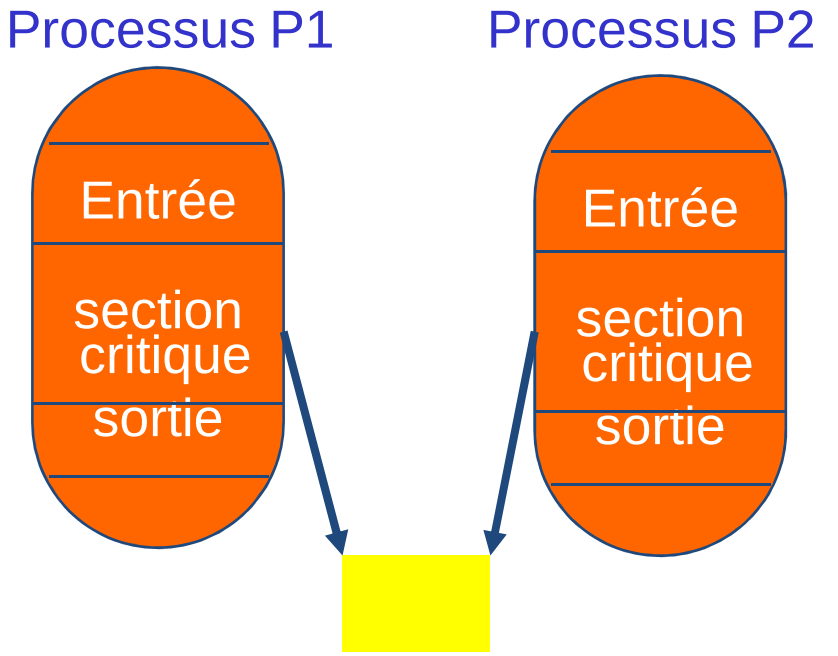
Processus P2

```
static char a;
void echo()
 {
    cin >> a;
    cout << a;
 }
```



Section critique

- Lorsqu'un processus manipule une donnée (ou ressource) partagée, nous disons qu'il se trouve dans une Section Critique (SC) (associée à cette donnée)



- Problème de section critique :
  - Consiste à trouver un algorithme d'*exclusion mutuelle* de processus dans l`exécution de leur SC's afin que le résultat de leurs actions ***ne dépendent pas de l'ordre d'entrelacement*** de leur exécution
- L'exécution des sections critiques doit être mutuellement exclusive:
  - ⇔ à tout instant, un Seul Processus peut exécuter une SC pour une durée donnée (même lorsqu'il y a plusieurs processeurs)
- Remarque :
  - Pour simplifier, dorénavant nous faisons l'hypothèse qu'il n'y a qu'une seule SC dans un programme

Mise en place des sections critiques

Propriétés attendues ou souhaitables d'une solution au problème de l'exclusion mutuelle

1. Exclusion mutuelle : A tout instant, un processus au plus exécute des instructions de sa section critique
2. Absence de blocage : Si plusieurs processus sont en attente d'entrée en section critique (SC) et qu'aucun processus n'est en SC, l'un d'eux doit pouvoir entrer en SC au bout d'un **`temps fini`**
3. Condition de progression : Si un processus se trouve hors de sa SC et hors du protocole contrôlant la SC, alors il ne doit pas ***`empêcher un autre processus`*** d'entrer en SC
4. Absence de famine : Pour un processus voulant rentrer en SC, il existe une borne supérieure au nombre de fois où d'autres processus exécuteront leur SC.
    - La valeur de la borne permet de savoir à quel point la solution est équitable

Sémaphores

[Dijkstra 1965]

Sémaphores

- Un sémaphore S est un type abstrait (définition de DIJKSTRA) qui, sauf pour l'Initialisation, est accessible seulement par ces 2 ***opérations atomiques*** et ***mutuellement exclusives***:
  - ***wait(S)*** : (appelé P(S) dans certains livres)
  - ***signal(S)*** : (appelé V (S) dans certains livres)

wait (S) ⟺ **P (S) - test**

```
while  S ≤ 0  do  no-op
S = S - 1
```

signal (S) ⟺ **V (S) - incrémente**

```
S = S + 1
```

P (*Probeer*) ⇔
décrémenter

« pour demander un élément de la ressource ».

V (*Verhoog*) ⇔
Incrémenter

« libère un élément de la ressource »

Sémaphores d'Exclusion Mutuelle : vue globale sur deux processus

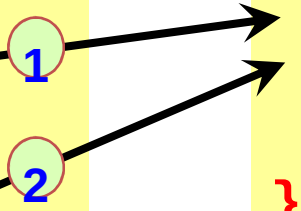
- Initialise S (Mutex = 1) à 1

```
Process Pi:
 repeat
   wait(Mutex);
     {S-- ⇔ Mutex =0}
     Entrée en SC
          |
          |
   signal(Mutex);
    {Mutex=1
     Réveil }
     SR
 forever
```

```
Process Pj:
 repeat

 wait(Mutex);
     { Mutex =0
       Mise en attente
}

       Entrée en SC
          |
          |
 Signal(Mutex);
   {Mutex=1
    Réveil }
    SR
 forever
```

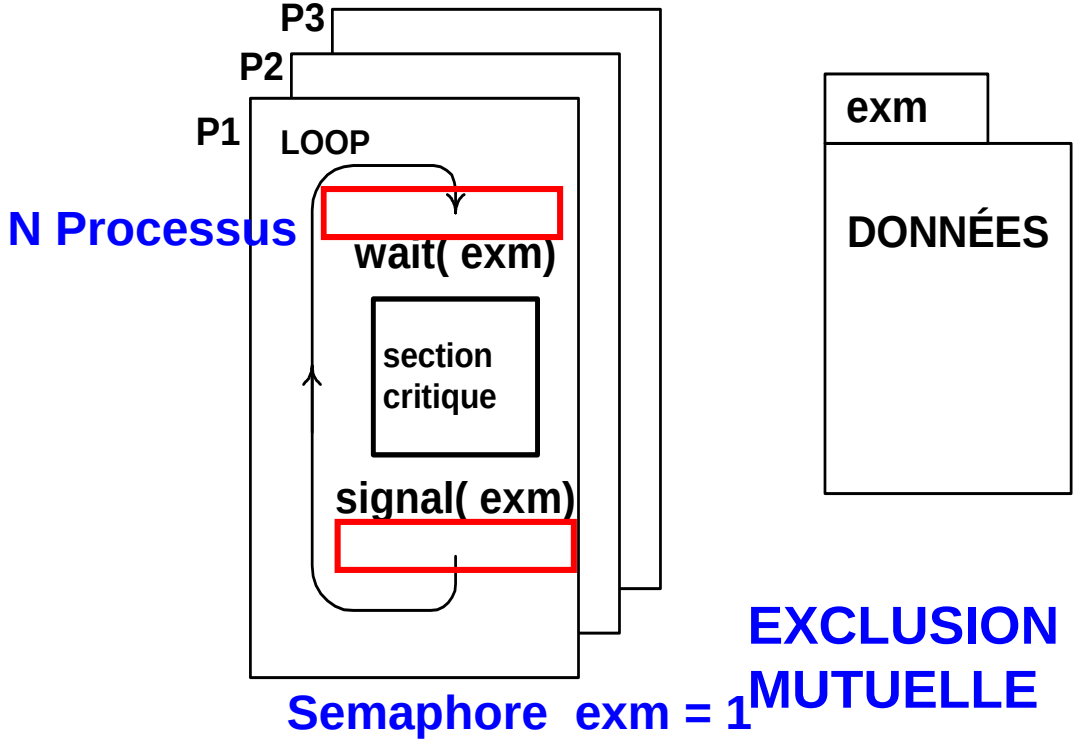


Peut être facilement généralisé à plus de deux processus

Sémaphores d'Exclusion Mutuelle : pour N processus

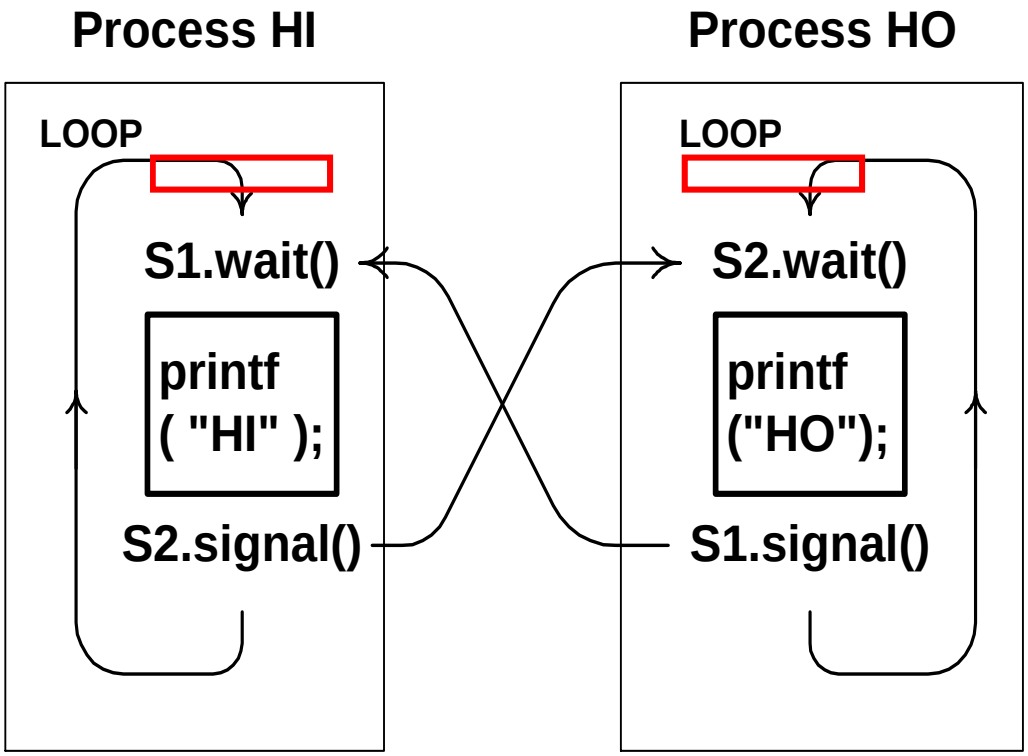
- Le sémaphore exm peut être considéré comme **un verrou** qui assure qu'un seul processus accède aux données à la fois (i.e., exécute sa section critique).

  - Etat initial : **exm =1** et aucun processus ne s'exécute dans sa section critique.
  - Le premier processus qui effectue l'opération **wait(exm)** pourra donc accéder aux données et mettra la valeur de **exm à 0.**
  - Tout autre processus effectuant l'opération **wait(exm)** sera donc mis en attente (sera bloqué).
  - Lorsque le premier processus quitte la section critique, il effectue l'opération **signal(exm).** (⇔ incrémente le sémaphore (**exm=1**), ce qui permet de débloquer un processus en attente et un seul.



ALTERNANCE FORCÉE
« synchronisation »

Semaphore S1 = 1
Semaphore S2 = 0



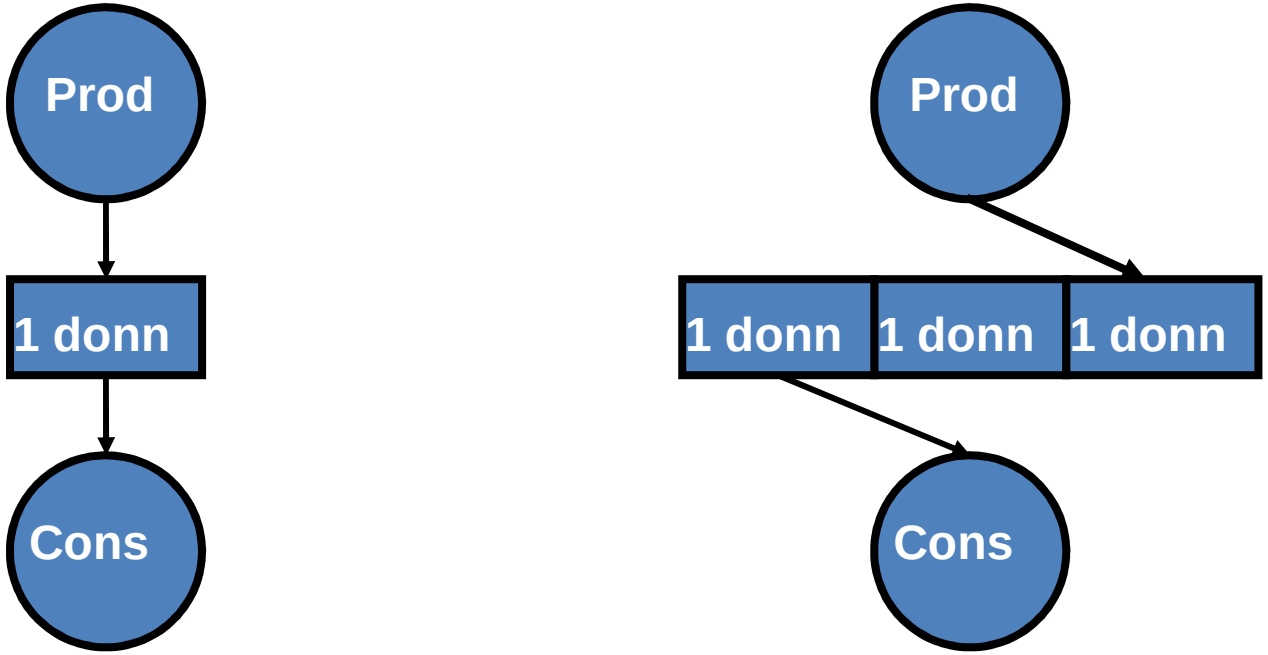
Problèmes classiques de synchronisation

- Applications classiques des sémaphores

1. Tampon borné (producteur-consommateur)
2. Écrivains - Lecteurs
3. Problème du dîner des philosophes

Le pb du producteur - consommateur

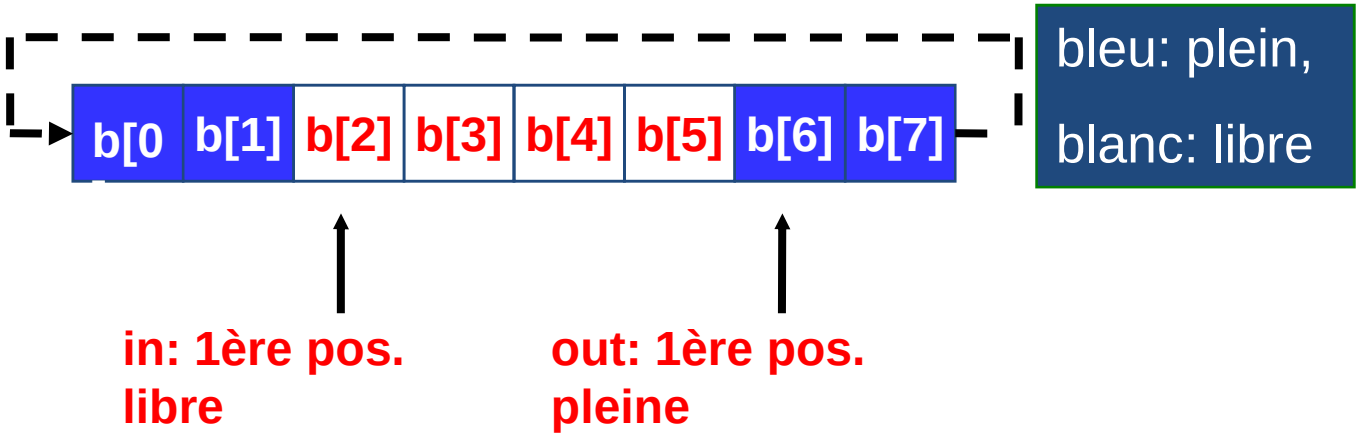
- Un problème classique dans l ’étude des *processus communicants*
  - un processus ***producteur*** produit des données (p.ex. des enregistrements d’un fichier) pour un processus ***consommateur***
  - **Tampons de communication**



- **Si le tampon est de longueur 1 ⇨le producteur et consommateur doivent forcement aller à la même vitesse**
- **Des tampons de longueur plus grandes permettent une certaine indépendance.**
  - **P.ex. à droite le consommateur a été plus lent**

Le tampon borné (bounded buffer)
une structure de données fondamentale dans les SE

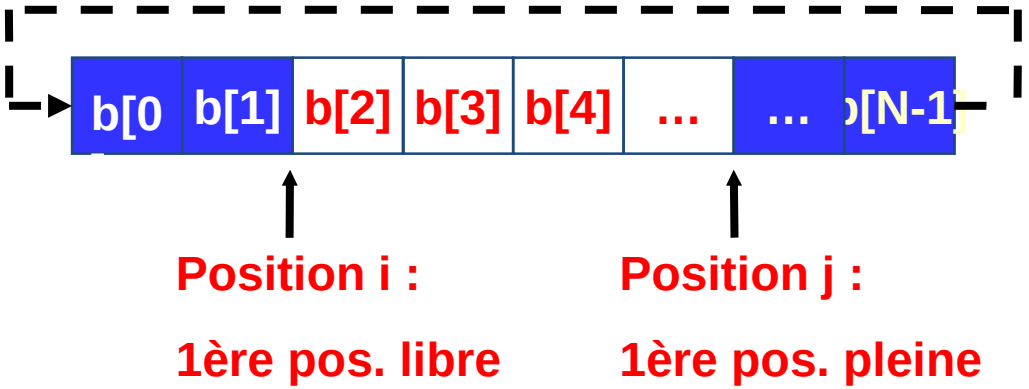
- Utilisation du tampon borné dans le schéma Producteur - Consommateur :
  - Le tampon borné se trouve dans la mémoire partagée entre consommateur et usager



- **Pb de synchronisation entre proc pour le tampon borné :**
  - Étant donné que le producteur et le consommateur sont des processus indépendants, des problèmes pourraient se produire en permettant accès simultané au tampon
  - Les sémaphores peuvent résoudre ce problème

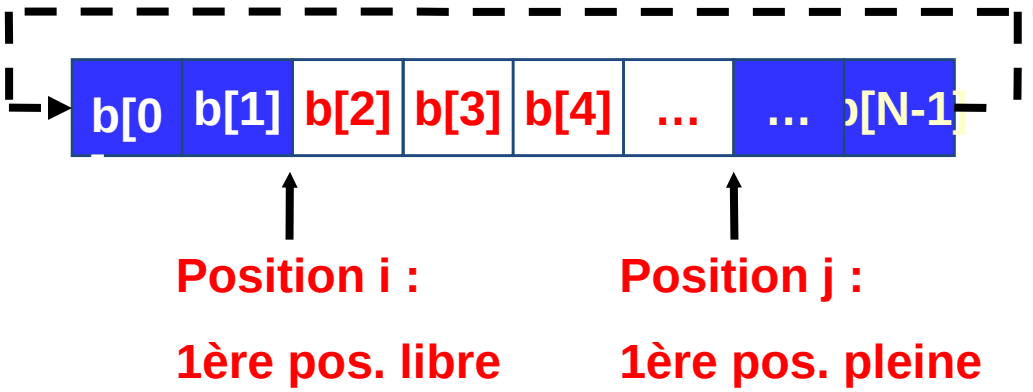
Programmation du schéma producteur/consommateur –1-

- **Hypothèses de base :** On considère ici ***deux processus*** communiquant par un tampon de ***N*** cases.
  - ☞ Le producteur dépose des messages dans le tampon dans la case pointée par ***l'index i.***
  - ☞ Le consommateur prélève les messages dans la case pointée par ***l'index j.***
  - ☞ Le tampon est géré selon un mode FIFO circulaire en consommation et en dépôt, c'est-à-dire :
    - ⇔ Le producteur dépose les messages depuis la ***case 0*** jusqu'à la ***case N-1***, puis revient à la ***case 0***
    - ⇔ Le consommateur prélève les messages depuis la ***case 0*** jusqu'à la ***case N-1***, puis revient à la **case 0.**



Programmation du schéma producteur/consommateur –2-

- Pour qu'aucun message ne soit perdu, les trois règles suivantes doivent être respectées :
  1. Un producteur ne doit pas produire si le tampon est plein
  2. Un consommateur ne doit pas faire de retrait si le tampon est vide
  3. Producteur et consommateur ne doivent jamais travailler dans une même case



Programmation du schéma producteur/consommateur –3-

- **Dans ce problème deux types de ressources distinctes peuvent être mises en avant :**
  - **Le producteur consomme des cases vides et fournit des cases pleines**
  - **Le consommateur consomme des cases pleines et fournit des cases vides**
- **Donc deux types de ressources :**
  - **ressource cases vides et des ressource cases pleines.**
  - **Le problème peut maintenant se rapprocher d'un problème d'allocation de ressources critiques tel que nous l'avons vu précédemment.**
- **Principe de la solution de programmation :**
  - **On associe donc un sémaphore à chacune des ressources identifiées et on initialise ce sémaphore au nombre de cases respectivement vides ou pleines initialement disponibles (N et 0).**
  - **On a donc deux sémaphores :**
    - **VIDE initialisé à N**
    - **PLEIN initialisé à 0.**

Programmation du schéma producteur/consommateur –4-

- **Un producteur :**
  - **Va s'allouer une case vide par une opération Wait(VIDE),**
  - **Remplir cette case vide (⇔ et de ce fait générer une case pleine).**
  - **Va signaler cette nouvelle case pleine par une opération Signal(PLEIN),**
    - **Cette opération réveillera éventuellement le consommateur en attente d'une case pleine.**
- **Un consommateur**
  - **Va s'allouer une case pleine par une opération Wait(PLEIN),**
  - **Vider cette case pleine (⇔ et de ce fait générer une case vide).**
  - **Va signaler cette nouvelle case vide par une opération Signal(VIDE),**
    - **Cette opération réveillera éventuellement le producteur en attente d'une case vide.**

Programmation du schéma producteur/consommateur –5-

Producteur

Si *il y a au moins une case libre*

alors

déposer le message

prévenir le consommateur

sinon

attendre

fsi

allocation de ressources cases vides
Wait (Sémaphore VIDE)

Une ressource case pleine disponible
Signal (Sémaphore PLEIN)

allocation de ressources cases pleines
Wait (Sémaphore PLEIN)

Une ressource case vide disponible
Signal (Sémaphore VIDE)

Consommateur

Si *il y a au moins une case pleine*

alors

prendre le message

prévenir le producteur

Sinon

attendre

fsi

Programmation du schéma producteur/consommateur –6-

- **Initialisation des sémamphores :**
  - **Sémaphore VIDE initialisé à N : Init(Vide, N)**
  - **Sémaphore PLEIN initialisé à 0 : Init(Plein, 0)**

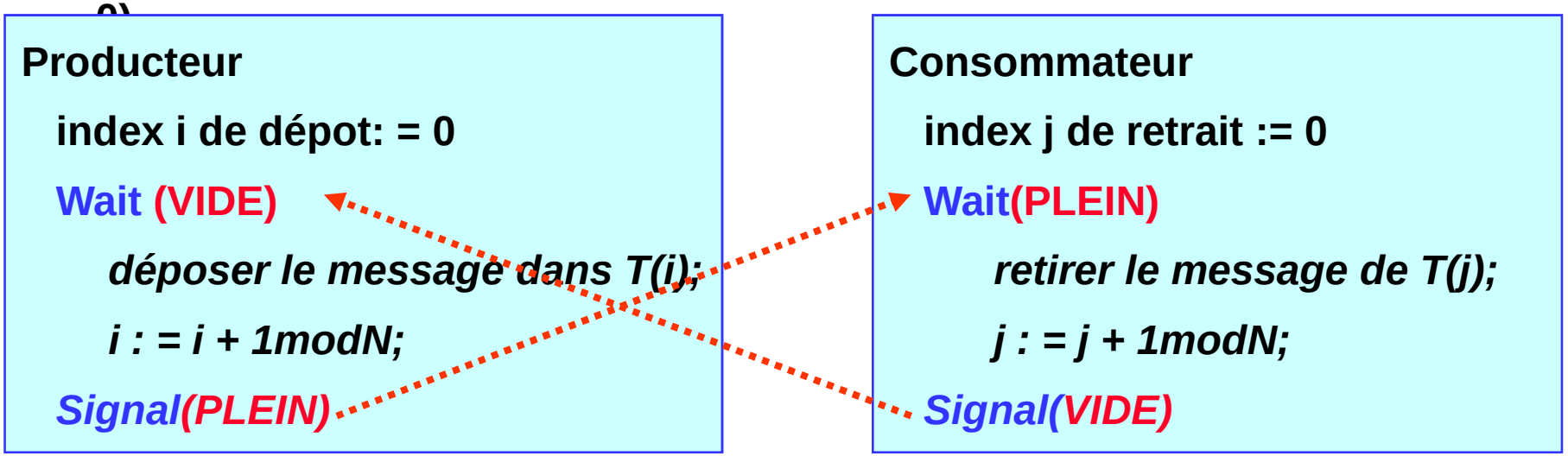


Schéma producteur / consommateur

Programmation du schéma producteur/consommateur –7-

- ***Bilan : Deux Sémaphores de synchronisation : VIDE et PLEIN***
- **Remarque :**
  - **Les sémaphores VIDE et PLEIN ne font pas l'EM**
  - **Un sémaphore PLEIN pour synchroniser producteur et consommateur sur le nombre de cases pleines dans le tampon (nombre d'éléments consommables)**
  - **Un sémaphore VIDE pour synchroniser producteur et consommateur sur le nombre de cases libres**
- Pour assurer l'exclusion mutuelle sur l'accès au tampon, il faut un autre sémaphore S d'Exclusion Mutuelle.

Programmation du schéma producteur/consommateur –8-

Initialization:

S.count=1; *//excl. mut.*

PLEIN.count=0; *//esp. pleins*

VIDE.count=N; *//esp. vides*

```
Producteur
repeat
    Produire v;
    wait(VIDE);
      wait(S);
        append(v);
      signal(S);
    signal(PLEIN);
forever
```

```
Consommateur
Repeat
    wait(PLEIN);
      wait(S);
        w =take();
      signal(S);
    signal(VIDE);
    Consomme (w);
forever
```

```
append(v):
  b[in]=v;
  In ++ mod N;
```

```
take( ):
  w=b[out];
  Out ++ mod N;
  return w;
```

■ Sections critiques