Chapitre I: GESTION DES E/S

Objectif:

Assurer la gestion et le contrôle des opérations d'entrées et de sortie

Dans cette partie, nous aborderons les thèmes suivants:

1. Interface matériel
2. Interface application

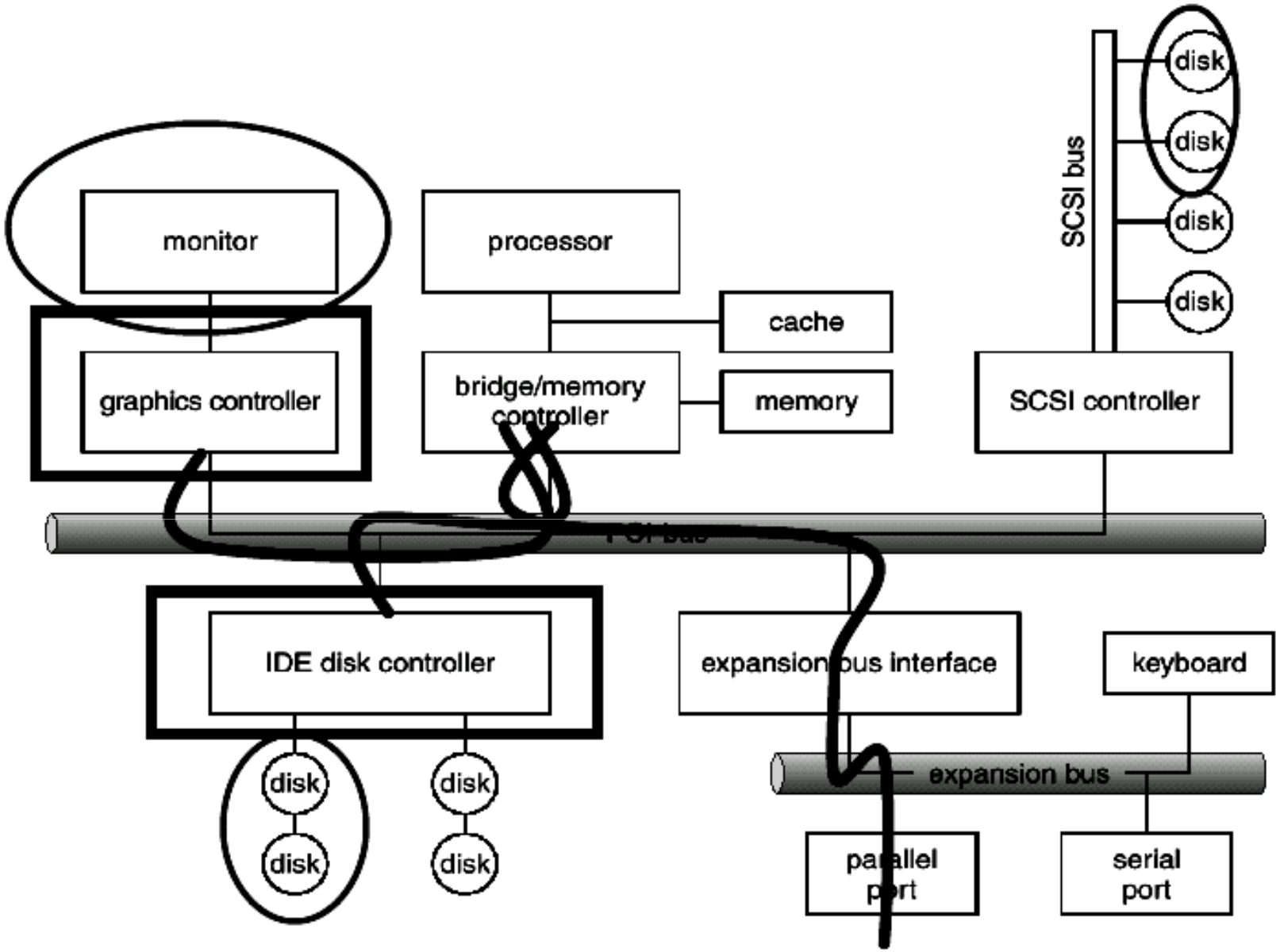
Thème 1. Interface matériel

Les points suivants seront traités:

1. Description des éléments hardwares
2. Interaction processeur/contrôleur
   - Méthode 1. principe du handshaking
   - Méthode 2. Contrôleur DMA

Point 1. Description des éléments hardware

- Un système informatique dispose d'une grande diversité de dispositifs E/S
- Les contrôleurs de chaque dispositif communiquent avec le processeur central via des signaux électriques.
- Ces signaux sont transportés via câble


monitor
processor
cache
graphics controller
bridge/memory controller
memory
SCSI bus
disk
disk
disk
disk
SCSI controller
IDE disk controller
expansion bus interface
keyboard
disk
disk
disk
disk
expansion bus
parallel port
serial port

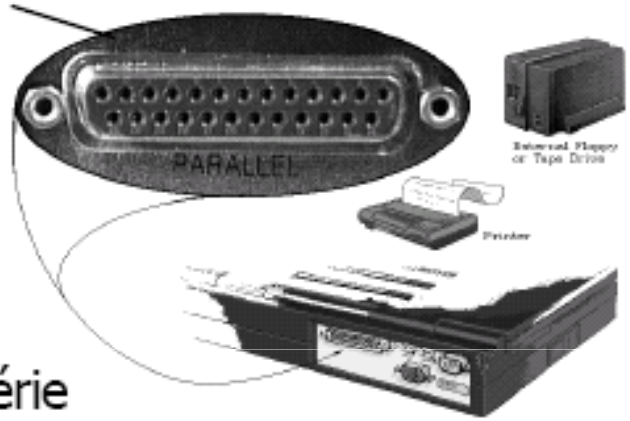
Si le périphérique est extérieur au système compris dans la tour, on retrouve au niveau du PC une interface d'accès pour le câble qu'on appelle un port. Au transparent suivant, on décrit les deux types de port que l'on distingue habituellement.

Nous allons envisager les divers composants intervenant dans ces schémas :

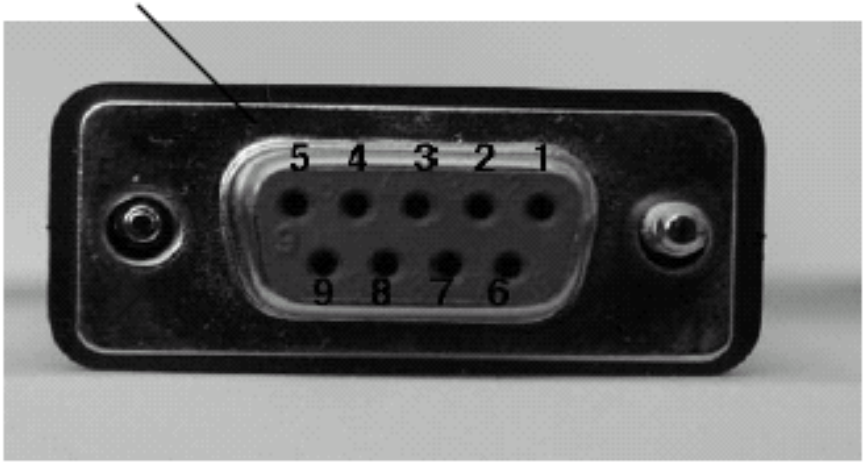
- Les ports
- Les bus de communication
- Les contrôleurs

Port série et port parallèle

Port parallèle

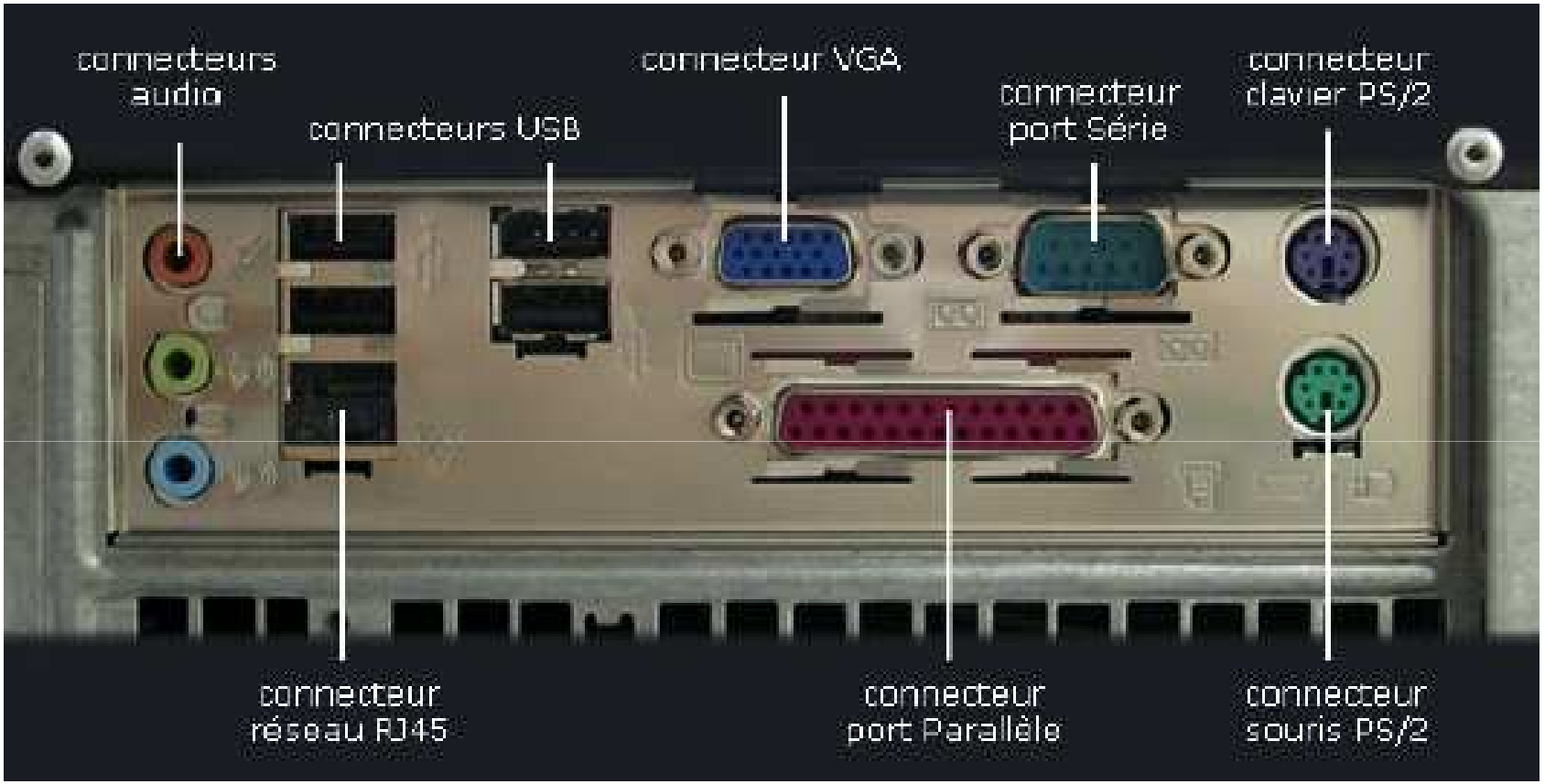


Port série



Port série :

- Est capable de supporter bcp d'applications, ne possède que peu de contraintes au niveau des fils de connexion, et n'utilise que quelques signaux
- Les signaux sont transmis les uns après les autres sur les fils (économie >< décodeur)


connecteurs audio
connecteurs USB
connecteur VGA
connecteur port Série
connecteur clavier PS/2
connecteur réseau RJ45
connecteur port Parallèle
connecteur souris PS/2

Familles de bus

Bus intérieurs & bus extérieurs (ou bus d'expansion)

• Le bus extérieur est un bus de communication plus lent, qui permet une connexion avec les périphériques extérieurs.

Contrôleur

Il s'agit d'un:

• Composant électronique qui agit sur le bus, le port, le périphérique

• Il contient un processeur, un microcode, un volume de mémoire privée

Point 2. Interaction processeur/contrôleur

Matériel disponible:

• Le contrôleur possède un ou plusieurs registres pour contenir les données et les signaux de contrôle

Principe:

• Le processeur communique via une combinaison de valeurs pour les bits contenus dans les registres

Les registres disponibles

Le système d'E/S comporte généralement 4 registres, à savoir :

– le registre d'état,

– le registre de contrôle,

– le registre d'entrées de données,

– le registre de sortie de données

Chaque registre peut contenir plusieurs bits.

Nous abordons deux méthodes d'interaction :

1. celle basée sur le principe du handshaking et
2. celle à la base du contrôleur DMA.

Méthode 1

1. Principe du handshaking

Envisageons l'exemple suivant.

Objectif :

• écrire le contenu de mots sur le support

• 4 bits sont alors nécessaires pour coordonner cette écriture, soit :

– Bit occupé du registre d'état

– Bit commande prête du registre de commande

– Bit écriture du registre de commande

– Bit erreur du registre d'état

Principe du handshaking suite….

1. La machine lit de façon répétée le bit occupé, jusqu'à ce qu'il soit égal à 0.

2. La machine positionne le bit écriture du registre de commandes et écrit un octet dans le registre de sortie de données.

3. La machine positionne le bit commande prête.

4. Dès qu'il remarque que le bit commande prête est positionné,

le contrôleur positionne le bit occupé.

5. Le contrôleur lit le registre de commandes et voit la commande d'écriture. Il lit le registre de sortie de données pour obtenir l'octet et effectue l'entrée/sortie vers le périphérique.

6. Le contrôleur efface le bit commande prête, efface le bit erreur du registre d'état pour indiquer l'E/S vers le périphérique a réussi, et efface le bit occupé pour signaler qu'il a terminé.

Principe du handshaking: Propriété

1. La machine lit de façon répétée le bit occupé, jusqu'à ce qu'il soit égal à 0.

On observe ici un problème d'attente active

• Dont la durée peut être relativement longue, ce qui implique une perte de temps CPU

• Solution : utiliser une interruption

Remarque : Utiliser une interruption

Motivation :

• permettre au contrôleur de notifier le processeur que le périphérique qu'il coordonne est prêt.

Mise en oeuvre :

• Le processeur possède une ligne de requête d'interruption.

• Sur cette ligne, le processeur détecte tout signal

• Si le processeur est dans un état interruptible, alors une routine de gestion sera exécutée pour réagir face à ce signal électrique.

5. Le contrôleur lit le registre de commandes et voit la commande d'écriture. Il lit le registre de sortie de données pour obtenir l'octet et effectue l'entrée/sortie vers le périphérique.

En termes de temps de traitement CPU, l'opération est une procédure coûteuse

• Solution : utiliser un contrôleur de type DMA

A chaque octet, l'opération doit être ré-éditée, et nécessite les mêmes interventions de la part du processeur.

Méthode 2. Contrôleur DMA

1. Le pilote du périphérique est exécuté. Le nombre d'octets à transférer et l'adresse en mémoire principale sont déterminés.

2. Le contrôleur du disque initialise un transfert DMA : il écrit un bloc de commandes DMA en mémoire.

3. Le contrôleur DMA via le bus reçoit les octets qu'il transfère vers la mémoire principale

4. A la fin du transfert, le contrôleur DMA interrompt l'unité centrale

DMA signifie **Direct Memory Access**.

Le pilote de périphérique est le nom du module de l'OS dont la tâche est d'interagir avec le contrôleur du périphérique afin de lui communiquer les informations pour l'échange.

Conclusion : aspects hardware

Les concepts principaux sont:

– Un bus

– Un contrôleur

– Un port d'entrée/sortie et ses registres

– La relation d'échange entre la machine et un contrôleur de périphérique

– L'exécution de ce protocole d'échange dans une boucle de scrutation ou via des interruptions

– La décharge de ce travail vers un contrôleur DMA pour les transferts volumineux

Thème 2. Interface application

Motivation :

• Chaque contrôleur est particulier, en termes de :

– opérations qu'il supporte,

– protocoles d'interaction,

– Contenu des registres,…

• OS doit tenir compte de ces particularités pour pouvoir réaliser des opérations d'E/S vers le périphérique correspondant.

• Pour cela, il doit respecter les principes de

- Abstraction
- encapsulation
- mise en couches OS

Principes pour construire un OS supportant différents périphériques

- **Abstraction des différences**: mise en oeuvre via un

classement des périphériques en types

- **Encapsulation**: définition d'une interface contenant un

ensembles de fonctions standardisées pour chaque type. Les différences entre périphériques d'une même classe sont encapsulées dans un module

- **Mise en couche**: utilisation d'un pilote de périphérique

(device driver), en dessous du module des périphériques de ce

type.

Point 1. Pilote de périphérique

Motivation:

• Permettre une abstraction des différences entre contrôleurs de périphériques d'un même type pour les couches supérieures de l'OS

Conséquence:

• Portabilité OS

• Simplification du travail du programmeur du noyau et du constructeur périphérique

On dit d'un OS qu'il est portable si il est indépendant (dans ses couches supérieures) du hardware sur lequel il est exécuté.

Une simplification… en effet : à l'ajout d'un périphérique au système informatique, seul un pilote de périphérique nouveau devra être défini.

Celui-ci, pour pouvoir travailler avec l'OS devra donc supporter les fonctions standardisées du type de périphérique correspondant et fournir du code capable de traduire ses fonctions en signaux à émettre vers le contrôleur.

Point 2. Types de périphérique différences

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only readÐwrite	CD-ROM graphics controller disk

Dans ce tableau, est présenté les différents aspects qui comptent pour permettre de différencier des familles de périphériques. Dans la colonne du milieu sont indiquées pour chacun de ces aspects, les différents types qu'on peut observer.

Par exemple, le terminal et le disque sont tous deux des périphériques orientés transfert de données, mais l'un travail au niveau du transfert d'un caractère, alors que pour l'autre, le transfert se fera bloc par bloc.

On distingue donc les aspects suivants :

•Type des données qui sont transférées

•Le type d'accès au périphérique, celui-ci est synchrone lorsque le système d'exploitation bloque l'exécution d'un ou l'autre processus en l'attente des résultats qui seront fournis.

•Partageable ou dédié, si oui ou non le périphérique peut être partagé par différents processus en même temps.

•La vitesse de fonctionnement et

•La direction de l'opération d'E/S.

Types de périphériques

On distingue donc les familles de périphériques suivantes :

1. Périphériques par blocs
2. Périphériques par caractère
3. Périphériques réseau
4. Horloge et minuterie
5. Périphériques aux E/S bloquantes ou non-bloquantes

Famille 1. Périphérique par blocs

Objectif :

Décrire les fonctions standardisées qui permettront un accès aux périphériques orientés blocs Hypothèses de travail :

– les commandes telles que **read** et **write** sont supportées.

– Le périphérique est à accès direct.

– Le périphérique possède une commande « seek »

– Les applications accèdent à cette interface habituellement via une interface de systèmes de fichiers.

Famille 2. Périphérique par caractères

Objectif :

• Il s'agit de périphériques produisant des données spontanées, tel le clavier.

Hypothèses de travail :

– Les opérations de base qui doivent être supportées sont :

  • Application de recevoir un caractère,

  • Application d'envoyer un caractère.

– Au dessus du module de gestion de périphérique par caractères, il sera nécessaire de construire des bibliothèques pour un accès ligne par ligne, avec des services de tamponnage ou d'édition

Famille 3. Périphérique réseau

Objectif :

• Il s'agit de périphériques assurant l'adressage des E/S du réseau, diffèrent des E/S de disque.

• OS offre une interface réseau.

Les services suivants sont offerts:

– Connecter deux applications distantes

– Ecouter les demandes des applications distantes pour une connexion

– Envoyer et recevoir des paquets entre applications distantes

– Supporter l'implémentation des serveurs

Famille 4. Horloge et minuterie

Hypothèses de travail :

• Les opérations de base à supporter sont :

– Donner l'heure actuelle

– Donner le temps écoulé

– Fournir une minuterie permettant de déclencher une opération X à l'heure T.

Famille 5. Entrées/sorties bloquantes ou non-bloquantes

Objectif :

• Réaliser une dichotomie entre les E/S bloquantes et non-bloquantes (asynchrones).

• On dit d'un appel système qu'il est bloquant, si l'exécution de l'application demandeur est suspendue jusqu' à la fin du traitement.

– Dans le cas de périphériques tels que le clavier ou la souris, il s'agit de périphérique avec E/S non bloquantes : les informations arrivent alors que les précédentes sont en cours de traitement.

Point 3. Services offerts par l'interface

- Ordonnancement des requêtes **(scheduling)**
- Mise en tampon **(buffering)** des opérations d'entrée/sortie
- Mise en cache **(caching)** des opérations d'entrée/sortie
- Mise en attente **(spooling)** des opérations d'entrée/sortie
- Réservation du périphérique
- Gestion des erreurs

Scheduling

Motivation:

- Les requêtes arrivent au cours du temps et sont mises en buffer, mises en tampon, au niveau du pilote.

Objectif :

- Programmer les réponses aux requêtes en fonction de politiques bien particulières.
  - Exemple d'ordonnancement : ordonnancement des requêtes pour l'accès aux blocs physiques du disque.

Buffering

Motivation :

- Un tampon est une zone de mémoire pour le stockage des informations pendant leur transfert. On envisage les transferts :
  - entre deux périphériques
  - entre un périphérique et une application

Objectif

- Permettre de résoudre des problèmes :
  - de vitesse entre producteur et consommateur
  - périphériques ne travaillent pas avec les mêmes volumes de transfert
  - Gestion de la sémantique copie E/S d'une application

Caching

Motivation :

- La cache est une zone de mémoire d'accès rapide pour conserver des copies des informations

Objectif

- Offrir le service suivant : vérification du contenu de la cache avant tout accès à la zone de mémoire auxiliaire.

Spooling & réservation

Motivation :

- Les demandes s'intercalent pendant le traitement complet d'une demande précédente.

Objectif :

- Ne pas mélanger les contenus
- Parfois gérer par un processus "démon", responsable de répondre aux demandes asynchrones
- Permettre la réservation d'un périphérique exclusivement à un ensemble de processus.

Gestion des erreurs

Motivation

- Le transfert des données, les commandes d'E/S, les périphériques peuvent tous être victime de défaillances hardware ou software. Le contrôleur peut avertir l'application de ces défaillances.

Objectif :

- Le pilote de périphérique doit réagir face à ces avertissements.
- Dans le cas d'une défaillance temporaire: OS réagit en fonction de l'erreur
  - en répétant la commande
  - en envoyant un message d'erreur

Mécanisme des interruptions

Au sommaire...

Introduction

Interruptions matérielles

Interruptions logicielles

Direct Memory Access

Introduction

INT / DMA …

INT = Interruptions

DMA = Direct Memory Access

- Accès direct à la mémoire

Mécanisme d'interruption

Déroutements

- Lors de l'exécution des instructions

Interruptions matérielles

- Gestion des périphériques

Interruptions logicielles

- Appels système

Déroutements :

Débordement numérique

Division par zéro

Non-reconnaissance d'une instruction

Accès à la mémoire illégale

Interruptions matérielles

Demande de données d'un périphérique

Alerte provenant du matériel (batterie faible)

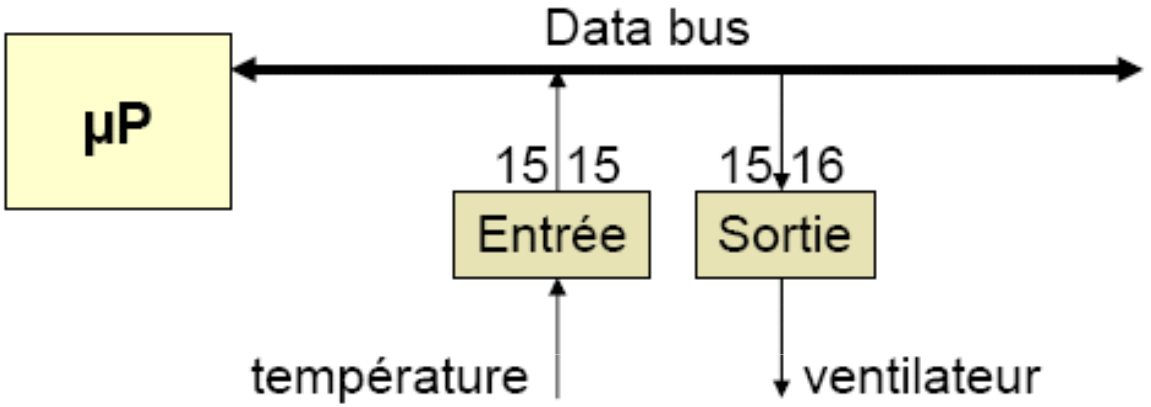
Interruptions logicielles

Appels de fonctions du système d'exploitation

Fonctionne sur le même principe que les deux précédentes, mais sont générées par une instruction spécifique : INT

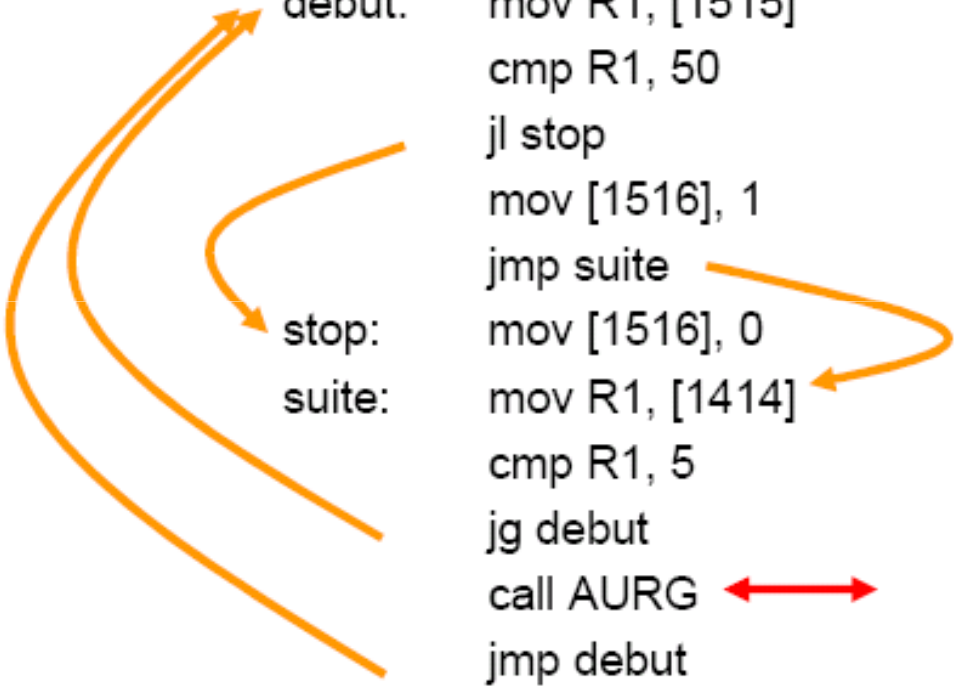
Interruptions matérielles

Ex. Programmation de régulation de la température



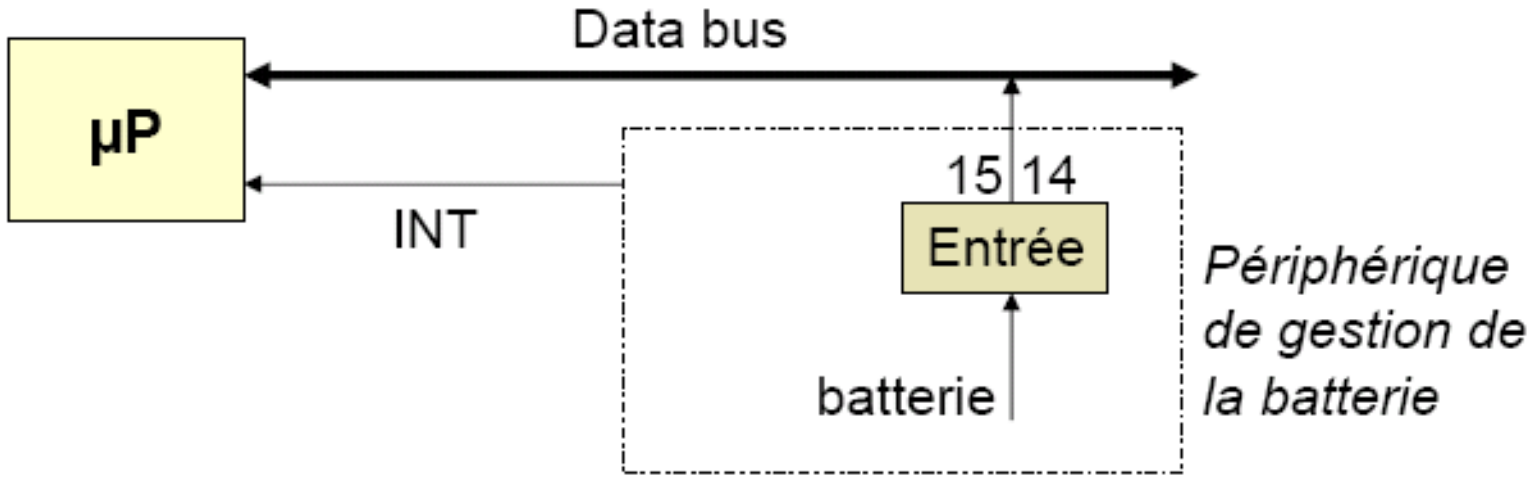

```
debut:  mov R1, [1515]      ; prise de la température
        cmp R1, 50          ; suffisamment basse ?
        jl stop
        mov [1516], 1       ; on met en route le ventilateur
        jmp debut
stop:   mov [1516], 0       ; on arrête le ventilateur
        jmp debut
```

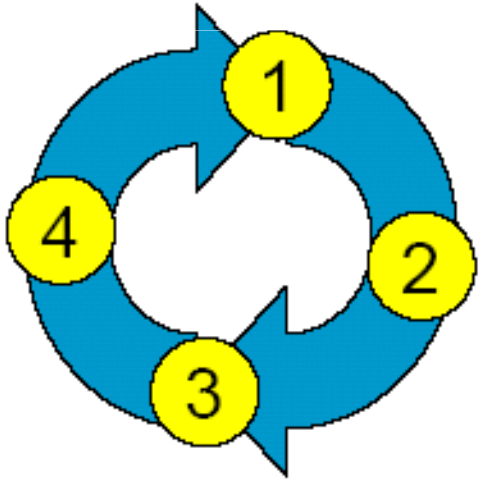
- On introduit un pb supplémentaire :
  - Prise en compte du niveau de la batterie
  - Première méthode → scrutation active (comme pour la t°)



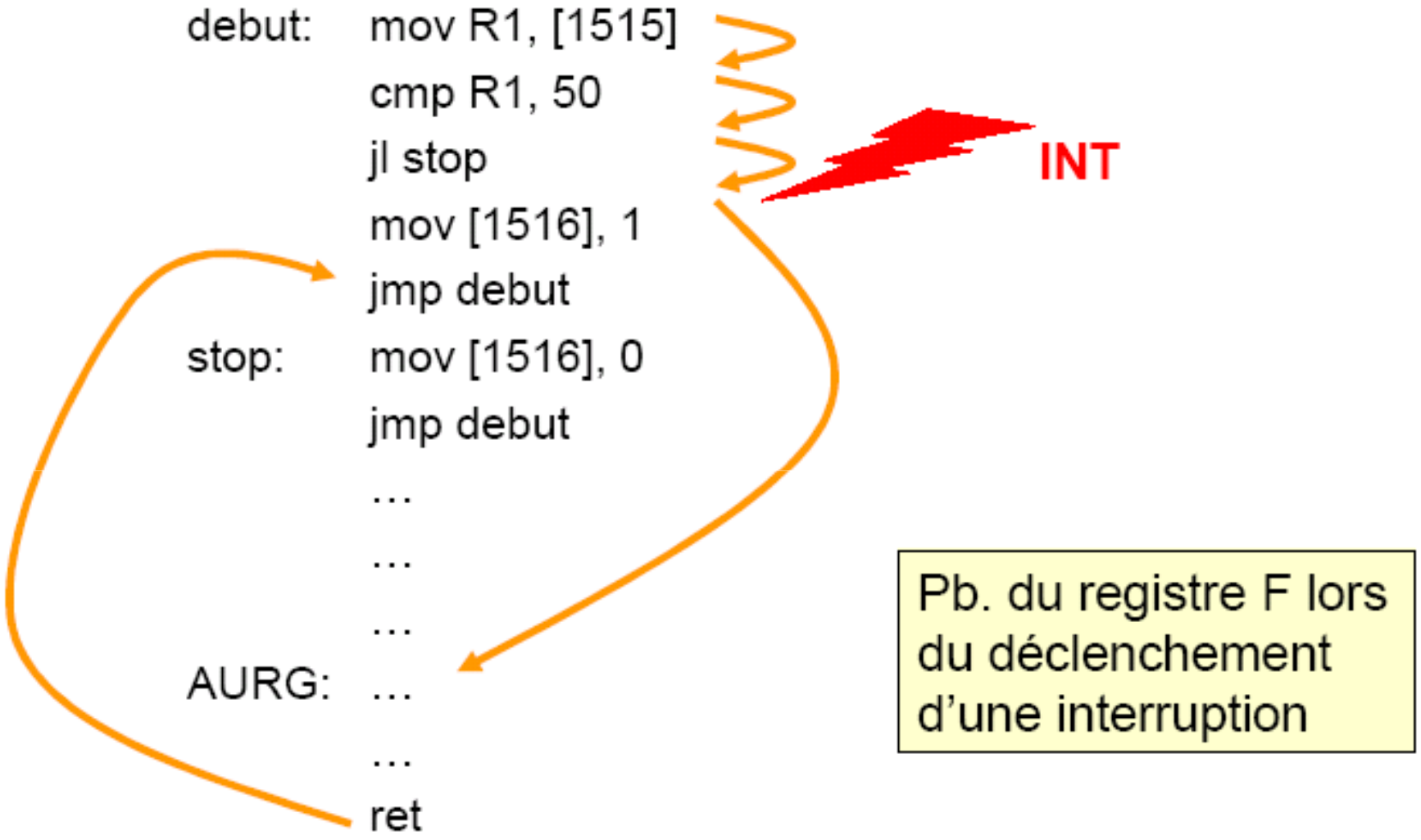
```
debut:   mov R1, [1515]        ; prise de la température
         cmp R1, 50            ; suffisamment basse ?
         jl stop
         mov [1516], 1         ; on met en route le ventilateur
         jmp suite
stop:    mov [1516], 0         ; on arrête le ventilateur
suite:   mov R1, [1414]        ; niveau de la batterie
         cmp R1, 5             ; moins de 5%
         jg debut
         call AURG             ; arrêt d'urgence
         jmp debut
```

- Deuxième méthode :



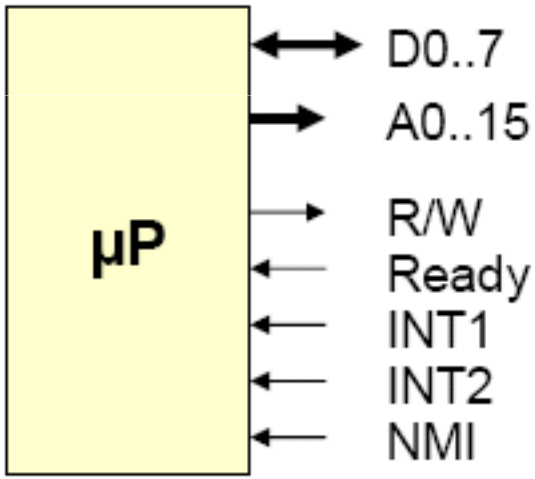


- Si à la phase 4 il y a INT, il ne faut pas exécuter l'instruction suivante, mais un programme particulier
  - Ici le programme particulier, c'est la sous-routine AURG



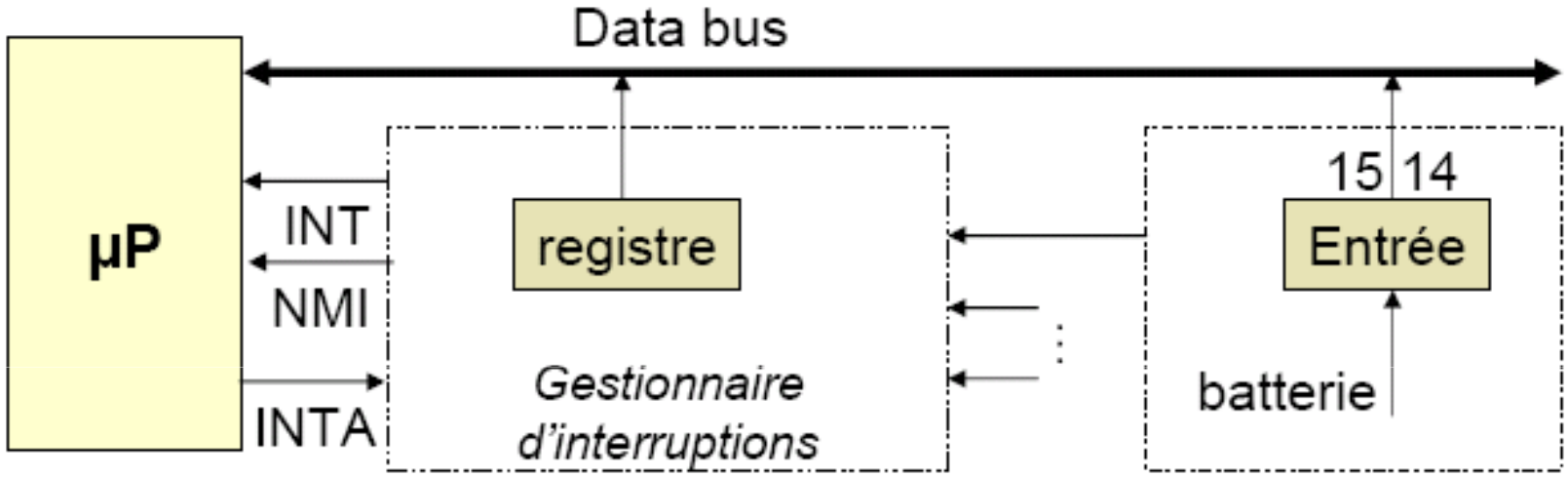
Pb. du registre F lors du déclenchement d'une interruption

- Suivant le µP, il peut y avoir plusieurs lignes d'interruptions
  - INT1, INT2, … (généralement 2 ou 3)
- Il y existe des interruptions non-masquable
  - NMI = Non-Maskable Interruption



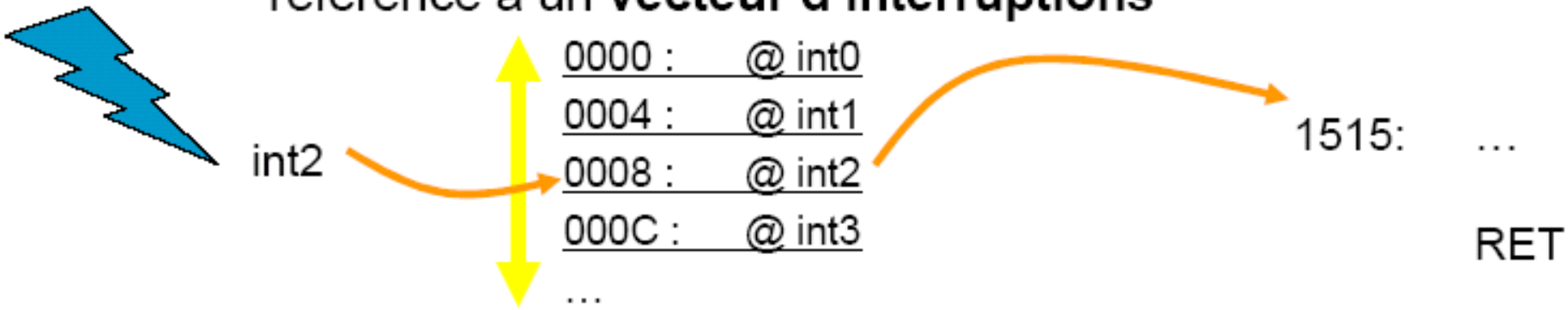
- Les autres interruptions sont donc masquables
  - Instructions DI (disable int.) et EI (enable int.)
  - Utile pour les sections critiques et initialisations

- Comment gérer tous les périphériques avec seulement 2 ou 3 signaux d'interruptions ?



- Le registre contient les informations sur l'interruption à traiter
  - La première instruction du programme de traitement des interruptions va donc être de lire ce registre sur le bus de données.
  - Pour éviter les conflits sur le bus de données, cette information n'est mise sur le bus que sur envoi du signal INTA

- L'information donnée par le gestionnaire d'interruption fait référence à un **vecteur d'interruptions**



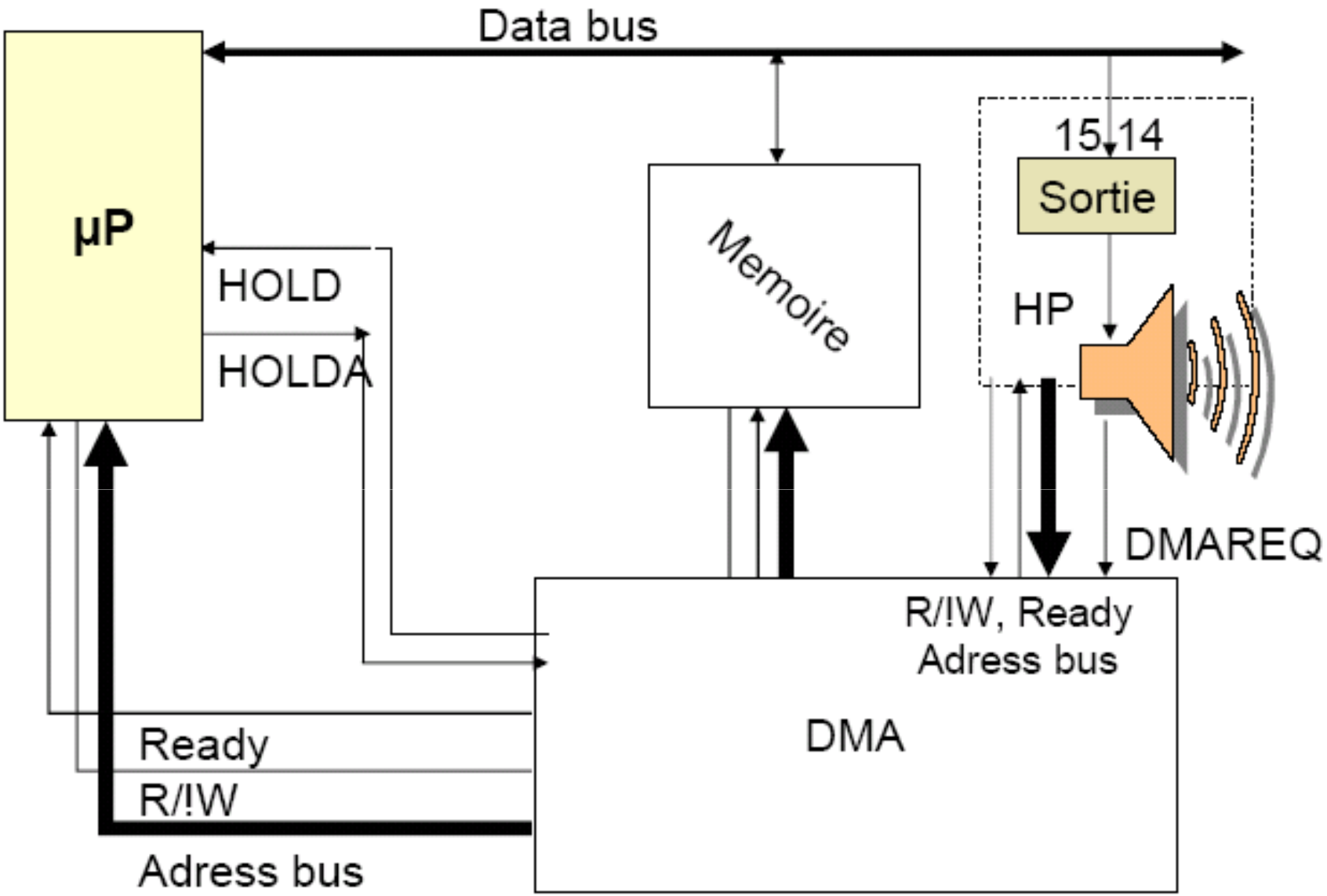
- Au choix :
  - Implémentation soft :
    - Un seul branchement d'INT lors de la phase 4 qui lui va consulter le vecteur d'interruption
  - Implémentation hard :
    - Lors de la phase 4, on consulte le registre d'interruption et on fait le bon branchement…

Interruptions logicielles

- Identique à une interruption matérielle, mais elle est déclenchée par une instruction :
  - INT X
  - où X est le numéro de l'interruption que l'on veut déclencher…
  - On se réfère toujours au vecteur d'interruption (qui peut être changé dynamiquement).
- Ex. avec l'architecture wintel :
  - INT 10h : appels BIOS
    - Basic Input/Output System
  - INT 21h : appels fonctions « système » (ici DOS)

Direct memory access

- Avec les interruptions, on a transformé les attentes actives (*e.g.* par scrutation) en attente passive :
  - Le processeur peut faire une autre tache en attendant le périphérique
- Mais pour le transfert d'informations, c'est le µP qui envoie sert de passerelle entre le périphérique et la mémoire…
- Solution : DMA
  - Accès Direct à la Mémoire
  - Direct Memory Access


Data bus
µP
HOLD
HOLDA
Memoire
15 14
Sortie
HP
DMAREQ
R/!W, Ready
Adress bus
DMA
Ready
R/!W
Adress bus

Chapitre II: Gestion des processus

1. Définitions, exemples
2. Politiques d'allocation du processeur
3. Synchronisation des processus

Les processus, à quoi ça sert ?

Ça sert à faire plusieurs activités en *"**même temps**"*.

Par exemple :

Faire travailler plusieurs utilisateurs sur la même machine. Chaque utilisateur a l'impression d'avoir la machine à lui tout seul.

Par exemple :

Compiler tout en lisant son mail.

Problème : Un processeur ne peut exécuter **qu'une seule instruction** à la fois.

But : Partager un (ou plusieurs) processeur entre différents programmes (les processus).

Attention ! ne pas confondre **Processus** avec **Processeur**

1 Définitions, exemples

Programme/Processus

- Un programme est une suite d'instructions (un objet statique).
- Un processus est un programme en exécution et son contexte
  - Dans un environnement mono-tâche la notion de processus est réduite à sa plus simple expression.
  - Dans un système multitâches plusieurs processus s'exécutent "simultanément". Ils doivent se partager l'accès au processeur.
  - Plusieurs processus peuvent exécuter simultanément des copies (ou instances) d'un même programme.
  - Plusieurs processus peuvent exécuter simultanément la même copie d'un même programme : on parle alors de réentrance.

Plusieurs processus peuvent se trouver simultanément en cours d'exécution (multiprogrammation et temps partagé), si un système informatique ne comporte qu'un seul processeur, alors, à un instant donné, un seul processus aura accès à ce processeur.

En conséquence, un programme en exécution peut avoir plusieurs ***états***.

Note :

La multiprogrammation a été introduite en partie pour éviter qu'un programme en attente de fin d'entrées-sorties ne monopolise le processeur.

États d'un processus

Dans un système multitâches, plusieurs processus peuvent se trouver simultanément en cours d'exécution : ils se partagent l'accès au processeur.

Un processus peut prendre **3 états :**

- Etat **actif** ou élu (running): le processus utilise le processeur.
- Etat **prêt** ou éligible (ready): le processus pourrait utiliser le processeur si il était libre (et si c'était son tour).
- Etat **en attente** ou bloqué : le processus attend une ressource (exp: fin d'une entrée-sortie).

Informaticien fait un gâteau d'anniversaire pour sa fille

- programme => recette de cuisine
- données entrées => ingrédients
- ressources nécessaires => instruments
- processeur => informaticien
- processus => activité de transformation des ingrédients en gâteau

Fils de l'informaticien se fait piquer par une abeille

- interruption du travail => sauvegarde de l'état du processus
- programme => livre de première urgence
- processus => soin à apporter, calmer son fils

Reprise du travail de cuisinier

- restitution de l'état du processus

Fille de l'informaticien annonce de nouveaux invités

- fabriquer 2 gâteaux
- 2 processus distincts sur le même programme
- une seule recette => programme réentrant
- séparation des données propres de chacun des processus

Le S.E choisit un processus qui deviendra actif parmi ceux qui sont prêts.

Tout processus qui se bloque en attente d'un événement (par exemple l'attente de la frappe d'un caractère au clavier lors d'un scanf) passe dans l'état bloqué tant que l'événement attendu n'est pas arrivé.

Lors de l'occurrence de cet événement, le processus passe dans l'état prêt.

Il sera alors susceptible de se voir attribuer le processeur pour continuer ses activités.

Les processus en attente sont marqués comme bloqués dans la table des processus. Ils se trouvent généralement dans la file d'attente liée à une ressource (imprimante, disque, ...).

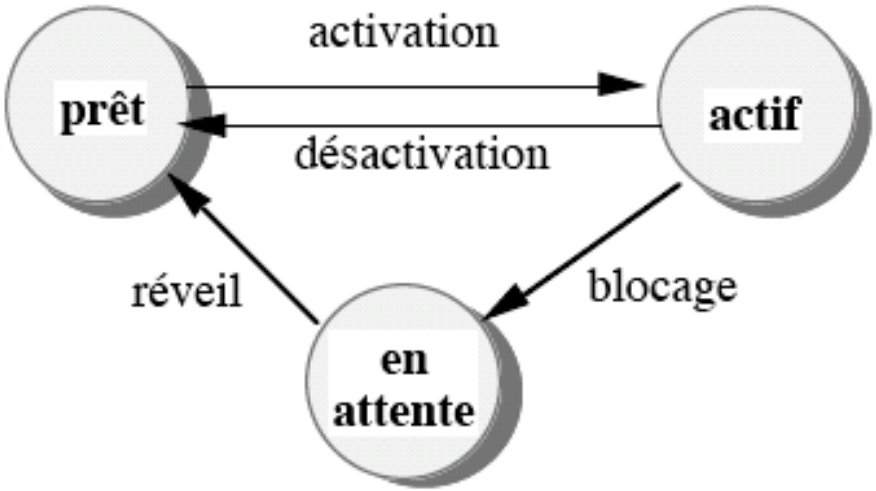
Le changement d'état d'un processus peut être provoqué par :

- un autre processus (qui lui a envoyé un signal, par exemple)
- le processus lui-même (appel à une fonction d'entrée-sortie bloquante,...)
- une **interruption** (fin de quantum, terminaison d'entrée-sortie, ...)

Transitions entre états

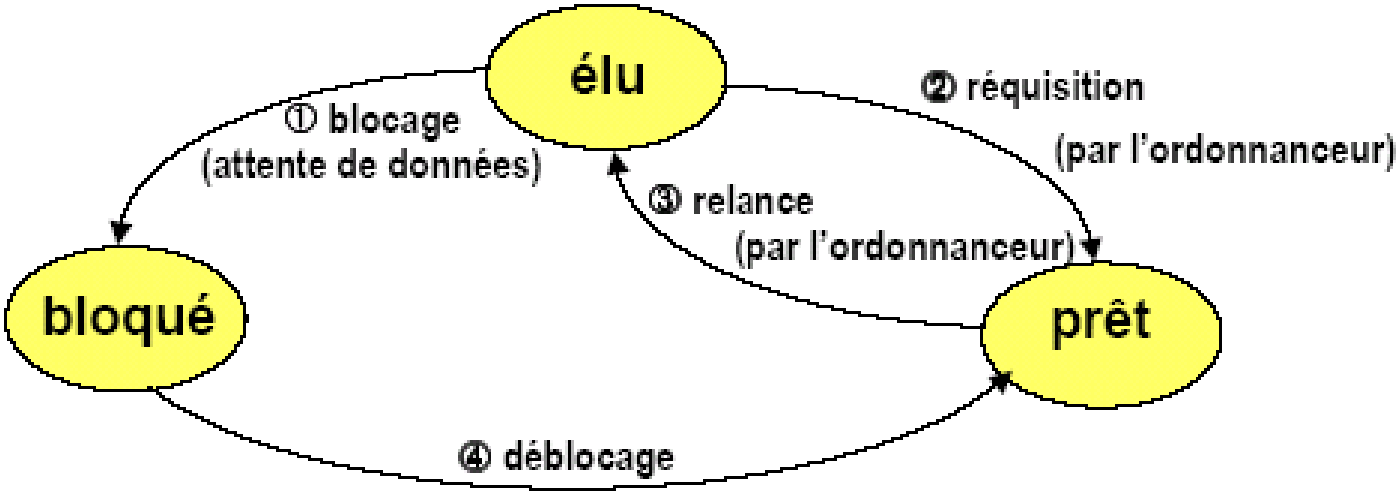
• Les transitions entre états provoquent le passage d'un état à un autre :

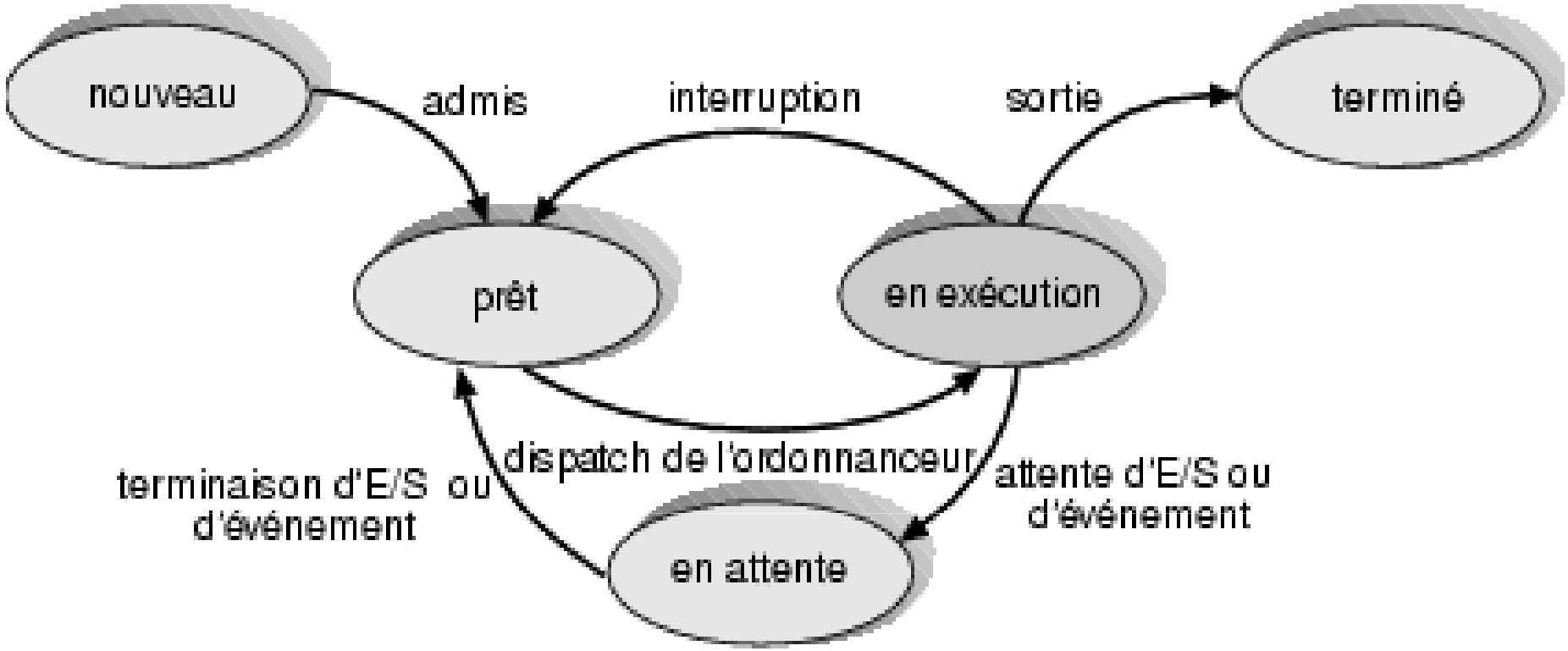
Activation	prêt -> actif.
Désactivation (ou réquisition)	actif -> prêt.
Mise en attente (ou blocage)	actif -> en attente.
Réveil	attente -> prêt.



Les différents états d'un processus

- Un processus peut être dans 3 états possibles :
  - élu (en cours d'exécution)
    processus OK, processeur OK
  - prêt (suspendu provisoirement pour qu'un autre processus s'exécute)
    processus OK, processeur occupé
  - bloqué (attendant un événement extérieur pour continuer)
    processus non OK, même si processeur OK





Les opérations que l'on peut effectuer sur les processus sont :

- Création (ex : fork, création d'un processus sous UNIX),
- Destruction (ex : kill, destruction d'un processus sous UNIX),
- Activation, Désactivation, Blocage(ex : wait sous UNIX),

Le système d'exploitation gère les transitions entre les états.

Pour ce faire il maintient une liste de processus éligibles et une liste de processus en attente.

Il doit également avoir une politique d'activation des processus éligibles (ou politique d'allocation du processeur aux processus) : parmi les processus éligibles, faut-il activer celui qui est éligible depuis le plus de temps, celui qui aurait la plus forte priorité ou celui qui demande l'unité centrale pour le temps le plus court (à supposer que cette information soit disponible)?

Les processus sont classés dans la file des processus éligibles par **l'*ordonnanceur*** (en anglais *scheduler*). C'est le ***distributeur*** (en anglais *dispatcher*) qui se chargera de leur activation au moment voulu.

Afin d'éviter qu'un processus occupe l'unité centrale, le système d'exploitation déclenche un temporisateur à chaque fois qu'il alloue l'unité centrale à un processus.

Quand ce temporisateur expire, si le Processus occupe toujours l'unité centrale (il peut avoir été interrompu par un événement extérieur ou s'être mis en attente), le système d'exploitation va le faire passer dans l'état prêt et activer un autre processus de la liste des processus éligibles. La désactivation d'un processus peut se faire sur expiration du temporisateur ou sur réquisition (en anglais preemption) du processeur central par un autre processus.



Un temporisateur peut être vu comme un système de compte à rebours ou de sablier. On lui donne une valeur initiale qu'il décrémente au rythme de l'horloge de l'ordinateur jusqu'à atteindre la valeur zéro.

Ici, la valeur initiale du temporisateur est le ***quantum*** de temps alloué à l'exécution du processus.

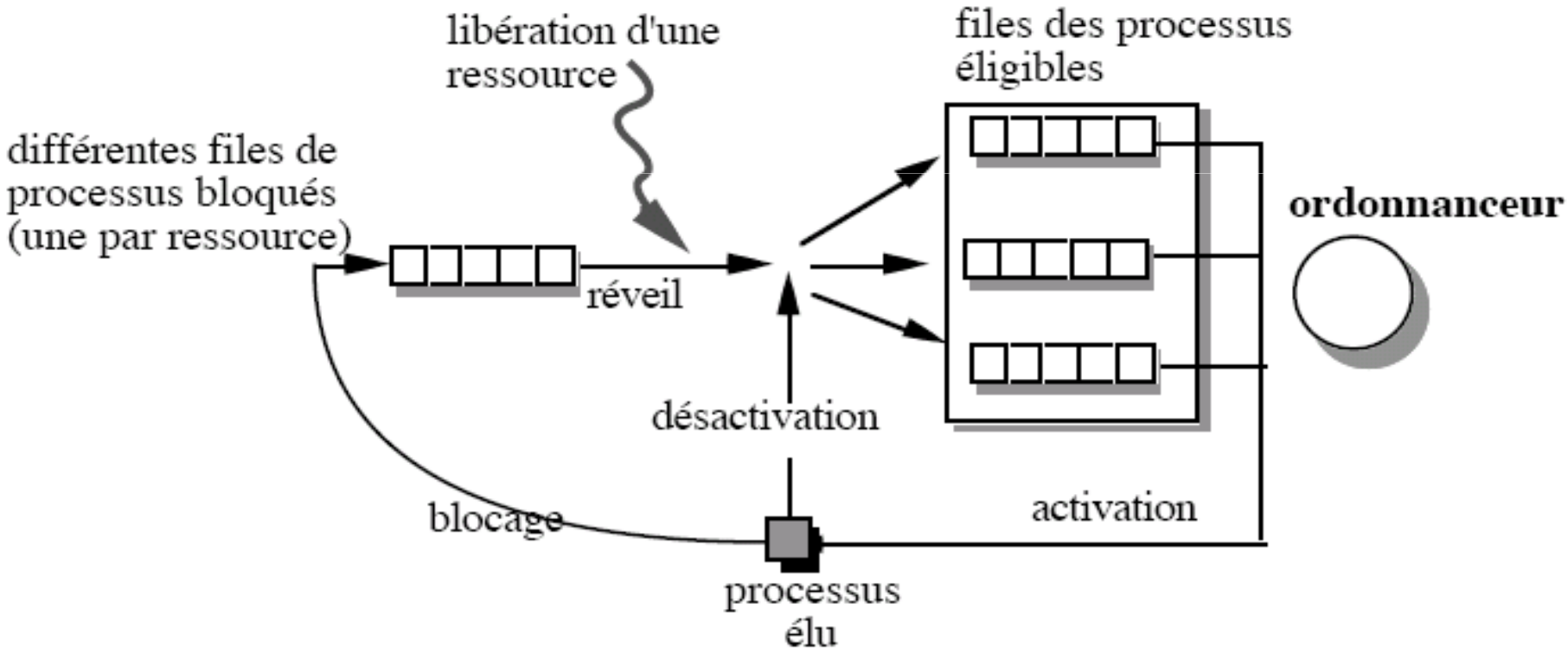
A l'expiration du temporisateur le système d'exploitation est Immédiatement prévenu par une *interruption*. Nous verrons cette notion plus en détail dans la suite de ce chapitre.

D'autres événements sont signalés au système par des interruptions :

c'est le cas des fins d'entrées-sorties. Le système est ainsi prévenu que le ou les processus qui étaient en attente sur cette entrée-sortie peuvent éventuellement changer d'état (être réveillés).

Ordonnancement

• Dans un système multitâches, le système d'exploitation doit gérer l'allocation du processeur aux processus. On parle d'ordonnancement des processus.



L`Ordonnanceur est la partie du SE qui décide quel processus dans la file ready/prêt obtient le processeur quand il devient libre.

L'Ordonnanceur doit prendre sa décision quand le contenu de la liste Prêt-Ready change, c-à-d.

- un processus se termine ou non
- un processus exécutant devient bloqué en attente
- un processus change d'exécutant/running à prêt/ready
- un processus change de attente à prêt/ready

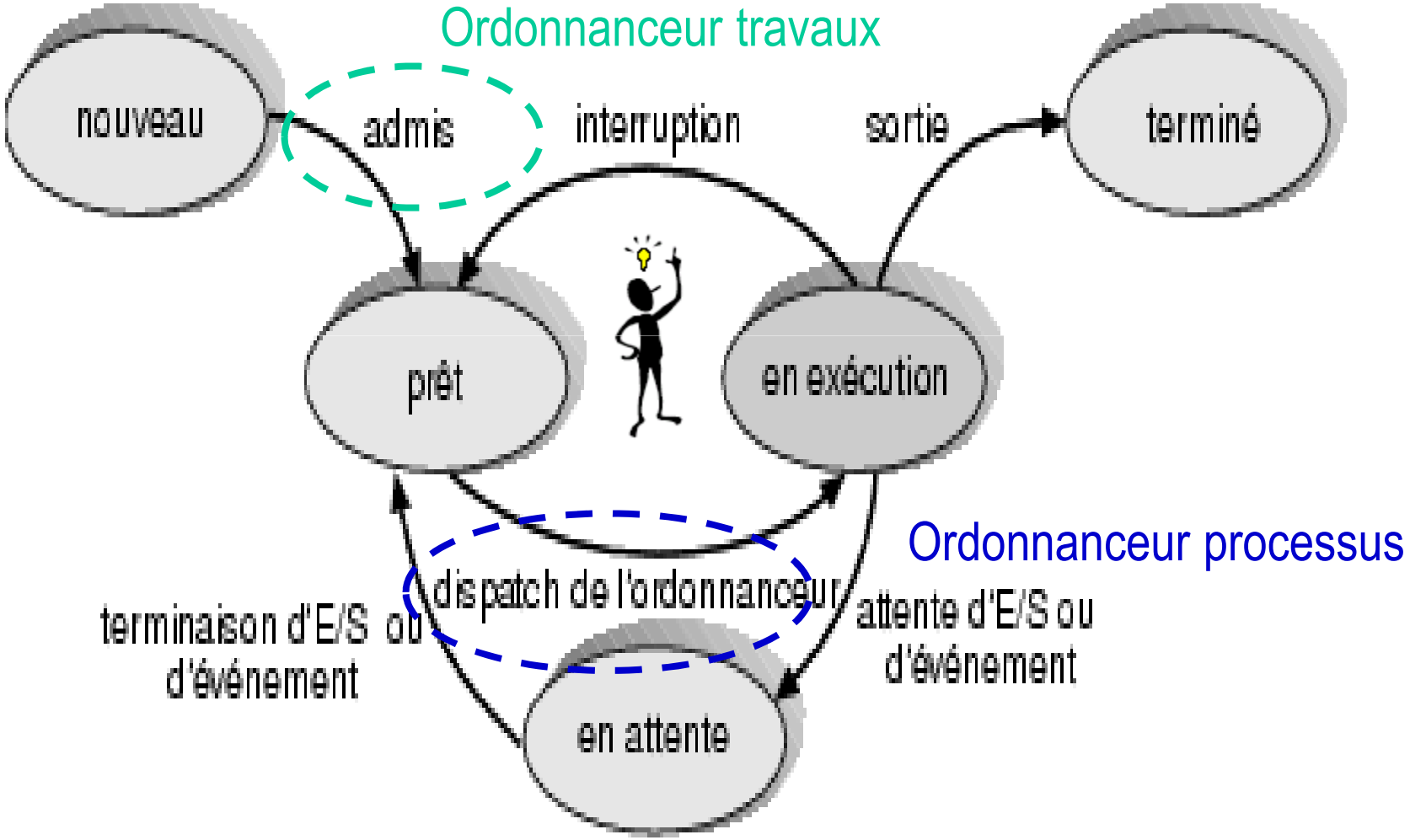


Ordonnanceurs (schedulers)

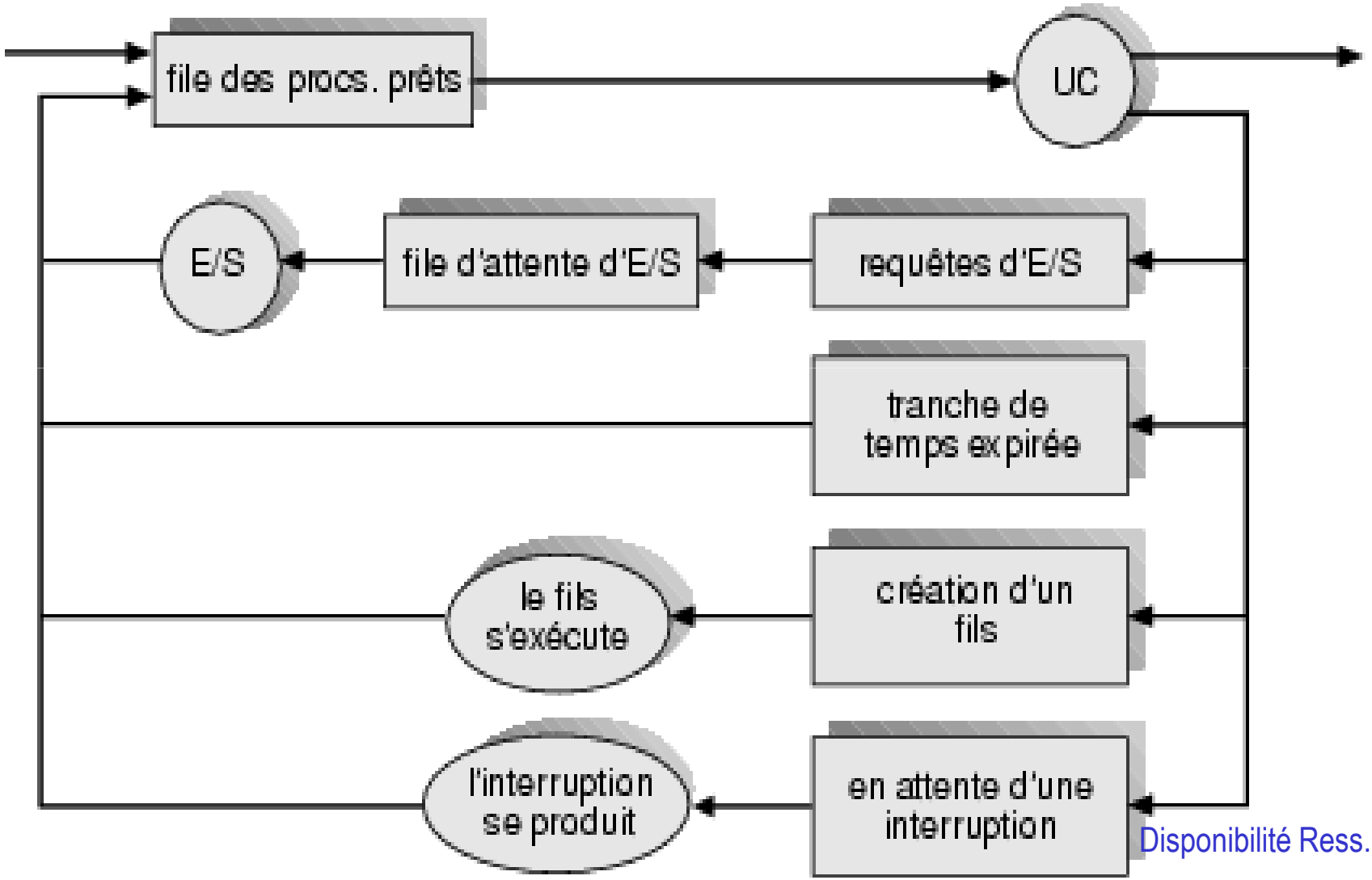
- Programmes qui gèrent l ’utilisation de ressources de l`ordinateur
- Trois types d`Ordonnanceurs :
  - À court terme = Ordonnanceur Processus: sélectionne quel processus doit exécuter la transition

    prêt $\rightarrow$ exécution
  - À long terme = Ordonnanceur Travaux: sélectionne quels processus peuvent exécuter la transition nouveau $\rightarrow$ prêt (événement *admitted*) (de spoule travaux à file prêt)
  - À moyen terme: nous verrons

Ordonnanceur travaux = long terme
et Ordonnanceur processus = court terme



Ordonnancement de processus (court terme)



Ordonnanceur à moyen terme

- Le manque de ressources peut parfois forcer le SE à *suspendre* des processus
  - ils seront plus en concurrence avec les autres pour des ressources
  - ils seront repris plus tard quand les ressources deviendront disponibles
- Ces processus sont enlevés de mémoire centrale et mis en mémoire secondaire, pour être repris plus tard
  - `swap out`, `swap in` , va-et-vient

Dispatcheur

Celui qui donne le contrôle au processus choisi par l'ordonnanceur.

Il doit se préoccuper de:

- changer de contexte
- changer à mode usager
- réamorcer le processus choisi



Contexte d'un processus

Par contexte, on entend toutes les informations relatives à l'exécution d'un processus, telles que :

- l'état des registres (compteur ordinal, registre d'état, registres généraux),
- les registres décrivant l'espace virtuel du processus et son espace physique d'implantation,
- les pointeurs de piles,
- les ressources accédées (ex : droits d'accès, fichiers ouverts),
-autres informations (ex : valeur d'horloge).

Le contexte doit être sauvegardé quand le processus est désactivé ou bloqué et doit être restauré lorsqu'il est réactivé.

Il existe deux *contextes* : le contexte *matériel* et le contexte *logiciel*.

Le contexte matériel est la photographie de l'état des registres à un instant *t*: compteur ordinal, pointeur de pile, registre d'état qui indique si le processeur fonctionne en mode utilisateur ou en mode noyau (*kernel*).

Le contexte logiciel contient des informations sur les conditions et droits d'accès aux ressources de la machine!: priorité de base, quotas sur les différentes ressources (nombre de fichiers ouverts, espaces disponibles en mémoires virtuelle et physique...).

Le *bloc de contrôle d'un processus* (en anglais *Process Control Block* ou *PCB*) est une structure de données qui contient toutes les information relatives au contexte d'un processus. Quand un processus est désactivé Ou bloqué, toutes les information relatives à son contexte sont Sauvegardées dans son bloc de contrôle. Quand un processus est activé, on restaure son contexte à partir de son bloc de contrôle.

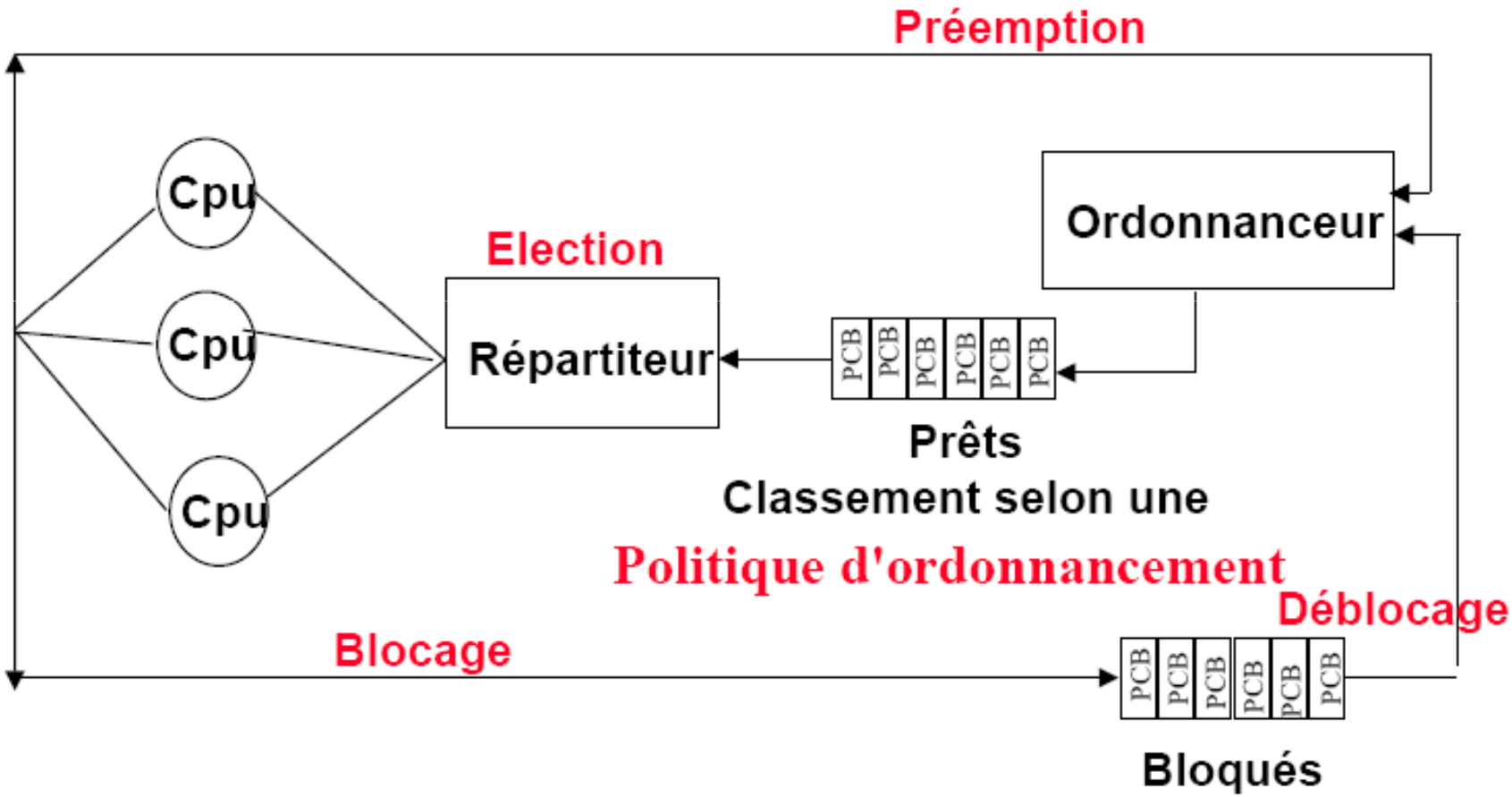


Chaque fois que le processeur passe à l'exécution d'un nouveau processus il doit changer les configurations de ses registres. On parle de *changement de contexte*. Certaines informations (une partie du contexte logiciel) concernant les processus doivent rester en permanence en Mémoire dans l'espace système. D'autres peuvent être transférées sur disque avec le processus quand celui ci doit passer de la mémoire au disque (quand il n'y a plus de place en mémoire pour y laisser tous les processus, par exemple). Il s'agit en général de tout le contexte matériel et d'une partie du contexte logiciel.

La plupart des processeurs proposent des instructions spéciales de changement de contexte (cf. machines Motorola, Intel, ...).

Système multiprocessus

Ordonnanceur et répartiteur



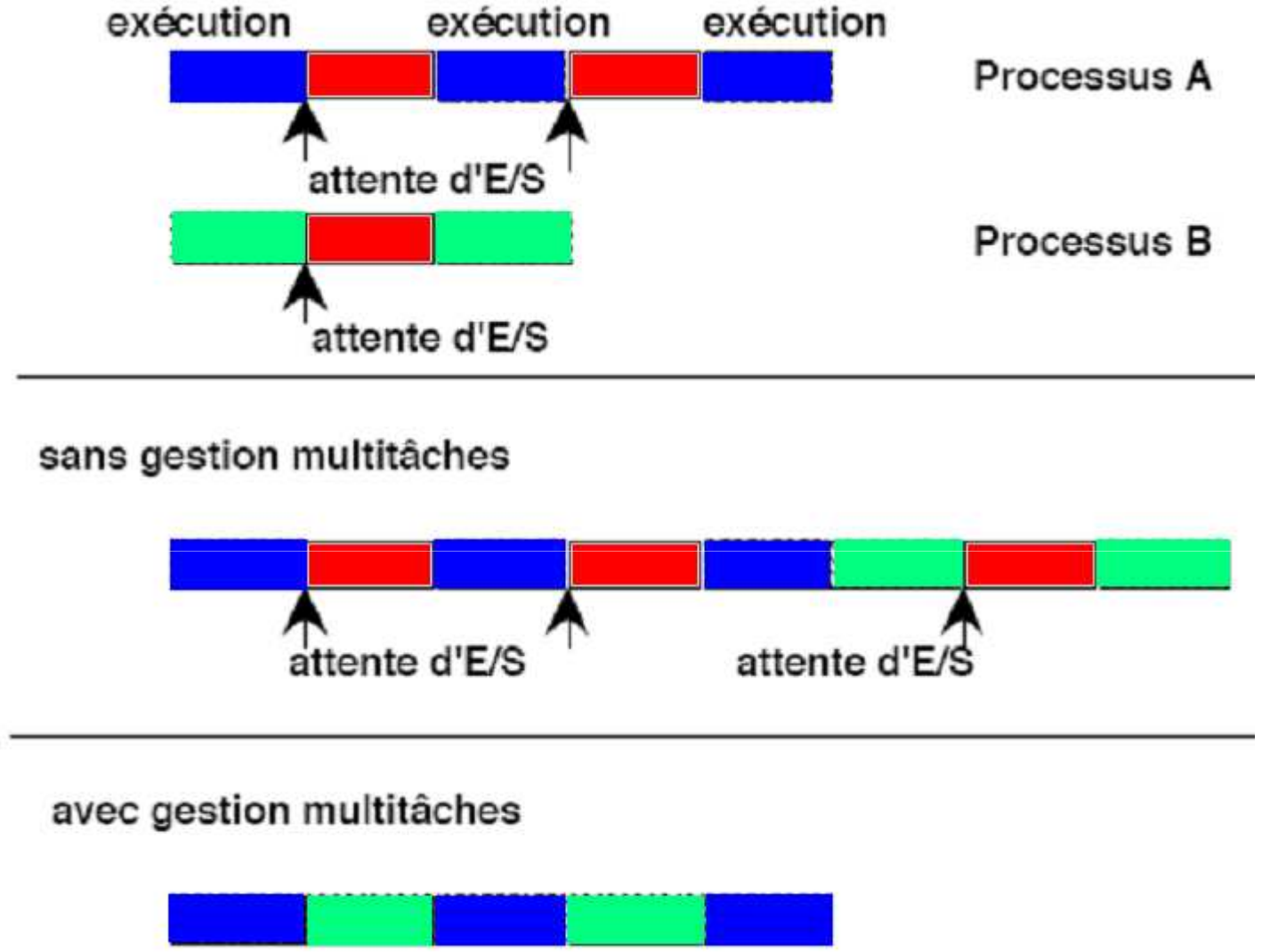
2. Politiques d'allocation du processeur

Un système propose la multiprogrammation s'il permet de charger simultanément plusieurs processus en mémoire et de leur faire partager l'unité centrale. Le principe de fonctionnement est de laisser un processus s'exécuter jusqu'à ce qu'il ait besoin de se mettre en attente (par exemple sur une fin d'entrée/sortie). Pendant que ce processus est en attente, un autre processus peut utiliser à son tour le processeur.

Pour illustrer ce fonctionnement, supposons que l'on ait deux processus **A** et **B** à exécuter, chacun alternant des périodes **d'exécution** dans l'unité centrale et des périodes **d'attente**. L'exécution totale du processus **A** nécessite **5** unités de temps, **dont 2 d'attente**, celle du processus **B** nécessite **3** unités de temps, dont **1 d'attente** (cf. schéma ci-contre).

L'exécution de ces deux processus sans multiprogrammation prend 8 unités de temps. Avec multiprogrammation, elle ne prends plus que 5 unités de temps (soit la durée d'exécution du processus A)(cf. schéma ci-contre).

Intérêt du multitâches



• **Problème :** quelle politique d'allocation choisir pour utiliser le processeur "au mieux" ?

Dans la suite de ce chapitre, nous allons montrer le fonctionnement des systèmes qui supportent la multiprogrammation. Il s'agira de décider comment l'unité centrale est partagée entre les processus chargés en mémoire. C'est ce que l'on appelle la ***politique*** (ou *algorithme*) ***d'allocation du processeur*** ou encore ***politique d'ordonnancement***. On peut décider que c'est le premier processus qui est prêt à s'exécuter qui s'exécute (politique ***Premier Arrivé Premier Servi***). On peut aussi choisir le processus dont la durée d'exécution est la plus courte pour lui attribuer l'unité centrale d'abord (politique ***Plus Court d'Abord***). Nous examinerons différentes politiques d'allocation dans la suite de ce chapitre.

Plusieurs politiques d'allocations étant possibles, la question se pose de savoir comment les évaluer. Il faudra tout d'abord s'assurer que la politique ne créée pas de *famine* (*il y a famine si un processus prêt à s'exécuter n'accède pas à l'unité centrale en un temps fini*). On pourra ensuite utiliser différents critères d'évaluation tels que le *rendement*, le *temps de service*, le *temps d'attente* et le *temps de réponse*, qui sont définis dans le paragraphe suivant.

Le ***rendement*** d'un système (en anglais *throughput*) est le nombre de processus exécutés par unité de temps.

Le ***temps de service*** *(turnaround time)* est le temps qui s'écoule entre le moment où un processus devient prêt à s'exécuter et le moment où il finit de s'exécuter (temps d'accès mémoire + temps d'attente dans la file des processus éligibles + temps d'exécution dans l'unité centrale + temps d'attente + temps d'exécution dans les périphériques d'entrée/sortie). On utilise en général un *temps moyen de service* calculé pour tous les processus mis en jeu pendant une période d'observation donnée.

Le ***temps d'attente*** (en anglais *waiting time*) est le temps passé dans la file des processus éligibles. On utilise en général un *temps moyen d'attente* calculé pour tous les processus mis en jeu pendant une période d'observation donnée.

Le ***temps de réponse*** (en anglais *response time*) est le temps qui s'écoule entre la soumission d'une requête et la première réponse obtenue. On utilise en général un *temps moyen de réponse* calculé pour tous les processus mis en jeu pendant une période d'observation donnée.

• Rendement (throughput) :
Nombre de travaux exécutés par unité de temps.

• Temps de service (turnaround time):
Temps qui s'écoule entre le moment où un travail est soumis et où il est exécuté (temps d'accès mémoire + temps d'attente en file des éligibles + temps d'exécution dans le processeur et E/S).

• Temps d'attente (waiting time) :
Temps passé dans la file des processus éligibles.

• Temps de réponse (response time):
Temps qui s'écoule entre la soumission d'une requête et la première réponse obtenue.

First Come First Served (FCFS)

Premier Arrivé Premier Servi (PAPS)

• L'ordre d'exécution est identique à l'ordre d'arrivée dans la file des processus éligibles.

• Exemple : On a trois processus 1, 2, 3 à exécuter de durées d'exécution respectives 12 , 2 et 2 unités de temps.

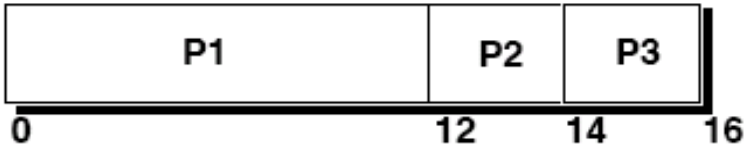


Diagramme de Gant, algorithme PAPS

- Si l'ordre d'arrivée est 1, 2, 3, les temps de service respectifs sont 12, 14, et 16 unités de temps, et le temps de service moyen est de 14 unités de temps.

- Mais si l'ordre d'arrivée est 2,3,1, le temps de service moyen est 7,6 (2+4+16 / 3).

Le Plus Court d'Abord (PCA)

Shortest Job First (SJF)

• On exécute d'abord les processus qui ont le temps d'exécution le plus court.

• Problème : comment prédire la durée d'exécution des processus a priori ?

• Solutions :
- faire de la prédiction sur la durée d'exécution des processus.
- faire accompagner la demande d'exécution de chaque programme d'une durée maximum d'exécution autorisée qui est utilisée comme durée d'exécution.

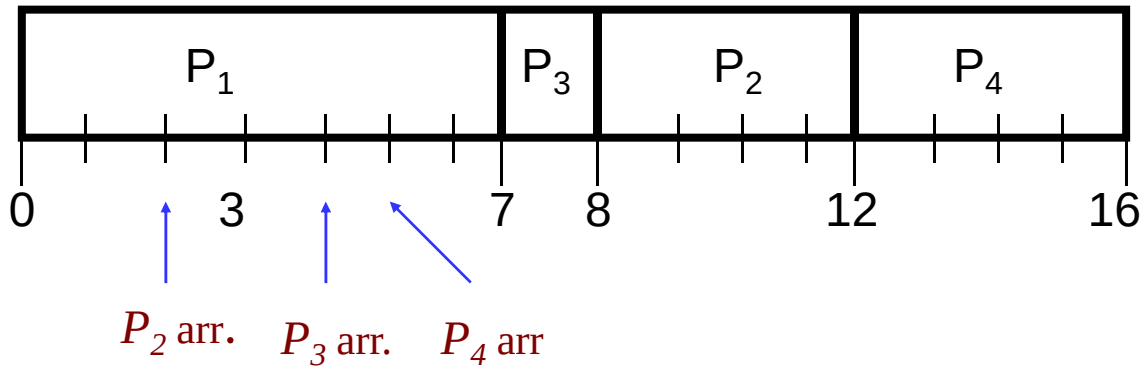
• Autre problème : risque de famine (les processus de longue durée peuvent n'avoir jamais accès au processeur si des processus de courte durée arrivent en permanence).

- Avec préemption: si un processus qui dure moins que le *restant* du processus courant se présente plus tard, le proc courant est interrompu (SRTF: shortest remaining-time first)
- Sans préemption: décision définitive

Example de SJF sans préemption

Processus	Arrivée	Cycle
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (sans préemption)

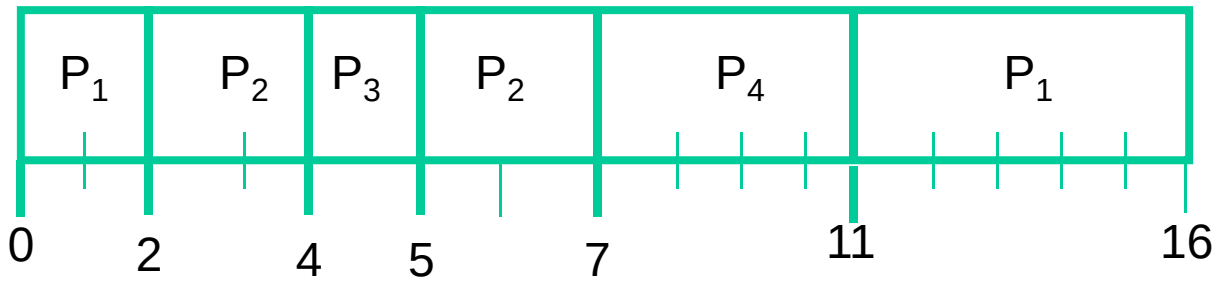


- Temps d'attente moyen = (0 + 6 + 3 + 7)/4 = 4

Exemple de SJF avec préemption

Processus	Arrivée	Cycle
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (préemptive)



P_2 arr. P_3 arr. P_4 arr

- Temps moyen d`attente = (9 + 1 + 0 +2)/4 = 3
  - **P1 attend de 2 à 11, P2 de 4 à 5, P4 de 5 à 7**

Priorité

- On associe une priorité à chaque processus. Les processus sont exécutés par ordre de priorité.
- Problème : risque de famine.
- Solution : utiliser une technique dite de *vieillissement* qui augmente périodiquement la priorité des processus.
- Intérêt : certains processus système doivent être exécutés en priorité.

Le principe de l'algorithme de priorité est d'associer une priorité à chaque processus. On exécute alors les processus de forte priorité avant ceux de plus basse priorité. On peut remarquer que ***si l'on alloue à chaque processus une priorité inversement proportionnelle à sa durée, on retrouve l'algorithme PCA***.

Le problème de l'algorithme de priorité est qu'il y a risque de famine (les processus de faible priorité pouvant n'avoir jamais accès à l'unité centrale si des processus de fortes priorité arrivent en permanence). Pour y remédier,

on peut utiliser une technique dite de ***vieillissement*** (en anglais *aging*). Le principe est d'accroître la priorité des processus à intervalles de temps réguliers, jusqu'à ce qu'ils accèdent à l'unité centrale.

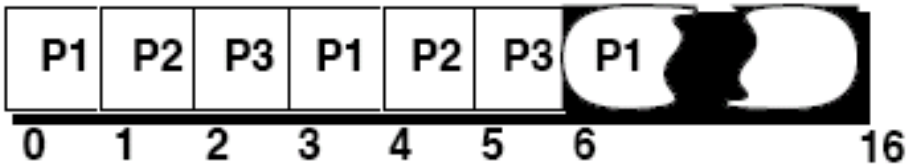
Algorithme du tourniquet (Round Robin)

• Le temps est découpé en tranches ou quantum de temps. Un quantum de temps est alloué à chaque processus de la file des processus éligibles à tour de rôle.

• Conçu pour les systèmes "temps partagé", afin d'obtenir une utilisation équitable du processeur par les processus présents en mémoire.

• Exemple : Si le quantum est d'une unité de temps

On a trois processus 1, 2, 3 à exécuter de durées d'exécution respectives 12 , 2 et 2 unités de temps.



- Temps moyen de service : 9 (5+6+16 / 3).

(Meilleur que celui donné par l'algorithme FCFS pour le même ordre d'arrivé:14)

- Performances sensibles à la taille choisie pour le quantum.

Cet algorithme a été conçu pour les systèmes ***temps partagé***, afin d'obtenir une répartition équitable de l'unité centrale entre les différents processus présents en mémoire. Son principe est le suivant : le temps est découpé en tranches ou quantum de temps. Un quantum de temps est alloué à chaque processus de la file des processus éligibles à tour de rôle.

En pratique, la file des processus éligibles est une FIFO. Les nouveaux processus sont placés à la fin de la FIFO. Le système sélectionne le premier processus de la file et arme un temporisateur initialisé à la durée d'un quantum. Le processus peut quitter l'unité centrale avant échéance du temporisateur s'il est terminé ou en attente d'événement. Sinon, à échéance du temporisateur le système reçoit une interruption et la traite en sauvegardant le contexte du processus et en le plaçant à la fin de la file des processus éligibles.

La qualité de cet algorithme va cependant beaucoup dépendre de la taille choisie pour le quantum : elle doit être grande par rapport au temps nécessaire pour le changement de contexte des processus.

Round Robin (tourniquet)

Cet algorithme est spécialement adapté aux systèmes en temps partagé. On définit un **quantum de temps** (time quantum) d'utilisation de l'unité centrale. La file d'attente des processus éligibles est vue comme une queue circulaire (fifo circulaire). Tout nouveau processus est placé à la fin de la liste.

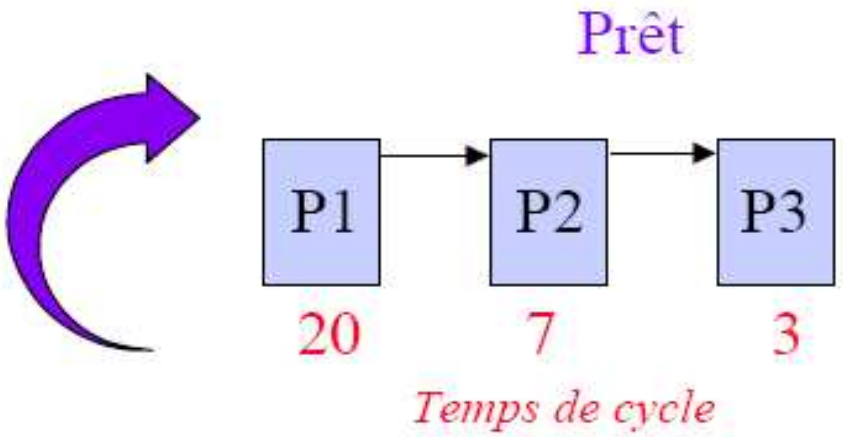
Deux cas: l'une, soit le processus actif rend l'Unité Centrale avant la fin de sa tranche de temps (pour cause d'entrée/sortie) soit il est préempté, et dans les deux cas placé en fin de liste.

Un processus obtiendra le processeur au bout de (n -1)*q secondes au plus (n nombre de processus et q longueur du quantum de temps), la famine est donc assurément évitée.

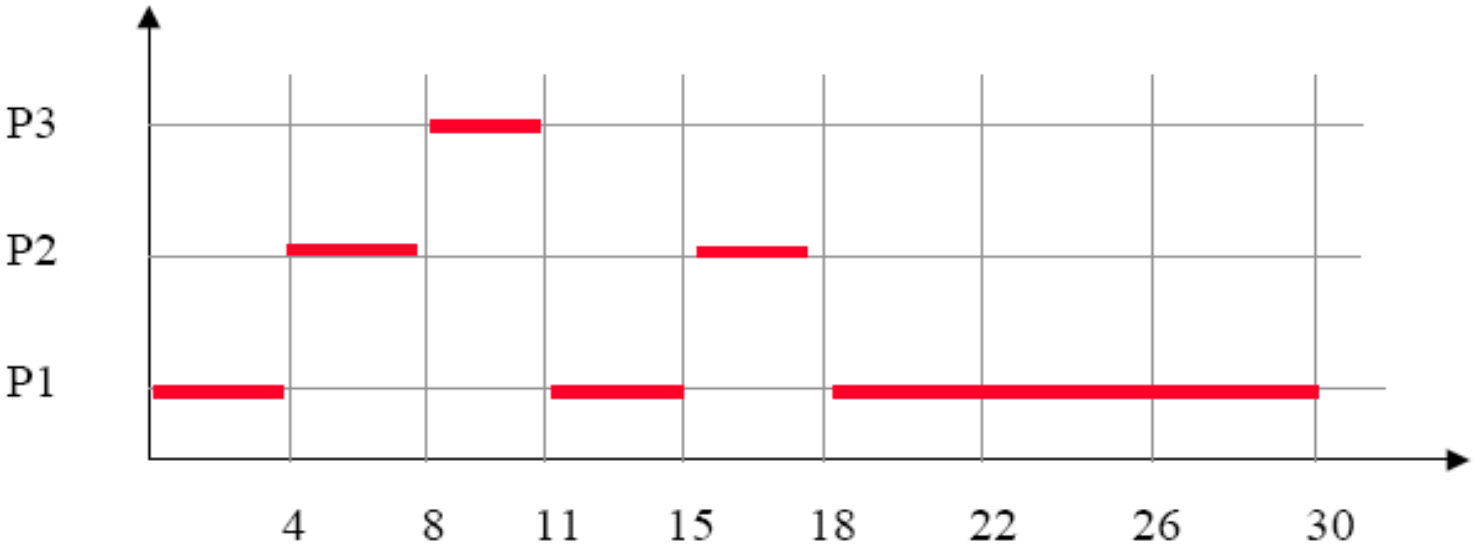
Remarquons que si le quantum de temps est trop grand, round-robin devient équivalent à FCFS. De l'autre coté si le quantum de temps est très court, nous avons théoriquement un processeur n fois moins rapide pour chaque processus (n nombre de processus).

Malheureusement si le quantum de temps est court, le nombre de changements de contexte dus à la préemption grandit, d'où une diminution du taux utile, d'où un processeur virtuel très lent.

Algorithme : tourniquet



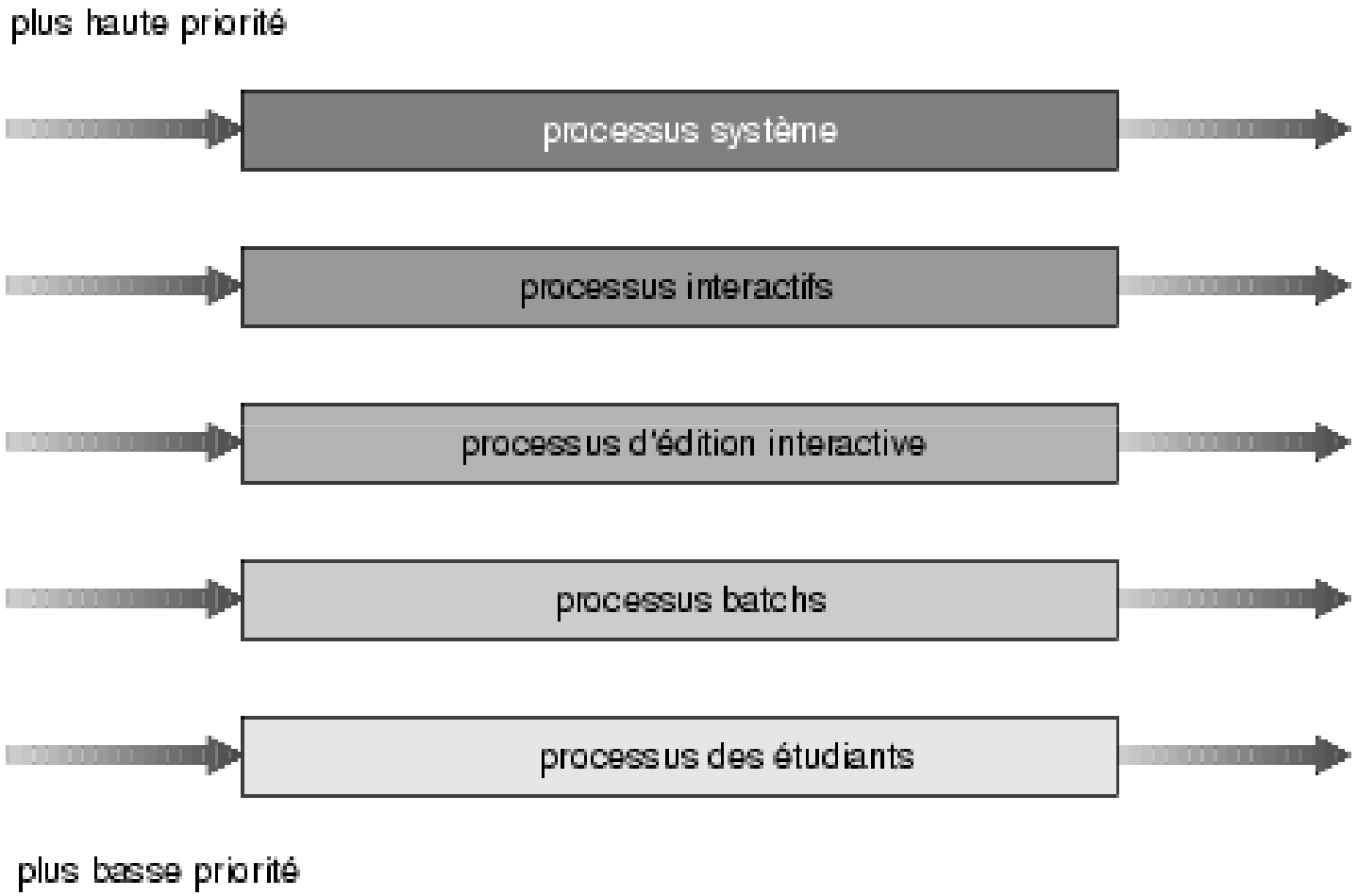
Quantum = 4



Files à plusieurs niveaux (multiples)

- La file *prêt* est séparée en plusieurs files, p.ex.
  - travaux `d'arrière-plan` (background - batch)
  - travaux `de premier plan` (foreground - interactive)
- Chaque file a son propre algorithme d'ordonnancement, p.ex.
  - FCFS pour arrière-plan
  - Tourniquet pour premier plan
- Comment ordonnancer entre files?
  - Priorité fixe à chaque file → famine possible, ou
  - Chaque file reçoit un certain pourcentage de temps UCT, p.ex.
    - 80% pour arrière-plan
    - 20% pour premier plan

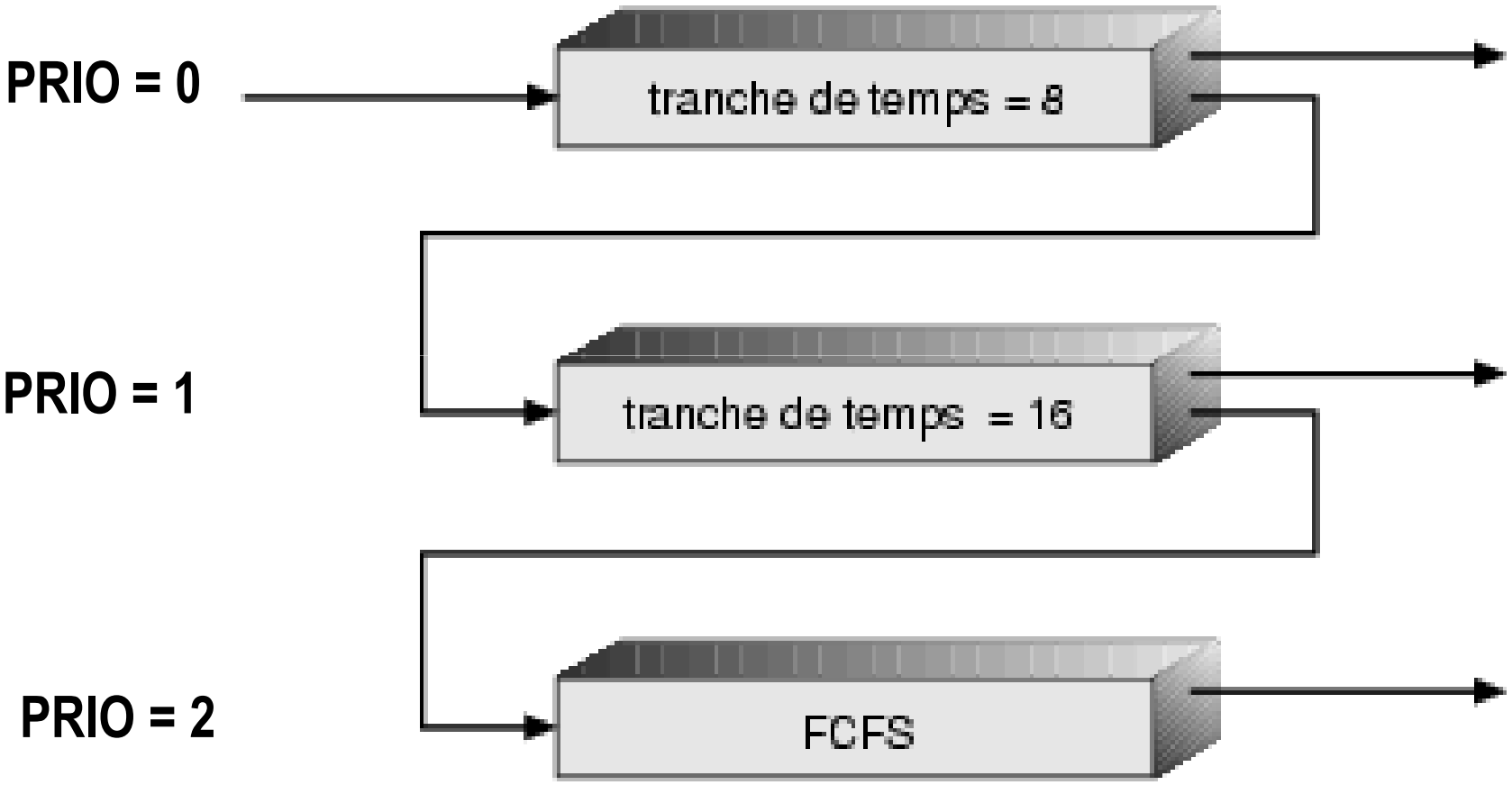
Ordonnancement avec files multiples



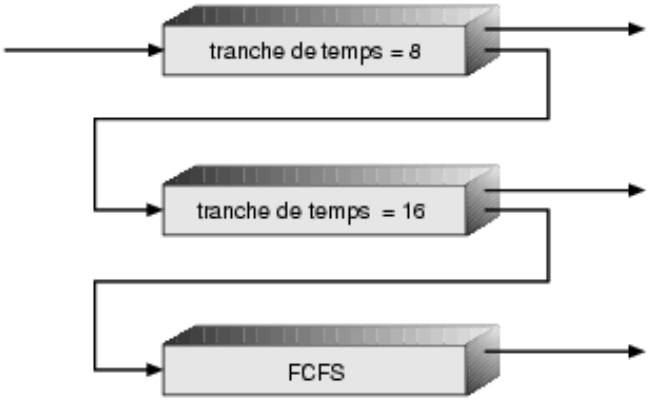
Files multiples et à retour

- Un processus peut passer d ’une file à l ’autre, p.ex. quand il a passé trop de temps dans une file
- À déterminer:
  - nombre de files
  - algorithmes d ’ordonnancement pour chaque file
  - algorithmes pour décider quand un proc doit passer d ’une file à l`autre
  - algorithme pour déterminer, pour un proc qui devient prêt, sur quelle file il doit être mis

Files multiples et à retour



Exemple de files multiples à retour



- Trois files:
  - Q0: tourniquet, quantum 8 msecs
  - Q1: tourniquet, quantum 16 msecs
  - Q2: FCFS
- Ordonnancement:
  - Un nouveau processus entre dans Q0, il reçoit 8 msecs d ’UCT
  - S ’il ne finit pas dans les 8 msecs, il est mis dans Q1, il reçoit 16 msecs additionnels
  - S ’il ne finit pas encore, il est interrompu et mis dans Q2
  - Si plus tard il commence à demander des quantums plus petits, il pourrait retourner à Q0 ou Q1

3. Synchronisation des processus

Introduction

Les processus concurrents s'exécutant dans le système d'exploitation peuvent être des processus coopératifs ou indépendants.

♦Un processus est *indépendant* s'il n'affecte pas les autres processus ou ne peut pas être affecté par eux.

♦Un processus qui ne partage pas de données avec d'autres processus est indépendant

♦Un processus est *coopératif* s'il peut affecter les autres processus en cours d'exécution ou être affecté par eux

♦ Un processus qui partage des données avec d'autres processus est un processus coopératif

♦ Les données partagées par les processus coopératifs peuvent se trouver en mémoire principale ou en mémoire secondaire dans un fichier

♦ Les accès concurrents à des données partagées peuvent produire des incohérences de données comme le montre l'exemple suivant :

Exemple

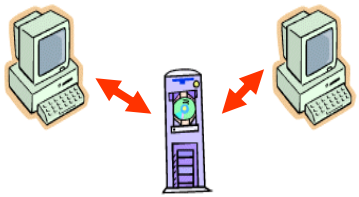
Considérons la procédure suivante de mise à jour d'un compte bancaire d'un client. Pour simplifier, on considère qu'un compte est implémenté par un simple fichier qui contient le solde courant du client:

```
Procedure crediter_compte( entier numero_client, entier somme)
    entier solde ;
    debut
    solde=lire_dans_fichier(numero_client);
     /* lecture du solde dans le fichier du client */
    solde = solde + somme ;
    ecrire_dans_fichier(numero_client, solde) ;
    /* écrire dans le fichier le nouveau solde */
    fin;
```

Supposons maintenant que cette même procédure est exécutée simultanément par deux processus P0 et P1 dans un système d'exploitation à temps partagé. Sans faire aucune hypothèse sur la durée du quantum, on peut faire les exécution suivantes :

P0 : crediter_compte(1233, 1000)	**P1 :** crediter_compte(1233, 500)
solde = lire_dans_fichier(1233) ;	solde = lire_dans_fichier(1233) ;
/* P0 bloqué car P1 s'exécute */	/* P1 bloqué car P0 s'exécute */
solde=solde+1000 ; Ecrire_dans_fichier(1233,solde) ;	solde=solde+500 ; Ecrire_dans_fichier(1233,solde) ;

Comme le montre cet exemple, le résultat final n'est pas le résultat escompté (*le client a perdu 500F dans cet affaire* ).



P1

P2

Interruption ou retard

M. Leblanc demande une réservation d'avion

M. Guy demande une réservation d'avion

Base de données dit que fauteuil 30A est disponible

Base de données dit que fauteuil 30A est disponible

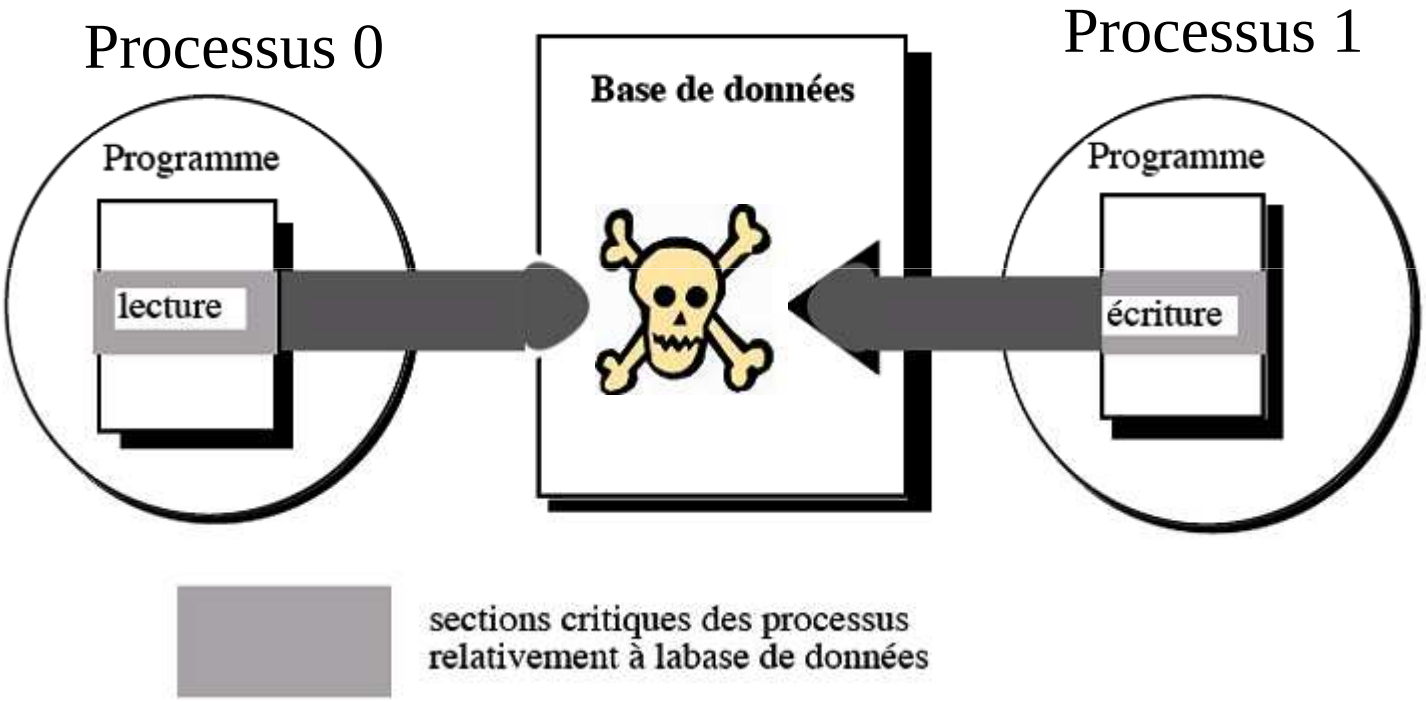
Fauteuil 30A est assigné à Leblanc et marqué occupé

Fauteuil 30A est assigné à Guy et marqué occupé

Sections critiques

Le problème dans le programme précédent est dû aux conflits d'accès au même fichier. Ces accès sont en lecture et en écriture. Evidemment, il ne peut y avoir de conflit si les accès aux données ne sont qu'en lecture seule.

* On appelle **section critique** la partie d'un programme où se produit le conflit d'accès



* Comment éviter ces conflits d'accès ?

♦ Il faut trouver un moyen d'interdire la lecture ou l'écriture des données partagées à plus d'un processus à la fois

♦ Il faut une **exclusion mutuelle** qui empêche les autres processus d'accéder à des données partagées si celles-ci sont en train d'être utilisées par un processus

♦ Dans l'exemple précédent, il faut obliger le processus P1 à attendre la terminaison de P0 avant d'exécuter à son tour la même procédure



Pour une bonne coopération entre processus

Pour que les processus qui partagent des objets (données en mémoire centrale ou dans fichiers) puissent coopérer correctement et efficacement, quatre conditions sont nécessaires :

1. **Exclusion mutuelle** : deux processus ne peuvent être en même temps en section critique
2. **Famine** : aucun processus ne doit attendre trop longtemps avant d'entrer en section critique
3. **Interblocage** (*deadlock*) : aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres d'y entrer .
4. **Aucun hypothèse** ne doit être faite sur les vitesses relatives des processus

Exclusion mutuelle par attente active

Un processus désirant entrer dans une section critique doit être mis en attente si la section critique n'est pas libre

Un processus quittant la section critique doit le signaler aux autres processus

Protocole d'accès à une section critique :

<entrer_Section_Critique> /* attente si SC non libre */
<Section_Critique> /* Un seule processus en SC */
<Quitter_Section_Critique>

L'attente peut être :

Active : la procédure *entrer_Section_Critique* est une boucle dont la condition est un test qui porte sur des variables indiquant la présence ou non d'un processus en Section Critique

Non active : le processus passe dans l'état endormi et ne sera réveillé que lorsqu'il sera autorisé à entrer en section critique

Que contiennent les procédures *entrer_Section_Critique* et *quitter_Section_Critique ?*

1ère Solution : Masquage des interruptions

- Le moyen le plus simple est que chaque processus puisse masquer les interruptions avant d'entrer en section critique
  - ⇨ l'interruption horloge qui permet d'interrompre un processus lorsqu'il a épuisé son quantum (temps CPU), serait ignorée
  - ⇨ plus de commutation de processus

- Lorsqu'un processus quitte la section critique, doit restaurer les interruptions

- Solution dangereuse en mode utilisateur :

  - ➢ Si dans un processus, le programmeur a oublié de restaurer les interruptions, c'est la fin du système ☹

2ème solution : les variables de verrouillage

- Un verrou est une variable binaire partagée qui indique la présence d'un processus en Section Critique :
  - ⇨ si verrou = 0 alors la section critique est libre
  - ⇨ si verrou = 1 alors la section critique est occupée

- Procédure *entrer_Section_Critique () :*

```
void entrer_Section_Critique ()  {
if (verrou == 0) verrou=1 ;
else while (verrou == 1) ;    /* attente active */
verrou=1 ;
}
```

- Procédure *quitter_Section_Critique ()*

```
void quitter_Section_Critique () {
        verrou=0 ;
}
```

- L'exclusion mutuelle n'est assurée que si le test et le positionnement du verrou est ininterruptible (sinon le verrou constitue une section critique)

  - ⇨ Pour cela il faut disposer d'une instruction (test and set) qui teste et modifie le contenu d'un mot en mémoire centrale de façon indivisible.

3ème Solution : l'alternance

- On utilise une variable partagée (tour) qui mémorise le numéro du processus autorisé à entrer en section critique

- Exemple d'utilisation pour N processus :
  - ⇨ void entrer_Section_critique (int MonNumero) {
    while (tour != monNumero) ; /* attente active */
    }

  - ⇨ void quitter_Section_critique () {
    tour=(monNumero +1) % N ; /* au suivant ! */
    }

- Avantage : simple et facile à utiliser
- Inconvénient : problème de **famine**, un processus possédant *tour*, peut ne pas être intéressé immédiatement par la section critique

Solution de *Peterson* (1981)

- Pour le cas de deux processus P0 et P1 :

```
#define FAUX      0
#define VRAI      1
#define N         2

int tour ;                  /* à qui le tour */
int interesse[N] ;          /* initialisé à FAUX */

void entrer_Section_Critique (int  process)  /* n° de processus : 0 ou 1*/
{
int autre ;
autre = 1-process ;
interesse[process]=VRAI ;       /* indiquer qu'on est intéressé  */
tour = process ;                /* lever le drapeau */
while (tour == process && interesse[autre] == VRAI) ;
}

void quitter_Section_Critique(int process)
{
        interesse[process]=FAUX ;
}
```

- Pourquoi l'exclusion mutuelle est assurée par cette solution?

Réponse : Considérons la situation où les deux processus appellent *entrer_Section_Critique* simultanément. Les deux processus sauvegarderont leur numéro dans la variable *tour*. La valeur sauvegardée en dernier efface la première. Le processus qui entrera en SC est celui qui a positionné la valeur *tour* en premier.

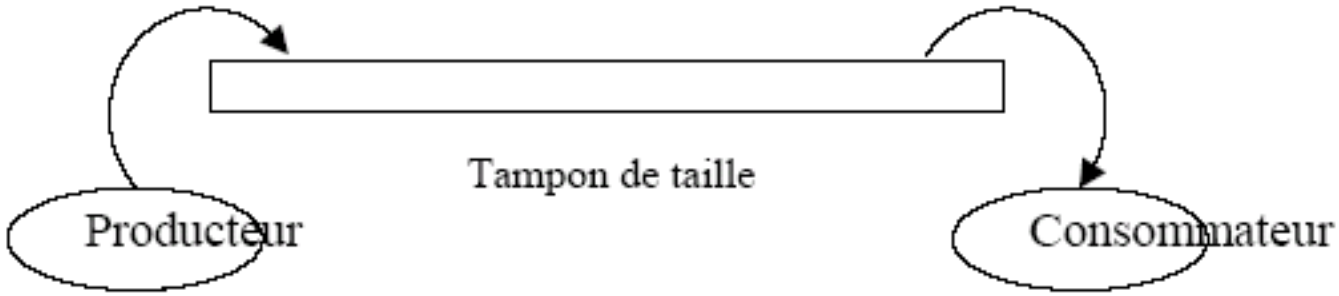
- Cette solution malgré qu'elle fonctionne bien, elle présente un gros inconvénient : elle basée sur l'attente active ; un processus ne pouvant entrer en SC utiliserait l'UC inutilement .

Solution évitant l'attente active

- **Idée :** un processus ne pouvant entrer en section critique, passe dans un état endormi, et sera réveillé lorsqu'il pourra y entrer.

  - ⇨ nécessite un mécanisme de réveil
  - ⇨ Le SE fournit deux appels système :
    - Sleep (dormir) qui suspend le processus appelant
    - Wakeup (réveiller) qui réveille le processus donné en argument

- Application au modèle Producteur/Consommateur



- Les deux processus coopèrent en partageant un même tampon
  - ⇨ Le producteur produit des objets qu'il dépose dans le tampon
  - ⇨ Le consommateur retire des objets du tampon pour les consommer

- **Conflits**
  - ⇨ Le producteur veut déposer un objet alors que le tampon est déjà plein
  - ⇨ Le consommateur veut retirer un objet du tampon alors que celui-ci est vide
  - ⇨ Le producteur et le consommateur ne doivent pas accéder simultanément au tampon

Code des processus producteur et consommateur

```
#define N 100          /* nbre d'emplacement ds tampon */

int compteur = 0 ;     /* nbre d'objets ds tampon */

void producteur () {
while (VRAI) {
        produire_objet(…) ;
        if (compteur == N) sleep () ;
        mettre_objet(…) ;
        compteur = compteur + 1 ;
        if (compteur == 1)
                wakeup(consommateur)
        }
}
```

```
void consommateur () {
while (TRUE) {
        if (compteur == 0) sleep() ;
        retirer_objet(…)
        compteur = compteur – 1 ;
        if (compteur == N-1)
                wakeup (producteur) ;
        consommer_objet(…) ;
        }
}
```

- Analyse de cette solution :
  1. L'accès à la variable *compteur* n'est pas protégé, ce qui peut entraîner des incohérences dans les valeurs prises par cette variable
  2. Réveils perdus : c'est le principal défaut de ce mécanisme. Un signal wakeup envoyé à un processus qui ne dort pas (encore) est perdu.