SMI - S4

Chapitre 1:
Introduction aux systèmes d'exploitation

Cours donné par:
Pr. N. ALIOUA

Année universitaire:
2019-2020

Références bibliographiques

1. ***Modern operating systems fourth edition***, Andrew s. Tanenbaum, Herbert bos.
2. ***Cours et exercices corrigés, systèmes d'exploitation***, *J. Archer Harris, ediscience.*
3. ***Conception de systèmes d'exploitation le cas linux***, Patrick Cegielski, deuxième édition.
4. ***Systèmes d'exploitation des ordinateurs: principes de conception***, CROCUS, DUNOD.
5. ***Operating system concepts***, Abraham Silberschatz and al., John wiley & sons.
6. ***Cours de systèmes d'exploitation***, Pr. Hafid Bourzoufi, ISTV Valenciennes, France.
7. ***Cours de systèmes d'exploitation***, Pr. Audrey Queudet, université de Nantes.
8. ***Cours Operating Systems***, Pr. Geoffrey Challen, buffalo university, New York, USA.

Quelques règles à retenir SVP

- Contact: nawal.alioua@gmail.com
- Les TPs seront réalisés sur les stations de travail Linux (obligatoirement en équipe de 2 étudiants).
- Les TPs pourront être programmés en C
- Pour chaque TP, remettre un exécutable et un rapport. Le rapport doit contenir une introduction, le code source, les résultats obtenus, les réponses aux questions et une courte conclusion qui commente les résultats. S'il y a lieu, d'autres points spécifiés dans l'énoncé du TP doivent être inclus dans le rapport.
- Les TPs doivent être réalisés par les étudiants eux-mêmes, en utilisant au maximum un apprentissage par recherche de documentation sur Internet et dans les « man » pages de Linux.

Motivations

- Les systèmes d'exploitation (SE) au quotidien:
  - Machines à laver, consoles de jeux, Smartphones, ordinateurs, SmartTV…
  - «Le système démarre», «Le système a planté» , « version 3.0.1 du système », « bug système »,

- Académique/Professionnel:
  - Génie Logiciel: le logiciel s'appuie sur un SE
  - Administration réseau: Les machines du réseaux (PCs, routeurs, serveurs, …) possèdent des SE!

- Mais encore: Les SE ont été particulièrement importants dans le développement de l'informatique, à côté de l'évolution technologique des ressources matérielles.

Motivations

- L'étude des SE permet d'accéder à des formations avancées (Systèmes distribués, virtualisation, cloud computing, Big Data …).
- *un ordinateur se compose d'éléments matériels « Hardware » et d'éléments logiciels « Software ».*

Cependant, comment coexistent-ils?
Quel est le rôle joué par le SE?
Comment réalise-t-il ce rôle?

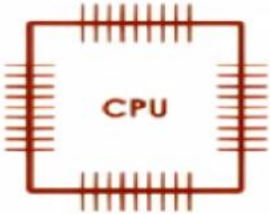
- Un SE est fortement lié aux ressources matérielles sur lesquelles il s'exécute.
- Il **exploite l'ensemble des instructions** exécutables par l'ordinateur (càd le matériel) et **gère ses ressources**.

Chapitre 1: introduction aux SE

1. Rappel: point sur le matériel
2. Rôles d'un SE
3. Définition d'un SE
4. Structure interne des SE
5. Le noyau
6. Quelques familles de SE
7. La notion de multitâches multi-utilisateurs

1. Rappel: point sur le matériel
2. Rôles d’un SE
3. Définition d’un SE
4. Structure interne des SE
5. Le noyau
6. Quelques familles de SE
7. La notion de multitâches multi-utilisateurs

1. Rappel: point sur le matériel!



1)Processeur:

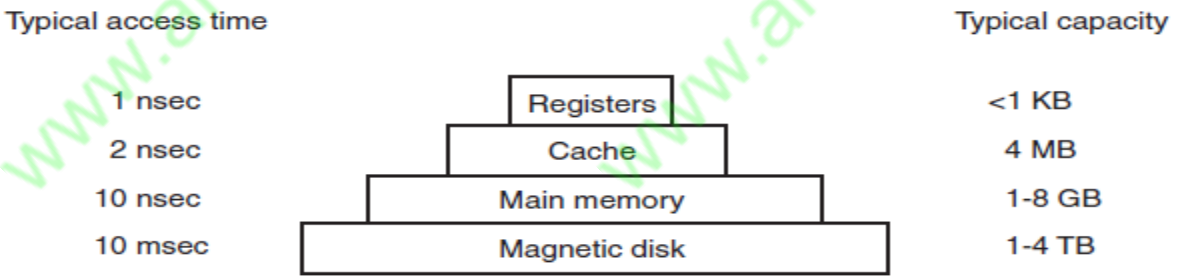
L'Unité Centrale de Traitement (UCT) ou processeur central (CPU), est « le cerveau » de l'ordinateur qui interprète et exécute les instructions du programme situées en mémoire centrale.

•L'UCT est composée de l'Unité arithmétique et logique (UAL) et de l'Unité de commande (ou de contrôle).

- •L'UAL effectue les opérations arithmétiques et logiques.
- •L'Unité de commande dirige le fonctionnement de toutes les autres unités: UAL, mémoire, entrées / sorties, etc., en leur fournissant les signaux de cadence (l'horloge) et de commande.

1. Rappel: point sur le matériel!

2)Mémoire:

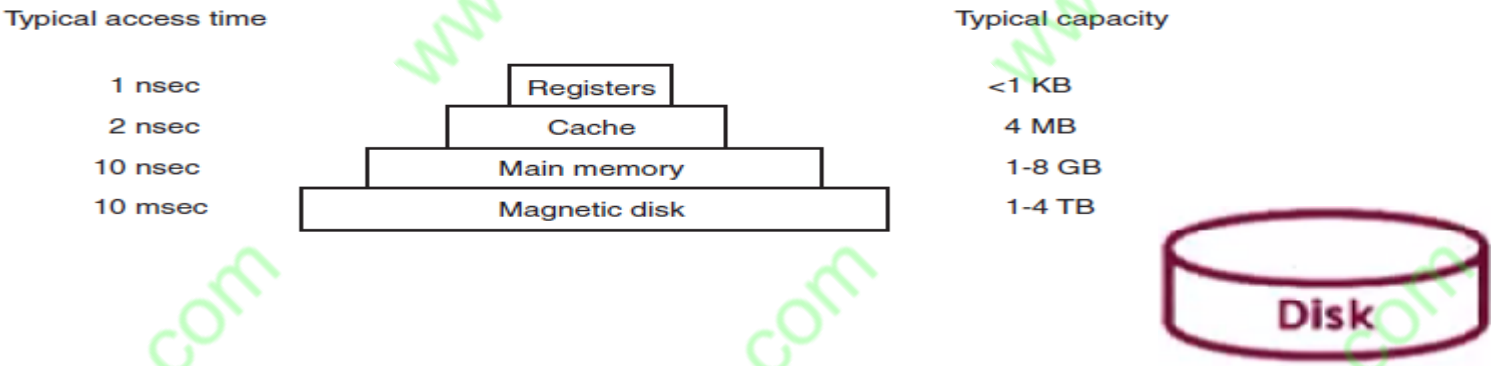


- Les registres:
  - type de mémoire qui sert à stocker des données à traiter, des résultats intermédiaires ou des informations de commande.
  - Le temps d'accès est minimal, très petit espace de stockage et à prix élever.
- La mémoire cache:
  - Un bloc de cette mémoire est appelé **ligne de cache** qui est composée de plusieurs **mots mémoire** consécutifs en mémoire.
  - Le processeur essaie d'accéder à un mot d'abord dans la cache, avant de passer à la mémoire principale:
    - En cas d'échec (miss), le mot est gardé dans la cache pour un accès futur.
    - En cas de succès (hit), la mémoire principale n'est pas accédée.
  - mémoire très rapide, de petite taille.

- La mémoire vive:
  - RAM (Random Access Memory), on **peut accéder instantanément à n'importe quelle espace mémoire**.
  - Volatile et contient les données et les instructions des applications en cours.
  - Mémoire rapide, de taille plus importante et à prix moyen.

1. Rappel: point sur le matériel!

2)Mémoire:



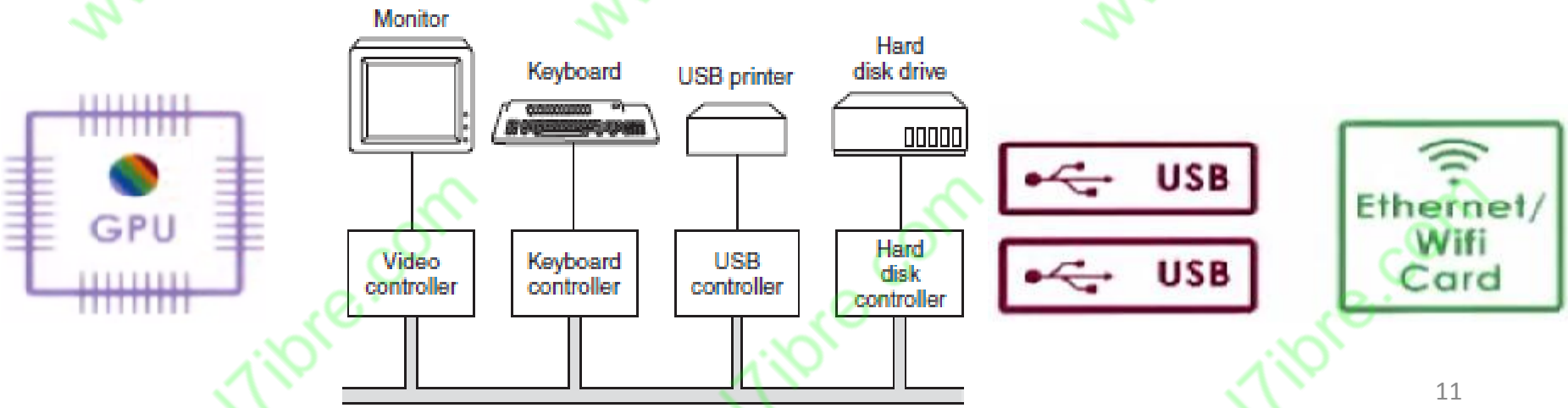
- Le disque:
  - sert principalement à stocker les données d'une manière non-volatile.
  - L'accès est mécanique, introduisant un délai important en lecture/écriture relativement à la RAM.
  - Sert éventuellement à « étendre » la RAM.
  - Un espace plus important et à coût bas (selon les technologies).
- La mémoire morte:
  - ROM (Read Only Memory) **mémoire en lecture seule**.
  - Permanente, contenant des microprogrammes enregistrés à l'usine sur des puces électroniques de la carte mère, contenant les routines de démarrage de l'ordinateur.
  - C'est une mémoire non volatile, rapide et à bas coût.

1. Rappel: point sur le matériel!

3)Les périphériques d'entrée/sortie (E/S):

- Permettent le dialogue (échange d'informations) avec ce qui se trouve à l'extérieur de la machine.

- Ils se composent généralement de deux parties:
  - Le matériel (physique).
  - Le contrôleur: une puce ou un ensemble de puces qui contrôle physiquement le périphérique. Présente au système d'exploitation une interface « simple », nommée Driver, pour recevoir les commandes et retourner leurs résultats.



1. Rappel: point sur le matériel!

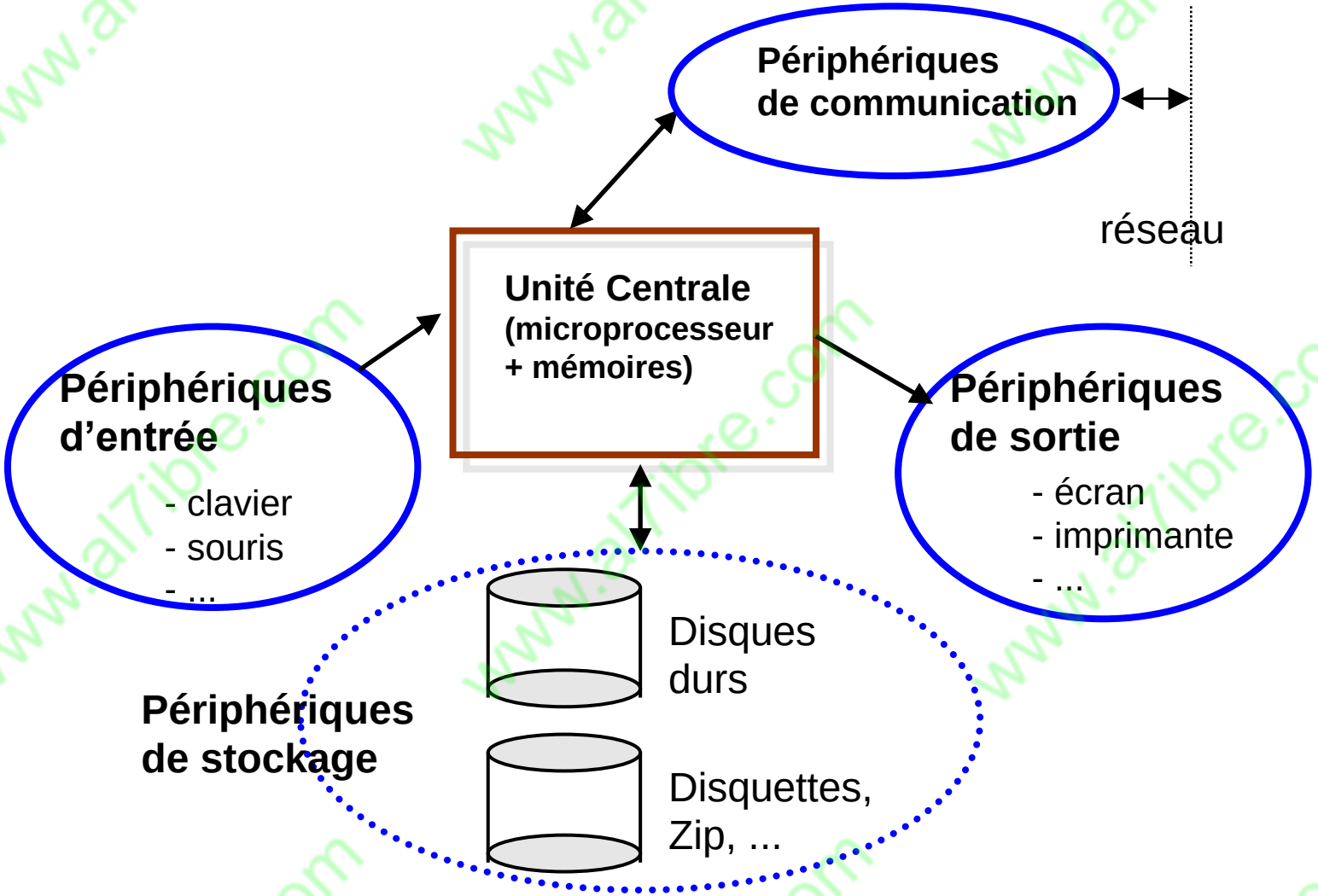
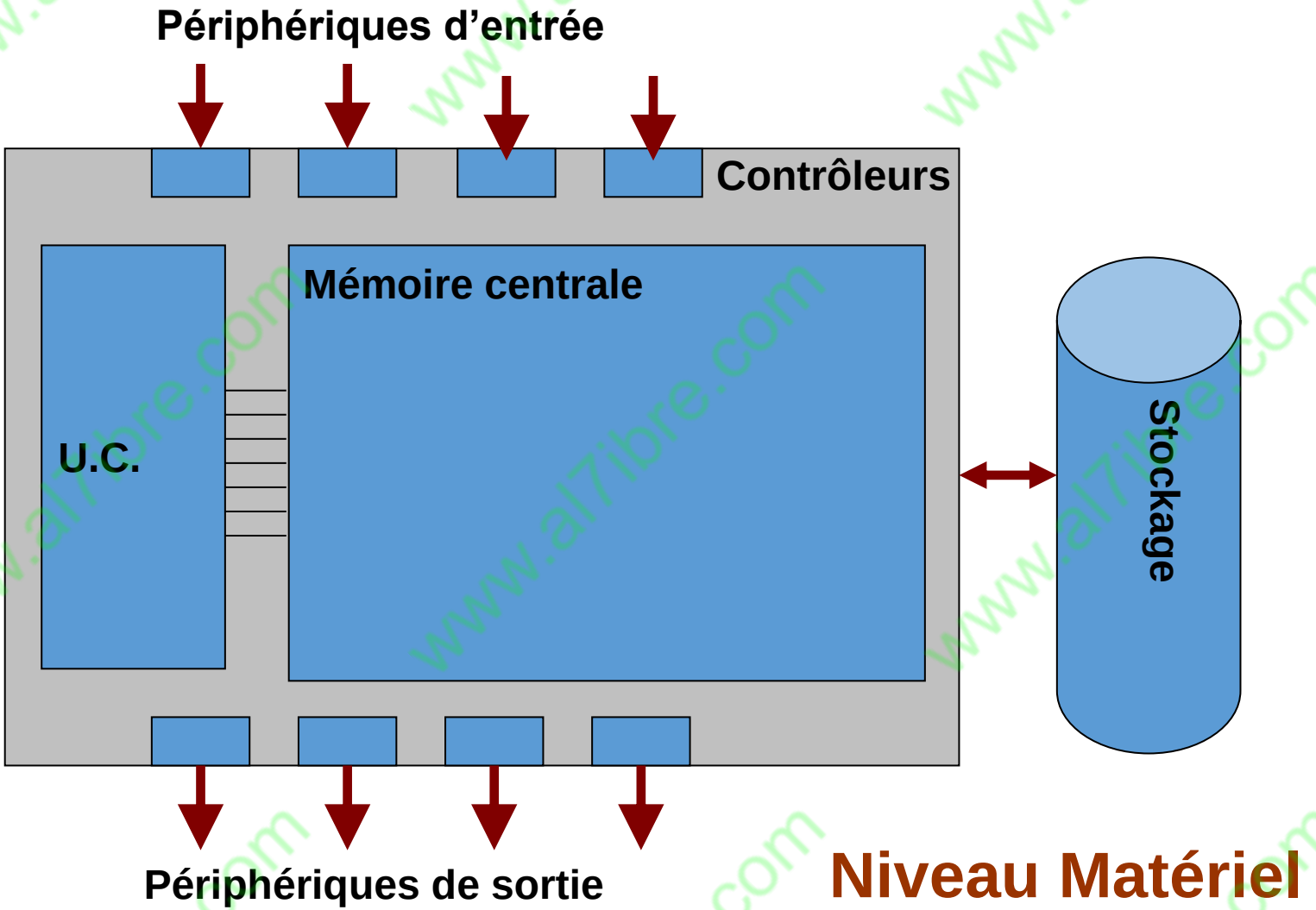


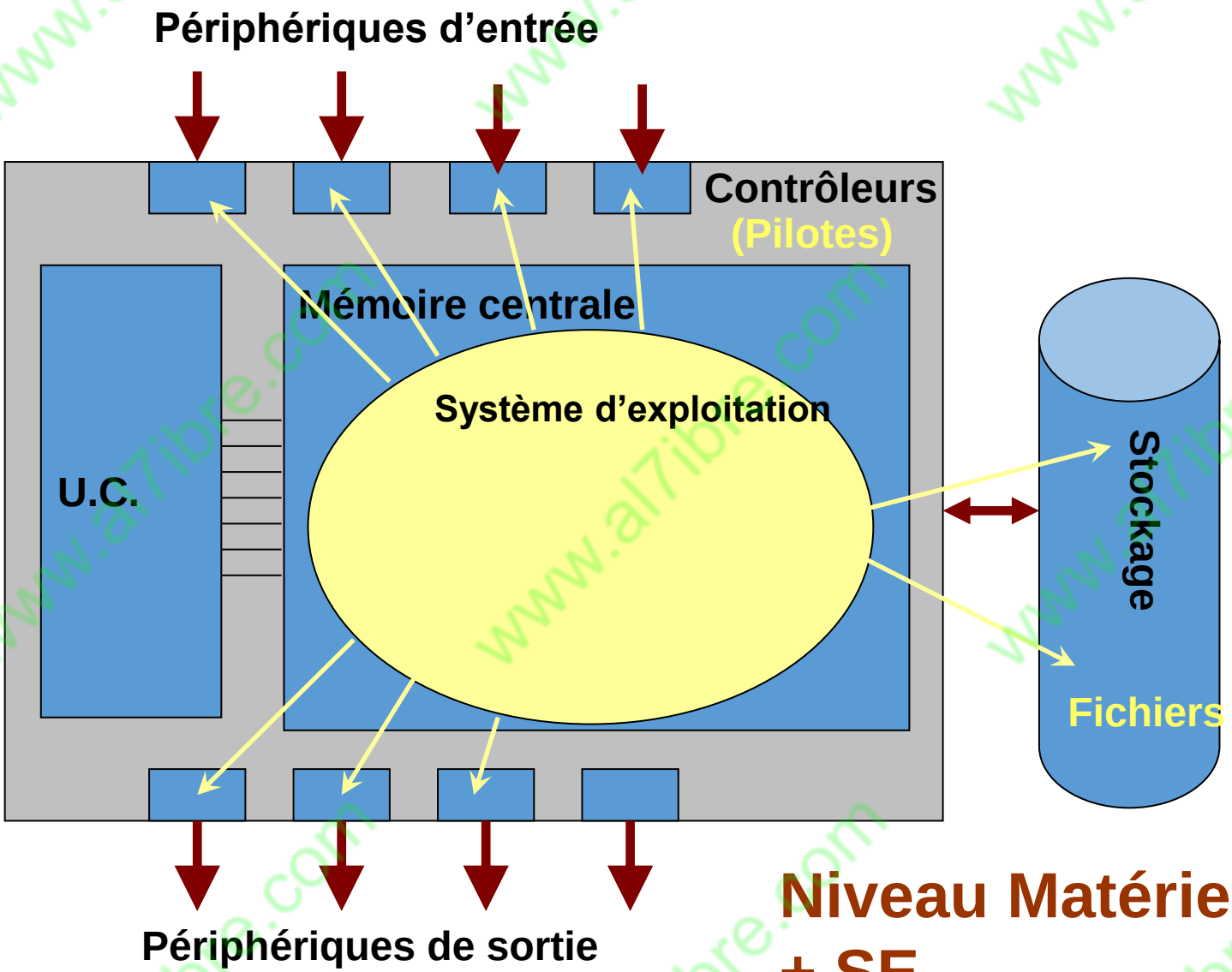
Schéma matériel général

1. Rappel: point sur le matériel!



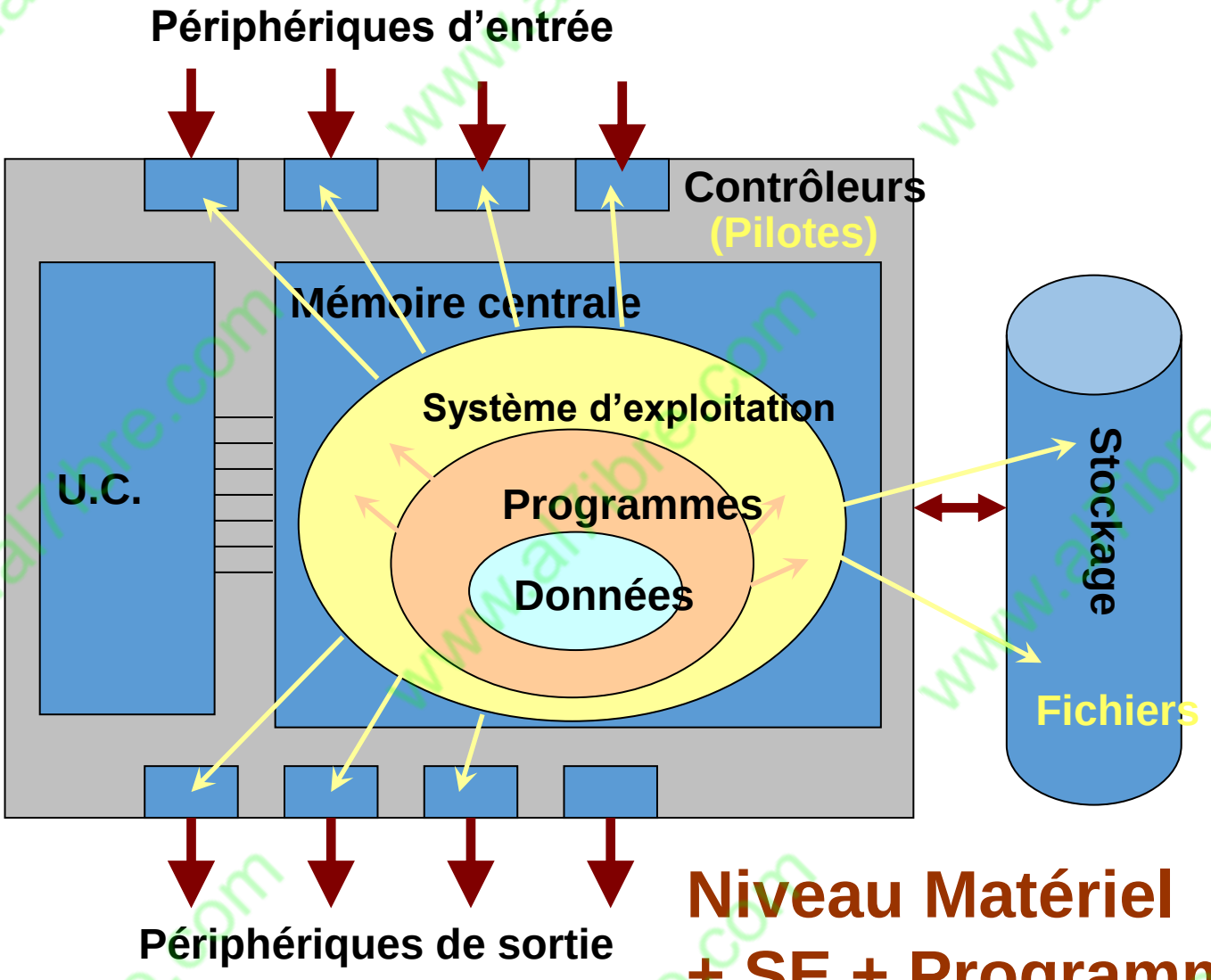
Niveau Matériel

1. Rappel: point sur le matériel!



Niveau Matériel
+ SE

1. Rappel: point sur le matériel!

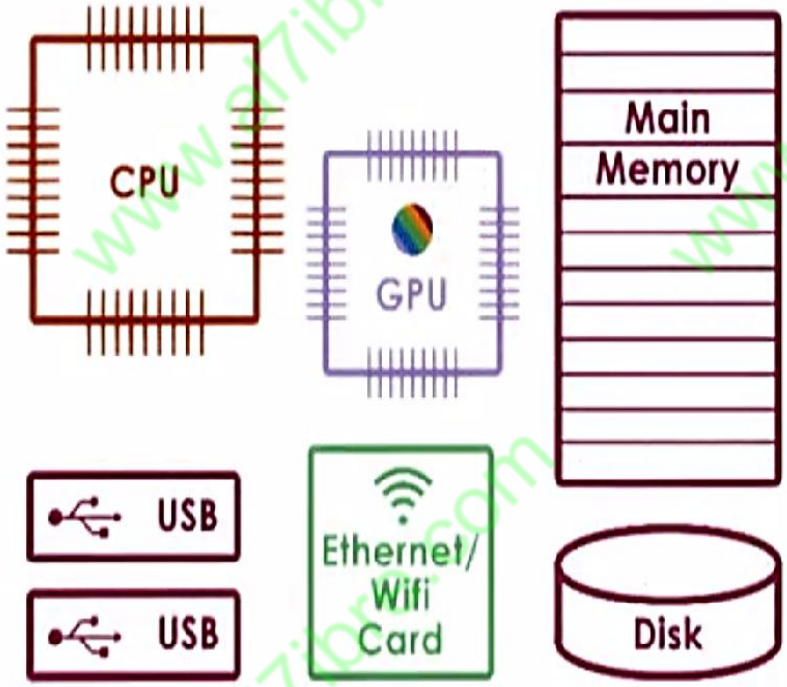


Niveau Matériel
+ SE + Programmes

1. Rappel: point sur le matériel
2. Rôle d'un SE
3. Définition d'un SE
4. Structure interne des SE
5. Le noyau
6. Quelques familles de SE
7. La notion de multitâches multi-utilisateurs

2. Rôle d'un SE

« Il est plus facile de définir un système d'exploitation par ce qu'il fait que par ce qu'il est. »



Définition : ensemble de programmes de gestion du système qui permettent de gérer les éléments fondamentaux de l'ordinateur: Le matériel , les logiciels , la mémoire - les données , les réseaux.

Rôles majeurs:

- Cache la complexité du hardware, que ce soit pour les applications ou pour les développeurs: diverses technologies de stockage, de transmission, … .
- Gère les ressources: allocation de mémoire, ordre d'exécution.
- Isolation et protection: des programmes, des utilisateurs, …

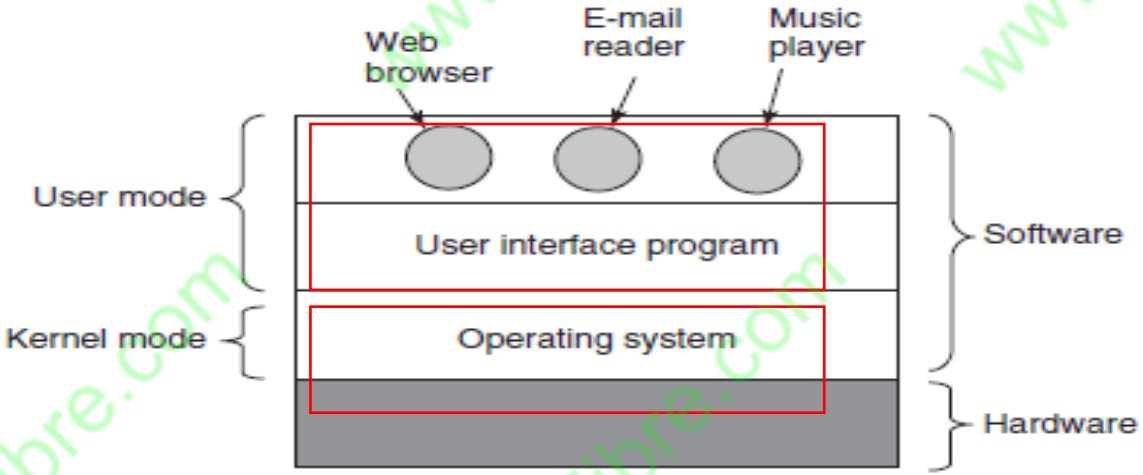
2. Rôle d'un SE

- D'une manière simple: simplifier l'interaction avec le matériel puis contrôler et gérer son utilisation.
- D'une manière plus soutenue (académique):
  - La gestion des processus: un processus est un programme en cours d' exécution. Le SE doit gérer l'allocation de ressources aux processus en proposant à chacun un environnement dans lequel il peut s'exécuter en toute sécurité.
  - La gestion de la mémoire : Le SE doit gérer l'allocation de mémoire aux processus et contrôler physiquement les emplacements auxquels peut accéder un processus.
  - La gestion des périphériques: Le SE doit gérer les périphériques afin de leur permettre d'être partagés de manière efficace entre les processus.
  - La gestion du système de fichiers : Le SE doit gérer des structures de données permettant de créer, stocker, supprimer, lire (etc. …) les informations et de les organiser dans des fichiers sur des mémoires secondaires (disque dur, CD-ROM, clé USB, disques SSD, etc.).
  - La protection: l'accès aux données doit être réglementé puisqu'il existe plusieurs utilisateurs et processus. Le SE doit garantir que les fichiers, les segments de mémoire, etc … ne peuvent être utilisés que par les processus ayant obtenu l'autorisation appropriée.
  - La sécurité: Le SE doit posséder des mécanismes pour se défendre contre les attaques externes et internes. Elles comprennent les virus, des attaques de déni de service (Denial of Service: DoS).

1. Rappel: point sur le matériel
2. Rôles d'un SE
3. Définition d'un SE
4. Structure interne des SE
5. Le noyau
6. Quelques familles de SE
7. La notion de multitâches multi-utilisateurs

3. définition d'un SE

- ***Définition:*** *Un système d'exploitation est une couche logicielle indispensable pour exploiter, d'une manière simple, les ressources matériels d'un ordinateur.*



- Deux modes de fonctionnement (en général):
  - le mode noyau (mode superviseur, privilégié): dispose d'un accès complet à tout le matériel et peut exécuter toutes les instructions que la machine est capable d'exécuter.
  - Le mode utilisateur: le reste du système s'exécute en mode utilisateur, dans lequel seul un sous-ensemble des instructions de la machine est exécutable.
- Le programme d'interface utilisateur: le shell ou l'interface graphique, est le niveau le plus bas du logiciel en mode utilisateur, et permet à l'utilisateur de démarrer d'autres programmes, comme un navigateur Web, un lecteur de courrier électronique ou un lecteur de musique…

I. Généralités sur les SE

1. Rappel: point sur le matériel
2. Définition d'un SE
3. Rôles d'un SE
4. Structure interne des SE
5. Le noyau
6. Quelques familles de SE
7. La notion de multitâches multi-utilisateurs

4. Structures internes de SE

- Monolithique (d'un seul bloc) :
  - L'ensemble du SE s'exécute en un seul programme en mode noyau.
  - Le SE est écrit comme une collection de procédures, reliées entre elles dans un seul programme binaire exécutable unique.

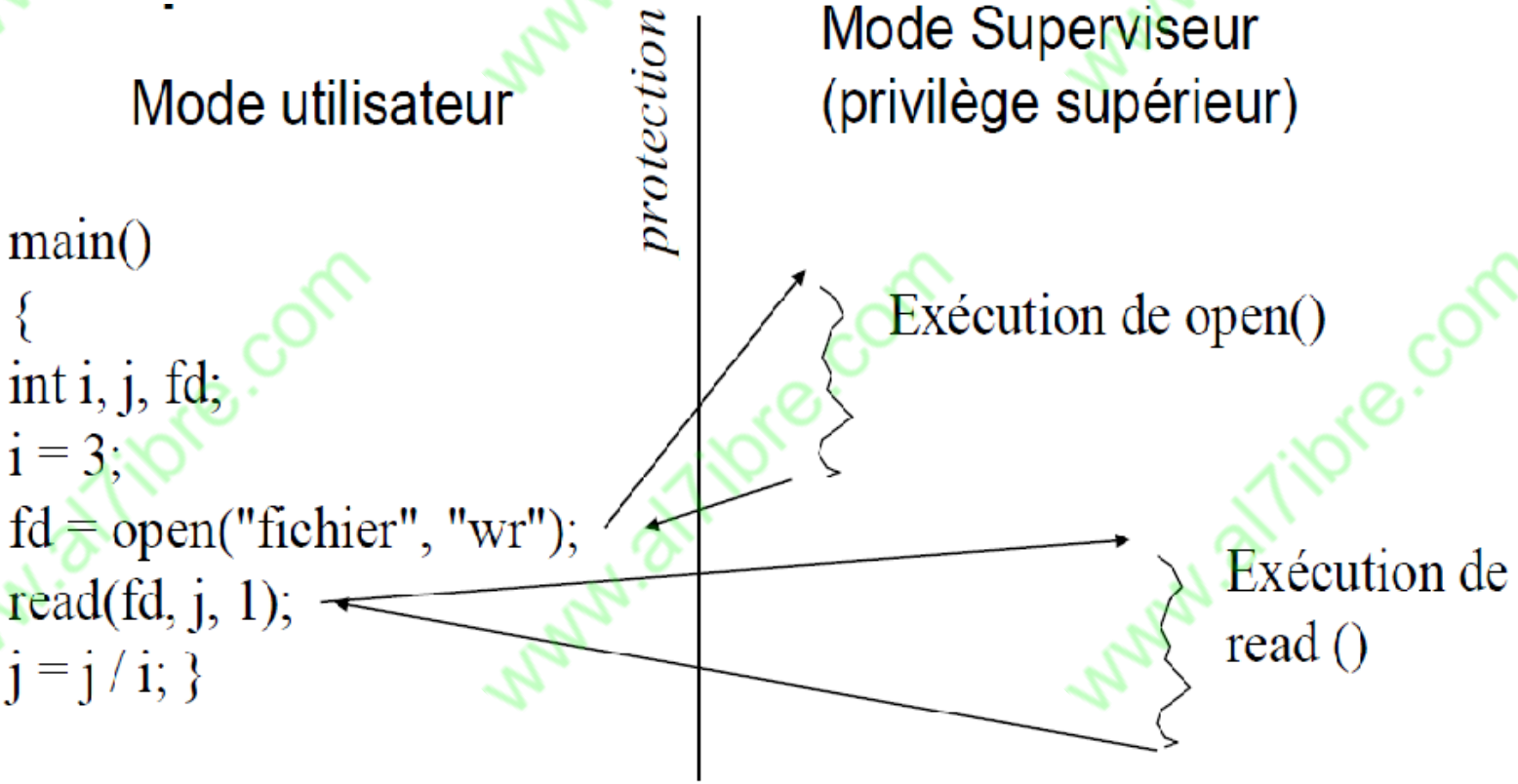
  ✓ Implémentation simple.

  ↓ Difficile à maintenir!
  - MS-DOS est un exemple d'un tel système.

- Systèmes à modes noyau et utilisateur:
  - Le SE démarre en mode noyau, ce qui permet d'initialiser les périphériques et de mettre en place des routines de service, et commute ensuite en mode utilisateur.
  - En mode utilisateur, on utilise les **appels système** pour avoir accès à ce qui a été prévu par le système.
  - Unix et Windows (tout au moins depuis Windows 95) sont de tels systèmes.

4. Structures internes de SE

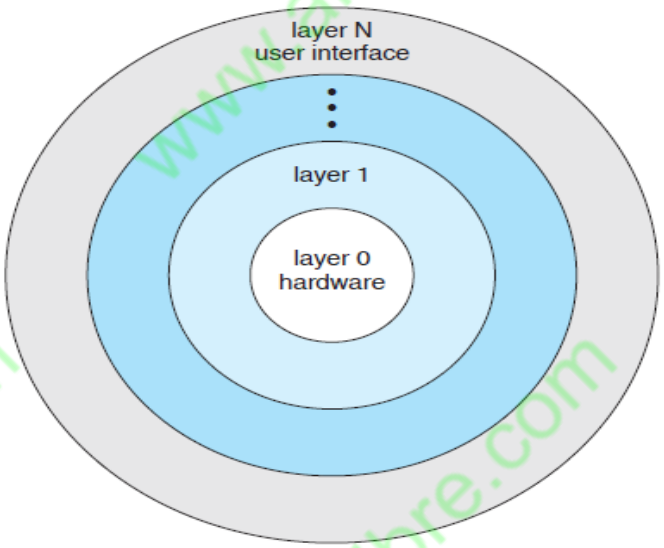
- Systèmes à modes noyau et utilisateur: Appel système



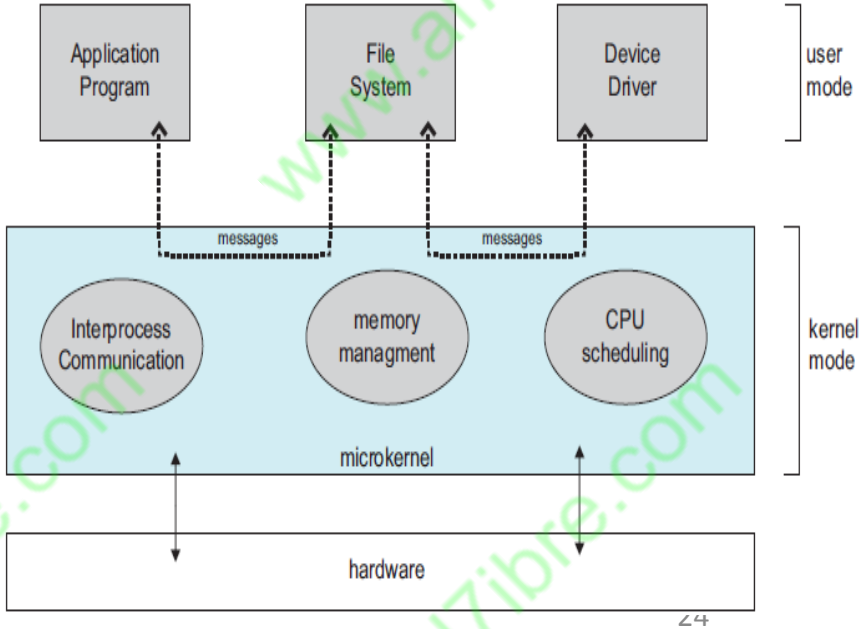
- **1)** Le noyau reçoit l'appel système,
- **2)** Vérifie qu'il s'agit d'une demande valable (en particulier du point de vue des droits d'accès),
- **3)** Exécute,
- **4)** Renvoie au mode utilisateur.

4. Structures internes de SE

- Systèmes en couches (généralisation):
  - Chaque couche réalise sa fonction en s'appuyant exclusivement sur l'ensemble des fonctions de la couche qui lui est immédiatement inférieure.
  - Une couche typique du SE consiste en des structures de données et un ensemble de routines pouvant être invoquées par des couches de niveau supérieur.
  - ✓ Vérification et débogage relativement simples.
  - ↓ Difficulté lors de la définition des couches; tendance à être moins efficaces.



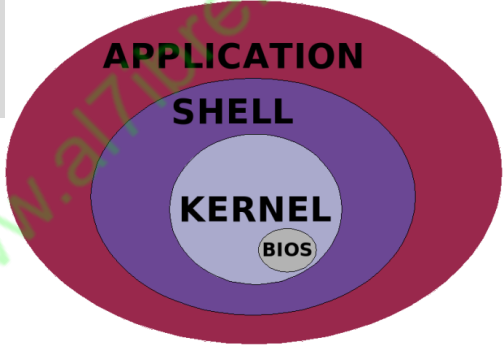
- Systèmes à micro Kernel:
  - Un noyau minimal (appelé micronoyau) de taille de code réduite.
  - Se contente en général de gestionnaires de tâches et de mémoire simples, et un mécanisme de communication entre processus.
  - Les gestionnaires de périphériques, les gestionnaires d'appels système, etc… sont implémentés en tant que programmes système et utilisateurs.
  - ✓ Un SE plus simple à étendre.
  - ↓ Problèmes de performance.



I. Généralités sur les SE

1. Rappel: point sur le matériel
2. Définition d'un SE
3. Rôles d'un SE
4. Structure interne des SE
5. Le noyau
6. Quelques familles de SE
7. La notion de multitâches multi-utilisateurs

5. Le noyau



- Le noyau (kernel en anglais) comporte un certain nombre des plus importantes routines (sous-programmes) du SE. Il est chargé en mémoire vive à l'initialisation du système. Les autres routines, moins critiques, sont appelées des **utilitaires**.
- Le noyau d'un système d'exploitation se compose de quatre parties principales : **le gestionnaire de tâches** (ou des processus), **le gestionnaire de mémoire**, **le gestionnaire de fichiers** et **le gestionnaire de périphériques d'entrée-sortie**.
- Il possède également deux parties auxiliaires : **le chargeur du système d'exploitation** et **l'interpréteur de commandes**.
  - **Le chargeur du système d'exploitation:**
    - Appelé, pour PC et MAC, BIOS (pour Basic Input Output System) et est chargé à une adresse bien déterminée en mémoire RAM.
    - Ce logiciel initialise les périphériques, charge un secteur du disque, et exécute ce qui y est placé.
  - **L'interpréteur de commandes (shell en anglais):**
    - Est souvent considéré comme une partie du SE
    - Exécute une boucle infinie qui affiche une invite (montrant par là que l'on attend quelque chose), lit le nom du programme et les paramètres saisis par l'utilisateur à ce moment-là et l'exécute.

5. Le noyau

- Gestionnaire de tâches (**ordonnanceur)** :
  - Divise le temps en laps de temps (en anglais **slices**, tranches),
  - Décide périodiquement d'interrompre le processus en cours et de démarrer (ou reprendre) l'exécution d'un autre.
- Gestionnaire de mémoire:
  - Connaître les parties libres et les parties occupées de la mémoire,
  - Allouer de la mémoire aux processus qui en ont besoin,
  - Récupérer la mémoire utilisée par un processus lorsque celui-ci se termine,
  - Traiter le va-et-vient entre le disque et la mémoire principale lorsque cette dernière doit être étendue.
- Gestionnaire de fichiers:
  - Faire abstraction des spécificités des disques et des autres périphériques d'entrée-sortie,
  - Offrir au programmeur un modèle agréable et facile d'emploi.
- Gestionnaire de périphériques:
  - Envoyer les commandes aux périphériques,
  - Intercepter les interruptions,
  - Traiter les erreurs.

1. Rappel: point sur le matériel
2. Définition d'un SE
3. Rôles d'un SE
4. Structure interne des SE
5. Le noyau
6. Quelques familles de SE
7. La notion de multitâches multi-utilisateurs

6. Quelques familles de SE

- L'existence des SE depuis fort longtemps ainsi que leur utilisation dans différents domaines technologiques ont permis l'émergence de plusieurs familles de SE.
- Mainframe*:
  - Ordinateurs de taille d'une pièce qu'on trouve dans les centres de données d'entreprise.
  - Grandes capacités: 1000 disques et des millions de gigaoctets de données.
  - Les SE sont amenés à traiter plusieurs tâches à la fois, dont la plupart nécessitent des quantités importantes d'E/S.

6. Quelques familles de SE

- Serveurs:
  - PC à ressources importantes ou des workstations ou des Mainframes.
  - Les SE doivent être capables de gérer plusieurs utilisateurs à la fois sur un réseau et de partager des ressources matérielles et logicielles.

- Multiprocesseurs:
  - Un moyen pour obtenir une puissance de calcul majeure est de connecter plusieurs CPU dans un seul système.
  - Ces systèmes ont besoin de SE assez spéciaux, des variations de SE pour serveurs.

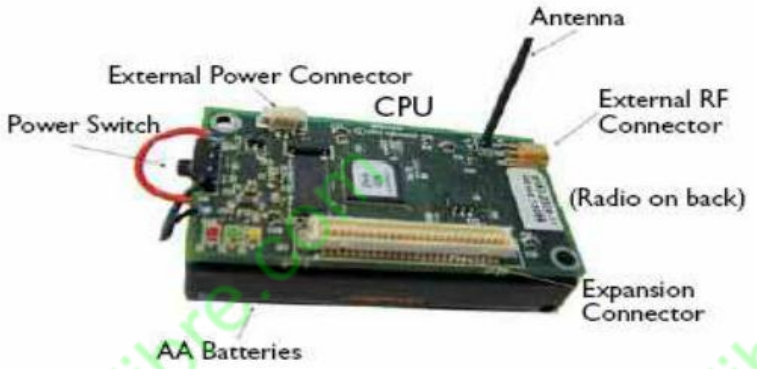
6. Quelques familles de SE

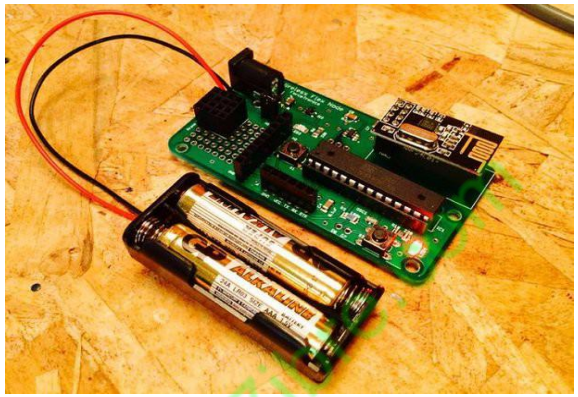
- **Personal Computer:** Les SE pour PC modernes supportent la multiprogrammation, pour fournir un bon support à un seul utilisateur.

- **Handheld:**
  - Connu auparavant sous le nom de PDA (Personal Digital Assistant), ancêtre des smartphones et des tablettes.
  - La plupart de ces appareils possèdent des CPU multicoeurs, GPS, caméras, capteurs…
  - Ils ont des exigences particulières ce qui nécessite des SE sophistiqués et bien adaptés.

6. Quelques familles de SE

- Les systèmes embarqués:
  - S'exécutent sur des circuits pour des périphériques n'acceptant pas de futures installations de logiciels par l'utilisateur (Ex: fours à micro-ondes, les téléviseurs, les voitures, etc): tout est en ROM.
  - Ces SE n'ont pas besoin de considérer l'aspect de protection contre les logiciels malveillants, ce qui entraîne une simplification de la conception.
- Nœud capteur*:
  - Unités qui composent un réseau de capteurs sans fil.
  - Chaque nœud est composé de CPU, RAM, ROM, capteurs (température, humidité, lumière, mouvement …): un vrai ordinateur.
  - Le SE doit être petit et simple car les nœuds ont peu de RAM et sont contraints en énergie.





6. Quelques familles de SE

- Systèmes temps réel*:
  - Se caractérisent par le temps (ou le délai) comme contrainte.
  - Si le système doit fournir des garanties absolues qu'une certaine action se produira à un certain moment (ou dans un certain intervalle temps), on parle d'un système en temps réel strict (**Hard real-time**).
  - Si le système doit fournir des garanties avec une certaine probabilité et ainsi un délai plus grand mais acceptable, on parle de temps réel souple (**Soft real-time**).

1. Rappel: point sur le matériel
2. Définition d'un SE
3. Rôles d'un SE
4. Structure interne des SE
5. Le noyau
6. Quelques familles de SE
7. La notion de multitâches multi-utilisateurs

7. La notion de multitâches multi-utilisateurs

- Systèmes multitâches:
  - Appelés aussi multi-programmés
  - Permettent l'exécution de plusieurs tâches à la fois : exécuter un programme utilisateur, lire les données d'un disque, afficher des résultats sur un terminal.

  - Ce contexte fait appel aux notions suivantes:
    - Processus (et non programme): un processus est une instance de programme en exécution. Le processus est représenté par un programme (le code), ses données et son état d'avancement communément appelés variables d'environnement .
    - Temps partagé: le micro-processeur à un instant donné, n'exécute réellement qu'un seul processus. Faire passer le processeur d'un processus à un autre, en exécutant chaque programme pendant quelques dizaines de millisecondes, donne l'impression que tout s'exécute en même temps : c'est **Le pseudo-parallélisme.**
    - Espace mémoire d'un processus: chaque processus possède son propre espace mémoire, non accessible aux autres processus. On parle de **l'espace d'adressage du processus**.

7. La notion de multitâches multi-utilisateurs

- Systèmes multi-utilisateurs:
  - Capables d'exécuter de façon **concurrente** et **indépendante** des applications appartenant à plusieurs utilisateurs.
  - **Concurrente?:**
    - les applications sont actives au même moment et se disputent l'accès aux différentes ressources.
  - **Indépendante?:**
    - chaque application peut réaliser son travail sans se préoccuper de ce que font les applications des autres utilisateurs.
  - Ce contexte fait appel aux notions suivantes:
    - Utilisateurs: matérialisé par un espace privé de travail sur la machine. Chaque utilisateur est identifié par un numéro unique, appelé l'identifiant de l'utilisateur, ou UID (pour l'anglais User IDentifier).
    - Groupe d'utilisateurs: permet de partager de façon sélective le matériel avec d'autres utilisateurs. Un groupe est également identifié par un numéro unique dénommé identifiant de groupe, ou GID (pour l'anglais Group IDentifier).
    - Super-utilisateur: ou encore superviseur (root en anglais), un utilisateur particulier qui peut pratiquement tout faire dans la mesure où le SE ne lui applique jamais les mécanismes de protection. Il peut, en particulier, accéder à tous les fichiers du système et interférer sur l'activité de n'importe quel processus en cours d'exécution.

SMI - S4

Chapitre 2:
Gestion des processus - Rappels

Cours donné par:
Pr. N. ALIOUA

Année universitaire:
2019-2020

1. Introduction et Définitions
2. Niveaux d'ordonnancement des processus
3. Etats des processus
4. Algorithmes d'ordonnancement
5. Superviseur des processus
6. Création de processus

II.1 Introduction et définitions

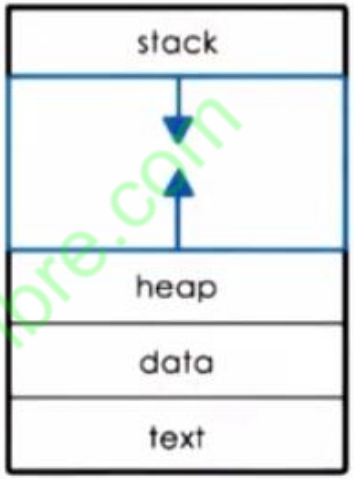
1) Qu'est-ce qu'un processus?

- Une tâche fondamentale des SE est d'assurer l'exécution de divers programmes.
- Un programme est une **entité statique** stockée dans le disque.
- Une fois chargé en mémoire pour s'exécuter, le programme devient un processus, qui est une **entité active**.
- *« Un processus peut être définit comme étant **une instance de programme en cours d'exécution ».***
- L'exécution d'un processus est en général une **alternance** de calculs effectués par l'UCT et de requêtes d'E/S effectuées par les périphériques.
- Un processus va **concourir** avec d'autres processus pour l'obtention d'une ressource (UCT, périphérique E/S,...).
- La gestion d'accès aux ressources est dirigée par la partie du SE appelée **ordonnanceur**.

II.1 Introduction et définitions

1) Qu'est-ce qu'un processus?

- Un processus est composé principalement:
  - Du code du programme (aussi appelé **section texte** du processus).
  - Du contenu des registres de l'UCT et de la valeur du compteur de programme (activité courante du processus).
  - De la pile du processus (stack), contenant les données temporaires (paramètres des fonctions, variables locales,…).
  - De la section données (data), contenant les variables globales du programme.
  - D'un tas (heap), qui est une mémoire dynamiquement allouée pendant l'exécution du processus (pour lecture de fichiers,…)



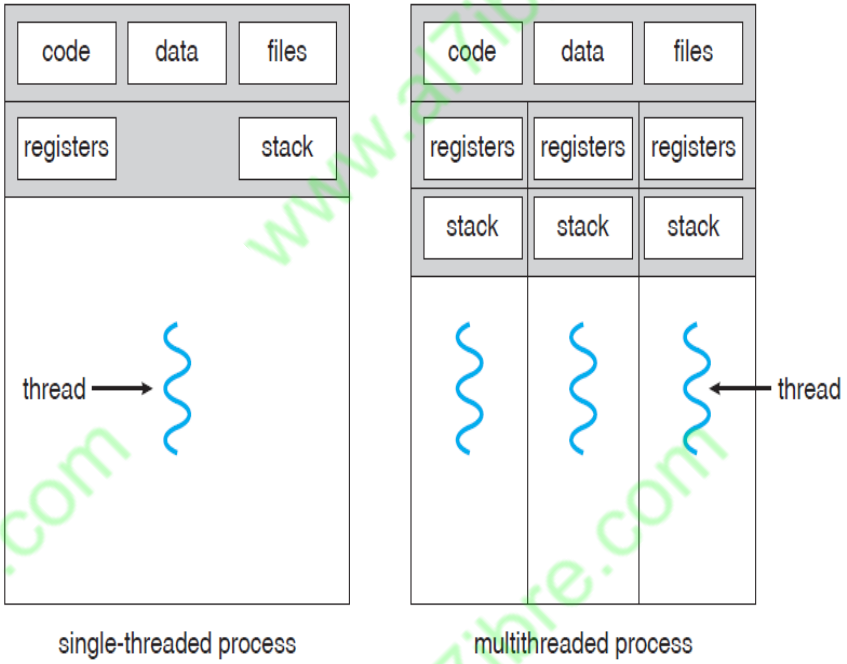
- Si un même programme est exécuté plusieurs fois, il correspond à plusieurs processus.
- Un processus peut **communiquer** des informations avec d'autres processus.

II.1 Introduction et définitions

2) Qu'est-ce qu'un processus léger (ou thread)?

- Un processus peut être composé d'un ou de plusieurs processus légers (threads).
- *« Un **thread** est une unité d'exécution rattachée à un processus, chargée d'en exécuter une partie. »*
    - Ex: pour un même document MS-Word, plusieurs threads: Interaction avec le clavier, sauvegarde régulière du travail, contrôle d'orthographe…)

- Un processus possède un ensemble de ressources (code, fichiers, périphériques…) que ses threads partagent.
- Cependant, chaque thread dispose :
    - d'un compteur programme (pour le suivi des instructions à exécuter)
    - de registres systèmes (pour les variables de travail en cours)
    - d'une pile (pour l'historique de l'exécution)



II.1 Introduction et définitions

2) Qu'est-ce qu'un processus léger (ou thread)?

- Avantages des threads:
  - Réactivité: Le processus peut continuer à s'exécuter même si certaines de ses parties sont bloquées (en chargement de fichiers par exemple).
  - Economie d'espace mémoire: Partage de ressources, surtout la mémoire, entre threads d'un même processus.
  - Economie de temps: Les threads partagent les ressources du processus auquel ils appartiennent. Ainsi, il est plus économique de créer et de gérer les threads que les processus entre eux.
  - Scalabilité: Un processus à thread unique ne peut s'exécuter que sur une CPU. Alors qu'un processus à multithreads, peut s'exécuter sur plusieurs CPU (quand elles existent) en même temps.

II.1 Introduction et définitions

3) Catégories des SE

Après avoir défini ces notions, les SE peuvent être divisés en 3 familles:

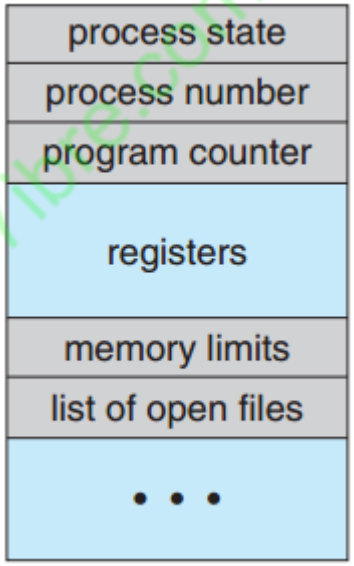
- Les SE mono-processus à thread unique(ex. DOS):
  - configuration la plus simple et la plus ancienne où un seul processus est exécuté à la fois.
- Les SE multiprocessus à thread unique (ex. Unix):
  - sur ces systèmes, l'allocation des ressources et l'ordonnancement de l'UCT agissent sur le processus et non pas les threads.
- Les SE multiprocessus multithread (ex: windows):
  - sur ces systèmes, des ressources sont allouées aux processus, mais l'ordonnancement de l'UCT agit sur les différents threads.

II.1 Introduction et définitions

4) Le bloc de contrôle du processus PCB

• Pour localiser et gérer tous les processus, le SE maintient une structure de données appelée «**table des processus**» qui contient les informations sur tous les processus créés.

• Le **Bloc de Contrôle de Processus** (Process control Bloc ou **PCB**) est une entrée dans cette table, composée principalement de:



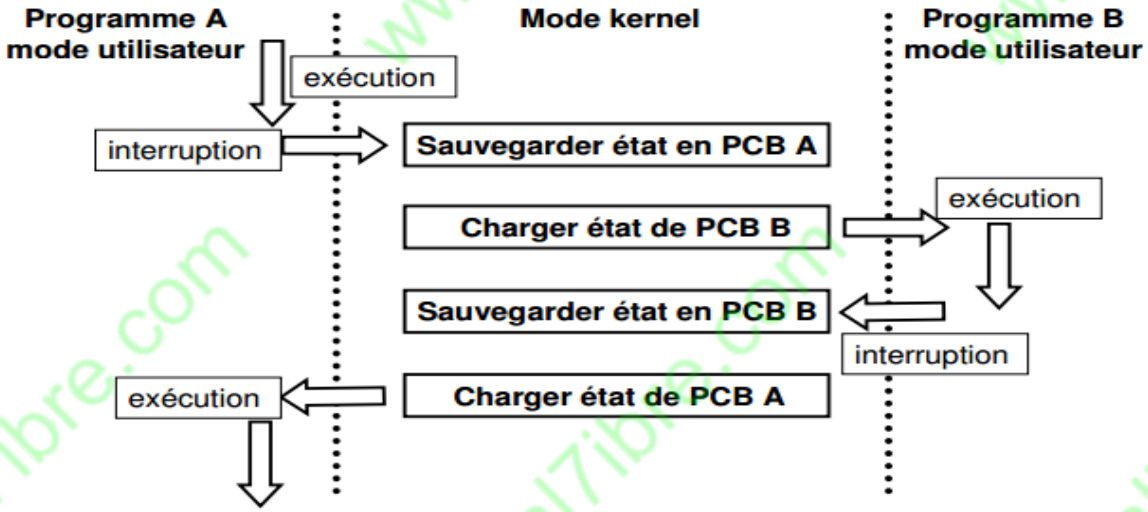
- État de processus (En exécution, prêt, bloqué, ...)
- Identifiant du processus PID (unique)
- Compteur de programme (adresse prochaine instruction à exécuter par ce processus).
- Registres de l'UCT : varient en nombre et type selon l'architecture de l'ordinateur (dont l'accumulateur, le registre d'indexe, pointeurs de pile,…)
- Information d'ordonnancement (dont la priorité du processus, pointeurs sur les files d'ordonnancement, …)
- Information sur la gestion de la mémoire (dont les limites de la mémoire attribuée au processus).
- Information sur le statut des E/S (dont la liste des périphériques d'E/S alloués au processus)

II.1 Introduction et définitions

5) Le changement de contexte

Pourquoi a-t-on besoin de toutes ces données (càd PCB)?

- Dans un système multiprogrammé, on a souvent besoin d'interrompre un processus et de redonner le contrôle de l'UCT à un autre processus.
- Il faut mémoriser toutes les informations nécessaires pour pouvoir relancer le processus courant dans le même état.
- Le processus en cours est interrompu et un ordonnanceur est appelé. Ce dernier s'exécute en mode noyau (kernel) pour pouvoir manipuler les PCB.
- Le changement de contexte a un coût: il va consommer de la mémoire et des cycles UCT, pour décharger le PCB du processus qui était en cours d'exécution et charger le PCB du processus qui va s'exécuter.



1. Introduction et Définitions
2. Niveaux d'ordonnancement des processus
3. Etats des processus
4. Algorithmes d'ordonnancement
5. Superviseur des processus
6. Création de processus

II.2 Niveaux d'ordonnancement des processus

Définition générale de l'ordonnanceur: partie du SE chargée d'allouer les ressources aux processus.

1) Notions utiles: équilibrage de travaux

- On peut distinguer entre 2 types de processus, selon le type de ressource qu'ils utilisent le plus:
  - Les processus **tributaires de l'E/S:** utilisent peu l'UCT et beaucoup l 'E/S.
  - Les processus **tributaires de l 'UCT:** utilisent beaucoup l'UCT et peu d'E/S.

Quel équilibre l'ordonnanceur devrait-il réaliser?

- Le temps d'UCT non utilisé par les processus tributaires de l'E/S peut être utilisé par les processus tributaires de l'UCT et vice-versa.
- L'UCT doit rester le moins possible inactive, sans pour autant saturer la mémoire principale du système.

- Equilibrage et priorité:
  - Processus longs et non-urgents Vs Processus courts et urgents.
  - L'ordonnanceur pourra donner la priorité aux deuxièmes et exécuter les premiers , quand il y a du temps machine disponible.

II.2 Niveaux d'ordonnancement des processus

1) Notions utiles: Traitement par lots Vs. traitement interactifs

- Traitement par lots (batch):
  - Processus (ou Job) non-urgents qui sont soumis au système groupés et exécutés les uns après les autres (d'où le nom par lots), pour récupérer la réponse plus tard.
  - Il existe en général une relation entre les jobs successifs.
  - Exemple: Tri de fichier, calcul d'une fonction complexe, sauvegarde régulière de fichiers usagers, etc.

- Traitement Interactif:
  - Processus qui demandent une interaction continue avec l'ordinateur.
  - L'utilisateur reçoit le(s) résultat(s) immédiatement.
  - Exemple: édition de documents ou d'un programme.

II.2 Niveaux d'ordonnancement des processus

1) Notions utiles: les interruptions

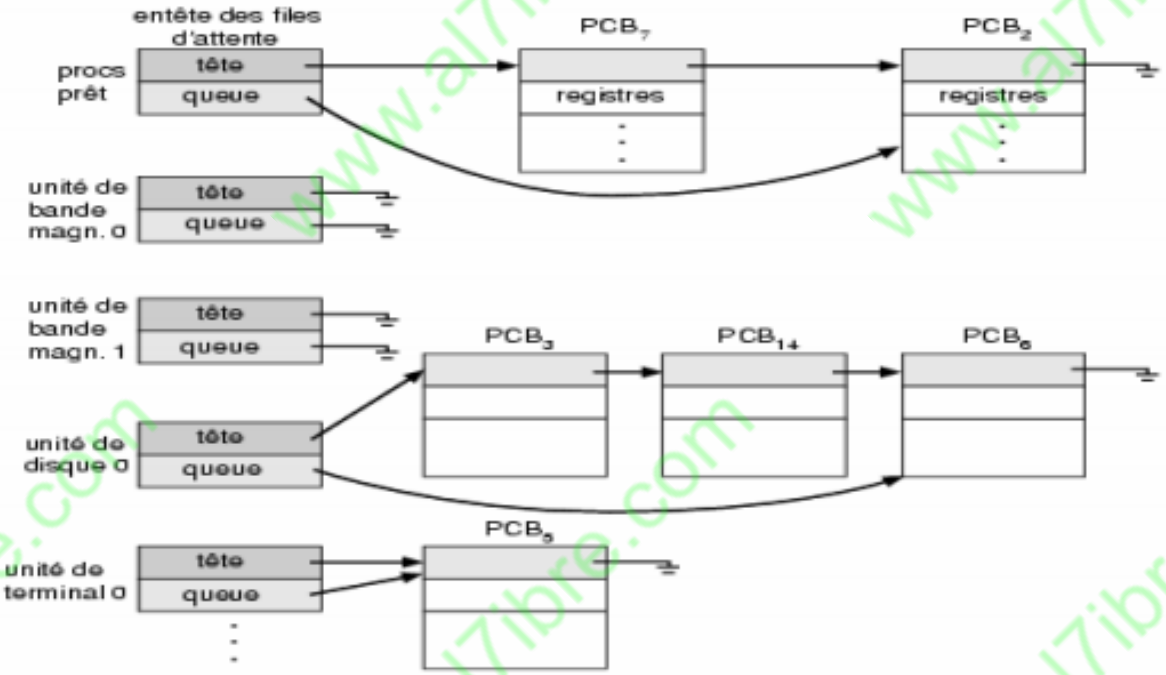
Une interruption est un signal pour arrêter un processus, qui peut avoir plusieurs causes:

- Interruptions causées par le programme utilisateur:
  - Exception: Division par 0, débordement, tentative d'exécuter une instruction protégée, référence au delà de l'espace mémoire du programme
  - Appels Système: demande d'entrée-sortie, demande d'autre service du SE, minuterie établie par le programme lui-même.
- Interruptions causées par le SE:
  - Le processus doit céder l'UCT à un autre processus (Préemption).
- Interruptions causées par les périphériques ou par le matériel:
  - Fin d'une E/S.

II.2 Niveaux d'ordonnancement des processus

1) Notions utiles: les files d'attente

- Les processus qui résident dans la mémoire principale et sont prêts et en attente d'exécution sont conservés sur une liste appelée **File des Processus Prêts** (ou **Ready Queue**).
- **Chaque ressource** a sa propre file de processus en attente.
- C'est généralement une liste chaînée, contenant un pointeur vers le PCB du processus, et un pointeur vers le PCB du processus suivant dans la file.
- En changeant d'état, les processus se déplacent d'une file à l'autre.



II.2 Niveaux d'ordonnancement des processus

2) Définition générale des 3 niveaux d'ordonnancement

- Un processus passe une bonne partie de sa durée de vie dans divers files d'attente.

- La sélection d'un processus à partir de ces files d'attente est effectuée par l'ordonnanceur.

- L'ordonnanceur opère sur 3 niveaux:
  - ***L'ordonnanceur à long terme*** *(ou ordonnanceur de travaux)*: décide du moment où les processus vont être chargés en mémoire.
  - ***L'ordonnanceur à moyen terme*** *(ou ordonnanceur de mémoire ou permutateur)*: décide de la suspension/reprise des processus lors d'un manque de mémoire.
  - ***L'ordonnanceur à court terme*** *(ou ordonnanceur de processeus ou répartiteur)*: décide quel processus aura le contrôle de l'UCT.

II.2 Niveaux d'ordonnancement des processus

3) Ordonnanceur de travaux (à long terme)

Dans un système par lots, il existe plus de processus soumis que ceux capables d'être exécutés immédiatement.

✓ **Ils sont alors chargés sur une file d'attente au niveau d'un périphérique de stockage de masse (le disque) pour être exécutés plus tard.**

- Le rôle de l'ordonnanceur de travaux est de:
  1. Sélectionner les processus à partir de cette file d'attente.
  2. Les charger dans une nouvelle file d'attente **des processus prêts** pour accéder à l'UCT.
  3. Déterminer le niveau de multiprogrammation: nombre de processus en mémoire pouvant être exécutés en parallèle par le SE.
     - Le nombre est choisi d'une manière à établir un équilibre entre les processus tributaires de l'UCT et ceux tributaires des E/S.
- Une fois que le processus est admis par l'ordonnanceur de travaux, il n'en sort que lorsqu'il est terminé ou s'il est détruit par le SE (suite à une erreur grave ou à la demande de l'utilisateur).
- N.B: La plupart des systèmes interactifs multiprogrammés ne disposent pas d'ordonnanceur de travaux. Chaque nouveau processus est mis en mémoire principale pour être pris en charge par l'ordonnanceur à court terme.

II.2 Niveaux d'ordonnancement des processus

4) Ordonnanceur de mémoire ou permutateur (à moyen terme)

- Les SE multiprogrammés introduisent un ordonnanceur à moyen terme.
- Le rôle du permutateur est d'effectuer:
  - **Swap out:** Supprimer un processus de la mémoire principale et le placer en mémoire secondaire. Il ne sera plus en concurrence avec les autres pour les ressources.
  - **Swap in:** Ré-introduire plus tard le processus en mémoire principale et son exécution pourra se poursuivre là où elle a été interrompue.
  - Réduire ainsi le niveau de multiprogrammation.

- Le swapping (permutation) est nécessaire pour:
  - améliorer l'équilibre entre processus tributaires de l'E/S et ceux tributaires de l'UCT
  - remédier au problème de dépassement de la mémoire principale disponible.

- Le swapping ne doit pas être trop fréquent pour ne pas gaspiller la bande passante des disques.

II.2 Niveaux d'ordonnancement des processus

5) Ordonnanceur de l'UCT ou répartiteur (à court terme)

Il est utilisé par tout type de SE et son rôle est principalement de:

- Choisir, parmi la file d'attente des processus prêts, à quel processus sera alloué l'UCT et pour quel laps de temps.

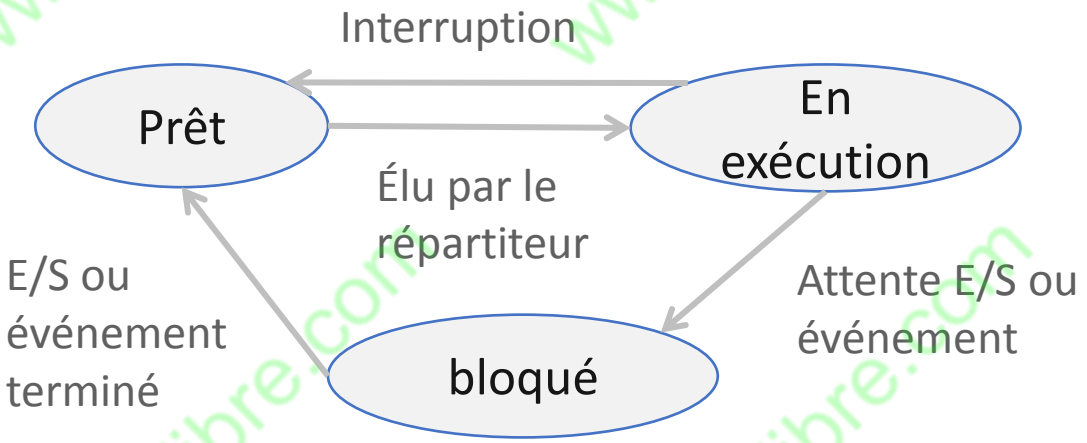
- Etre très rapide pour ne pas ralentir le SE puisqu'il est très fréquemment utilisé.

1. Introduction et Définitions
2. Niveaux d'ordonnancement des processus
3. Etats des processus
4. Algorithmes d'ordonnancement
5. Superviseur des processus
6. Création de processus

II.3 Etats des processus

1) Les états des processus dans un répartiteur (ordonnanceur de processus)

- Les processus sont concurrents et se partagent l'UCT, ils ne peuvent être continuellement actifs. Ils ont donc, si on ne considère pour commencer que le répartiteur**, 3 états et 4 transitions possibles**:
- Prêt (ready): état d'un processus qui n'est pas alloué à l'UCT, mais qui est prêt à être exécuté.
- En exécution (running) : état d'un processus exécuté sur une UCT.
- Bloqué (blocked): état d'attente d'un événement extérieur, tel qu'une E/S, nécessaire à la poursuite de l'exécution du processus.

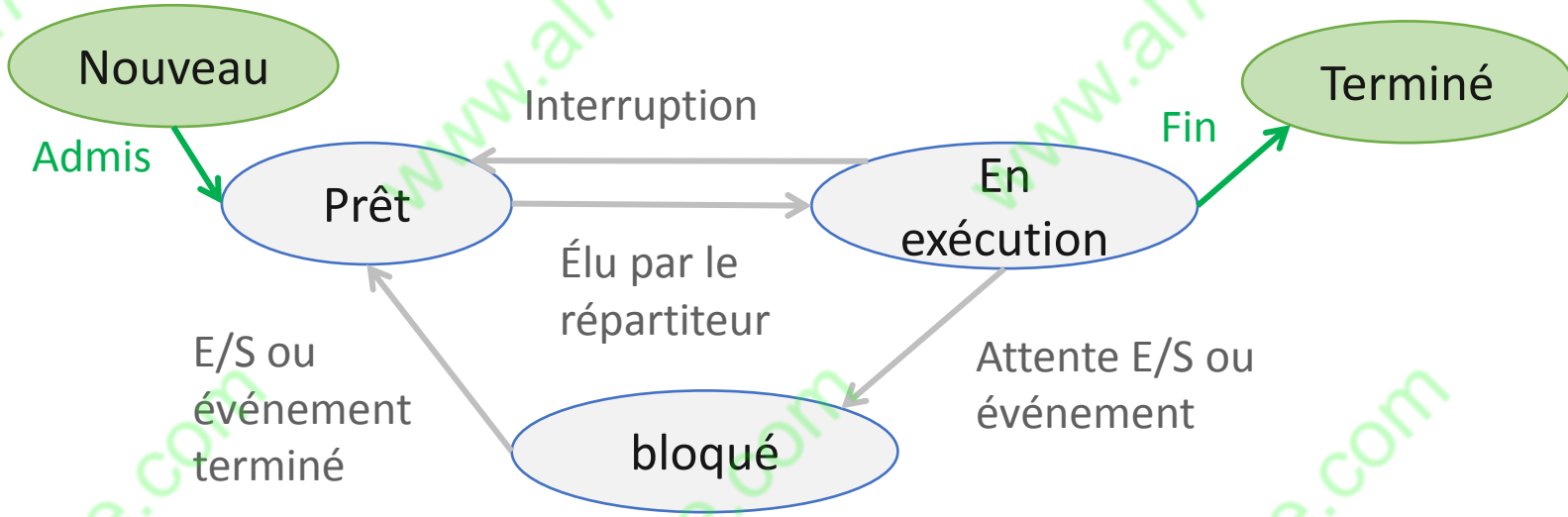


II.3 Etats des processus

2) Les états des processus dans un ordonnanceur de travaux

2 états sont ajoutés aux états précédents lorsqu'un ordonnanceur de travaux est utilisé:

- Nouveau (New): le processus vient d'être créé mais n'est pas encore admis par l'ordonnanceur de travaux pour concurrencer à l'accès à l'UCT.
- Terminé (Terminated): le processus a achevé sa tâche. Il sera détruit prochainement par le SE pour libérer l'espace. Il est parfois conservé pendant un temps à l'état terminé en attendant qu'une E/S s'achève ou que les données de ce processus soient exploitées par un autre processus. On parle alors de processus " zombie".



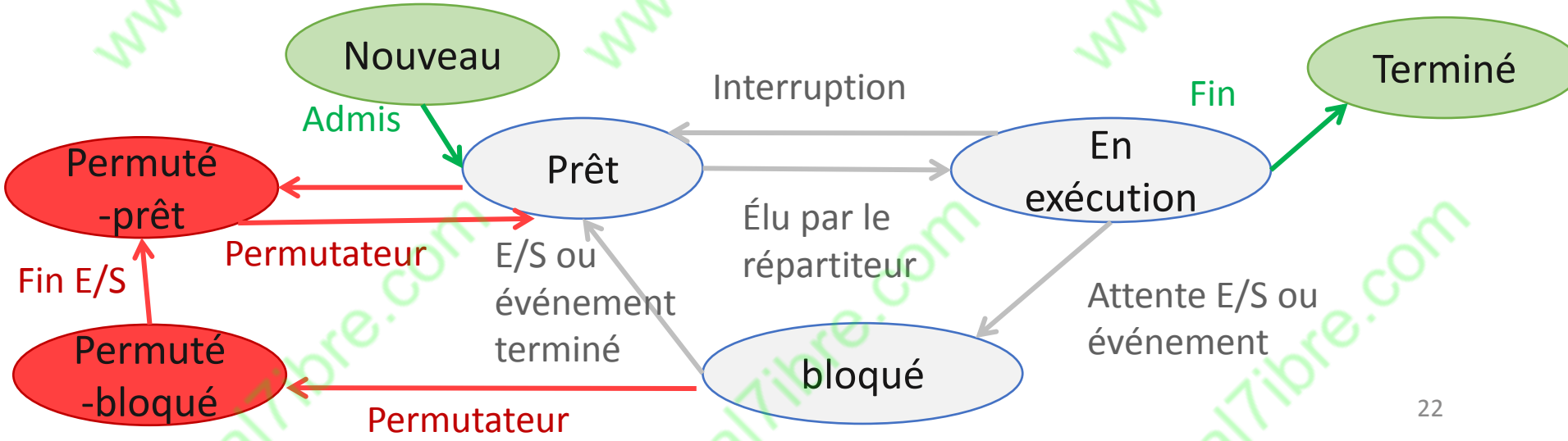
N.B: Quelque soit son état, un processus peut prendre fin suite à une action externe: le SE ou un autre processus peuvent mettre fin à un processus, en le passant en état terminé.

II.3 Etats des processus

3) Les états des processus dans un permutateur

2 états sont ajoutés aux états précédents lorsqu'un permutateur est utilisé:

- Permuté-prêt (Swapped-ready): le processus est pour l'instant transféré en mémoire secondaire. Le processus est réintroduit plus tard par le permutateur.
- Permuté-bloqué (Swapped-blocked): le processus était bloqué en attendant une E/S par exemple, puis a été transféré sur la mémoire secondaire pour faire de la place en mémoire principale. Lorsqu'il termine ses E/S, il passe à l'état permuté-prêt.



1. Introduction et Définitions
2. Niveaux d'ordonnancement des processus
3. Etats des processus
4. Algorithmes d'ordonnancement
5. Superviseur des processus

II.4 Algorithmes d'ordonnancement

1) Généralités

- Rôle: décider de l'allocation d'une ressource aux processus qui l'attendent: **algos d'ordonnancement pour l'UCT.**

- Objectif: aboutir à un partage efficace du temps d'utilisation de l'UCT:
  Mais que veut dire efficace?
  - L'algo doit identifier le processus qui conduira à la **«meilleure»** performance possible du système.
  - Il existe **différents critères** pour mesurer la performance et dont l'importance est relative à l'algo lui même.

II.4 Algorithmes d'ordonnancement

2) Les critères de performance

- **Utilisation UCT** à maximiser: pourcentage d'utilisation pendant une période d'observation donnée.
- **Débit** (ou rendement, Throughput) à maximiser: nombre de processus complétés pendant une période d'observation donnée.
- **Temps de rotation** (ou de service, turnaround time) à minimiser: Temps écoulé entre le moment où un processus devient prêt à s'exécuter et le moment où il finit de s'exécuter.
- **Temps d'attente** (waiting time) à minimiser : somme de tout le temps passé en file prêt.
- **Temps de réponse** (response time) à minimiser: utile pour les systèmes interactifs. Temps écoulé entre la soumission d'une requête et la première réponse obtenue.
- **Equité :** degré auquel tous les processus reçoivent une chance égale de s'exécuter. On essaie ainsi d'éviter **la famine**: c'est le cas où un processus n'obtient pas la ressource
- **Priorités :**attribue un traitement préférentiel aux processus dont le niveau de priorité est élevé.

Remarque: En général, on tente d'optimiser **les valeurs moyennes** pour tous les processus mis en jeu pendant une période d'observation donnée, pour les temps **d'attente**, de **rotation** et de **réponse**.

II.4 Algorithmes d'ordonnancement

3) Non préemptif Vs Préemptif

Il existe 2 types d'algo. Ordonnancement:

- **Non préemptif** (ou coopératif ou sans réquisition):
  - **Définition:** le processus sélectionné garde le contrôle de l'UCT jusqu'à ce qu'il se bloque ou qu'il termine.
  - ✓Avantages: facile à mettre en œuvre. Ne nécessite pas de mécanismes matériels spécifiques (horloges, …)
  - ↓Inconvénients: Correspond difficilement aux systèmes interactifs ou le temps de réponse est important.
- **Préemptif** (avec réquisition) :
  - **Définition:** l'algo retire l'UCT au processus en cours d'exécution pour l'attribuer à un autre processus. Ce type est indispensable pour les système interactifs.
  - ✓Avantages: Convient aux systèmes interactifs
  - ↓Inconvénients: Commutation fréquente de contexte des processus, ce qui peut diminuer le débit.

II.4 Algorithmes d'ordonnancement

4) First come first served (FCFS)

- Algo non préemptif très simple. Il est aussi appelé PAPS (Premier Arrivé Premier Servi).
- L'ordonnancement est fait dans l'ordre d'arrivée en gérant une file FIFO (First In First Out) unique des processus prêts, sans priorité ni réquisition : le processus élu est celui qui est en tête de liste.
- Chaque processus s'exécute jusqu'à son terme.

✓ **Avantages:**

- Simple
- Pas de famine

↓ **Inconvénients:**

- Temps d'attente moyen très important.
- Non adapté aux systèmes interactifs.

II.4 Algorithmes d'ordonnancement

4) First come first served (FCFS) – Exemple

- Soient les processus P1 , P2 , P3 qui arrivent à l'instant 0 dans cet ordre.

Processus	Cycle UCT
P1	24
P2	3
P3	3

- Le diagramme de Gant suivant correspondant à l'algo FCFS:

- **Les mesures de performances sont:**
  - Utilisation UCT= 30/30 (100%)
  - Temps d'attente moyen= (0+24+27)/3=17
  - Temps de rotation moyen=(24+27+30)/3=27
  - Débit= 3/30=0.1

Processus	Temps attente	Temps de rotation
P1	0	24
P2	24	27
P3	27	30

II.4 Algorithmes d'ordonnancement

4) First come first served (FCFS) – Exemple 2

- Soient les mêmes processus P1 , P2 , P3 qui arrivent à l'instant 0 dans l'ordre P2, P3, P1

Processus	Cycle UCT
P1	24
P2	3
P3	3

- Le diagramme de Gant correspondant à l'algo FCFS:

- **Les mesures de performances sont:**
  - Temps d'attente moyen= (6+0+3)/3=3
  - Temps de rotation moyen=(30+3+6)/3=13
  - Débit= 3/30=0.1

Processus	Temps attente	Temps de rotation
P1	6	30
P2	0	3
P3	3	6

- **Remarque:** Lorsque les processus les plus courts sont arrivés en premier (donc élu en premier), les performances sont nettement meilleurs. On pourrait donc penser à utiliser un algo qui avantage les processus courts => **algo SJF**

II.4 Algorithmes d'ordonnancement

5) Shortest Job First (SJF)

- Algo non préemptif. Il est aussi appelé Plus court temps d'exécution (PCTE)
- Le processus qui a le cycle UCT le plus court est exécuté en premier.
- Le FCFS est utilisé en cas d'égalité.

✓**Avantages:**

- Le meilleur pour le temps d'attente moyen (lorsque tous les processus arrivent en même temps.)

↓**Inconvénients:**

- Risque de famine: les processus longs peuvent ne jamais s'exécuter
- Nécessite de connaitre à l'avance le temps du cycle UCT (adapté aux traitements par lots où une estimation de la durée du cycle est donnée). Sinon, il devrait être prédit

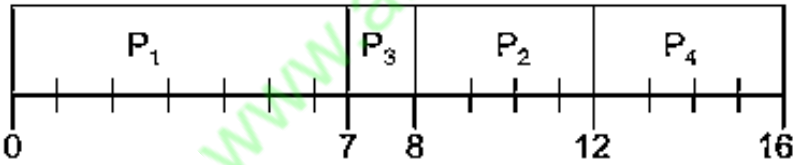
II.4 Algorithmes d'ordonnancement

5) Shortest Job First (SJF) – Exemple

- Soit les processus P1 , P2 , P3, P4 qui arrivent selon différents temps d'arrivée.

Processus	Arrivée	Cycle UCT
P1	0	7
P2	2	4
P3	4	1
P4	5	4

- Le diagramme de Gant correspondant à l'algo SJF



- **Les mesures de performances sont:**
  - Temps d'attente moyen= (0+6+3+7)/4=4
  - Temps de rotation moyen=(7+10+4+11)/4=8
  - Débit= 4/16=0.25

Processus	Temps attente	Temps de rotation
P1	0	7
P2	8-2=6	12-2=10
P3	7-4=3	8-4=4
P4	12-5=7	16-5=11

II.4 Algorithmes d'ordonnancement

6) Shortest Remaining Time First(SRTF)

- C'est la version préemptive du SJF. Il est aussi appelé Plus court temps d'exécution avec Réquisition (PCTER)
- Chaque fois qu'un nouveau processus est introduit dans la file des processus prêts, l'ordonnanceur compare sa durée du cycle UCT à la durée restante du processus en cours d'exécution. Si la durée du nouveau processus est inférieure, le processus en cours d'exécution est réquisitionné.

✓ **Avantages:**

- Plus efficace que SJF, car le temps d'attente moyen optimal est garantit quelque soit le moment d'arrivée des processus

↓ **Inconvénients:**

- Risque de famine
- Besoin de connaitre la durée du cycle UCT à l'avance

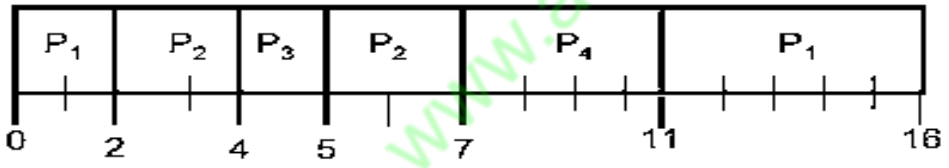
II.4 Algorithmes d'ordonnancement

6) Shortest Remaining Time First (SRTF)

Soit les processus P1 , P2 , P3, P4 qui arrivent selon différents temps d'arrivée.

Processus	Arrivée	Cycle UCT
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Le diagramme de Gant correspondant à l'algo SRTF:



Les mesures de performances sont:

- Temps d'attente moyen= (9+1+0+2)/4=3
- Temps de rotation moyen=(16+5+1+6)/4=7
- Débit= 4/16=0.25

Processus	Temps attente	Temps de rotation
P1	9	16
P2	1	7-2=5
P3	0	5-4=1
P4	2	11-5=6

II.4 Algorithmes d'ordonnancement

7) Round-Robin (RR)=Tourniquet

- Algo préemptif le plus utilisé en pratique.
- A chaque processus est allouée une tranche de temps, appelée quantum (généralement entre 10 et 100 ms.), pour s'exécuter.
- S'il s'exécute pour un quantum entier (sans autres interruptions), il est interrompu par la minuterie et l'UCT est donnée à un autre processus. Le processus interrompu redevient prêt (en fin de file d'attente).

✓Avantages:

- Pas de monopolisation de l'UCT, équitable
- Pas de famine
- Bon temps de réponse

↓Inconvénients:

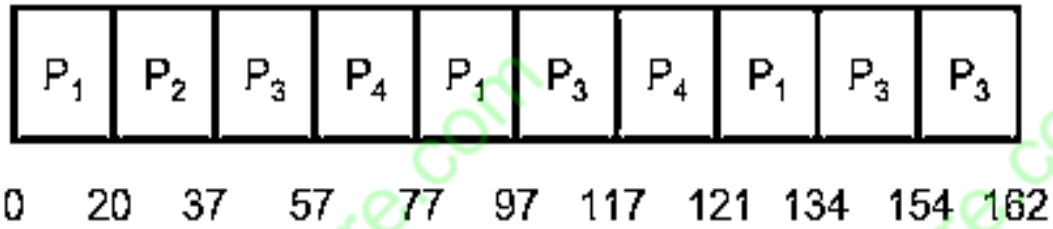
- Temps d'attente moyen en général important.
- Influence de la valeur du quantum, difficile à déterminer (grand : FIFO, petit: perte de temps dans les changements de contexte)

II.4 Algorithmes d'ordonnancement

7) Round-Robin (RR)=Tourniquet - Exemple

Soit les processus P1 , P2 , P3, P4. Le diagramme de Gant correspondant à l'algo RR avec quantum = 20

Processus	Cycle UCT
P1	53
P2	17
P3	68
P4	24



Processus	Temps de rotation
P1	134
P2	37
P3	162
P4	121

Les mesures de performances sont:

- Utilisation UCT= 162/162=100%
- Temps de rotation moyen=(134+37+162+121)/4=113.5
- Débit= 4/162=0.025
- Temps de rotation et temps d'attente moyens sont beaucoup plus élevés que les algos précédents, mais meilleur temps de réponse moyen.
- Le RR suppose que tous les processus sont aussi importants, mais en pratique, ce n'est pas le cas (processus vidéo plus important que processus qui affiche l'heure) => Algo HPF

II.4 Algorithmes d'ordonnancement

8) Comparaison

Soit les processus P1 , P2 , P3, P4.

Processus	Arrivée	Cycle UCT
P1	0	7
P2	1	4
P3	2	7
P4	3	5

Remarques:

- SRTF fournit le meilleur temps d'attente moyen.
- RR avec petit quantum augmente le temps d'attente moyen.

Temps d'attente:

Processus	FCFS	SJF	SRTF	RR (2)	RR(5)	RR(10)
P1	0	0	9	14	14	0
P2	6	6	0	7	4	6
P3	9	14	14	14	14	9
P4	15	8	2	14	11	15
Moyenne	**7.5**	**7**	**6.25**	**12.25**	**10.75**	**7.5**

II.4 Algorithmes d'ordonnancement

8) HPF (Highest Priority First ou haute priorité d'abord)

- Affectation d'une priorité à chaque processus (souvent nombre entier, avec 0 la plus haute).L'UCT est donnée au processus prêt avec la plus haute priorité.
- Peut être préemptif ou non. Quand un nouveau processus arrive:
  - **Cas préemptif:** comparer sa priorité à celle du processus en cours d'exécution. L'UCT est alors réquisitionnée en cas de plus haute priorité. Le processus sorti sera remis en tête de la liste d'attente prêt correspondant à sa priorité.
  - **Cas non préemptif:** placer le processus dans la file d'attente FIFO correspondant à sa priorité. Une fois qu'il sera élu, il ne sera pas interrompu par l'ordonnanceur.
- **N.B:** Il y a une file d'attente prêt pour chaque niveau de priorité. HPF choisit toujours dans la file la plus prioritaire.

✓**Avantages:**

- Simple, Prise en compte de l'importance des processus

↓**Inconvénients:**

- Risque de famine pour les processus moins prioritaire.
- **Solution:** Augmenter la priorité des processus qui attendent depuis longtemps.

II.4 Algorithmes d'ordonnancement

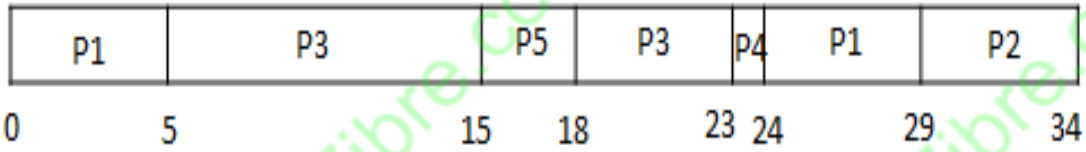
8) HPF (Highest Priority First ou haute priorité d'abord) - Exemple

Soit les processus P1 , P2 , P3, P4, P5. Le diagramme de Gant correspondant à l'algo HPF:

Processus	Arrivée	Cycle UCT	Priorité
P1	0	10	3
P2	2	5	7
P3	5	15	2
P4	10	1	2
P5	15	3	1

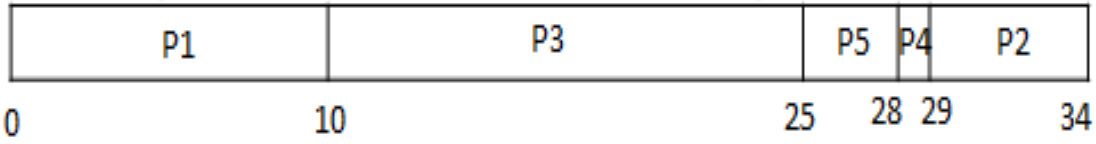
- **préemptif:** Temps de rotation moyen=

(29+32+18+14+3)/5 =19.2



- **Non préemptif:** Temps de rotation moyen=

(10+32+20+19+13)/5 =18.8



Processus	Temps de rotation préemptif	Temps rotation non préemptif
P1	29	10
P2	34-2=32	34-2=32
P3	23-5=18	25-5=20
P4	24-10=14	29-10=19
P5	18-15=3	28-15=13

- Avec HPF non préemptif, les processus prioritaire risquent d'attendre plus longtemps.

II.4 Algorithmes d'ordonnancement

9) Files multiples (à plusieurs niveaux)

- La file prêt est séparée en plusieurs files. Par exemple une pour processus d'arrière-plan (background - batch) et une autre pour processus de premier plan (foreground - interactive).
- Chaque file a son propre algo d 'ordonnancement. Par exemple: FCFS pour arrière-plan, RR pour premier plan.
- Comment ordonnancer entre les files?
  - Priorité fixe à chaque file
  - Ou bien chaque file reçoit un certain pourcentage de temps UCT, (exemple: 20% pour arrière-plan, 80% pour premier plan)

✓**Avantages:**

- Permet une catégorisation des tâches accomplies par le système.

↓**Inconvénients:**

- Risque de famine (cas priorité fixe)
- Priorités statiques (dépendent de la nature du processus)

II.4 Algorithmes d'ordonnancement

9) Files multiples avec feedback

- Un processus peut passer d'une file à une autre, par exemple quand il a passé trop de temps dans une file, il passe à une autre plus prioritaire.
- Ce type d'algo d'ordonnancement est défini par :
  - nombre de files, ordonnanceur pour chaque file.
  - règles pour changer un processus de file (vers le haut ou vers le bas).
  - règles pour décider de la file initiale d'un processus.

✓**Avantages:**

- Le plus général.
- Flexibilité (nb files, algo pour chaque file, quantum, certains types de tâche peuvent commencer dans une file peu prioritaire).

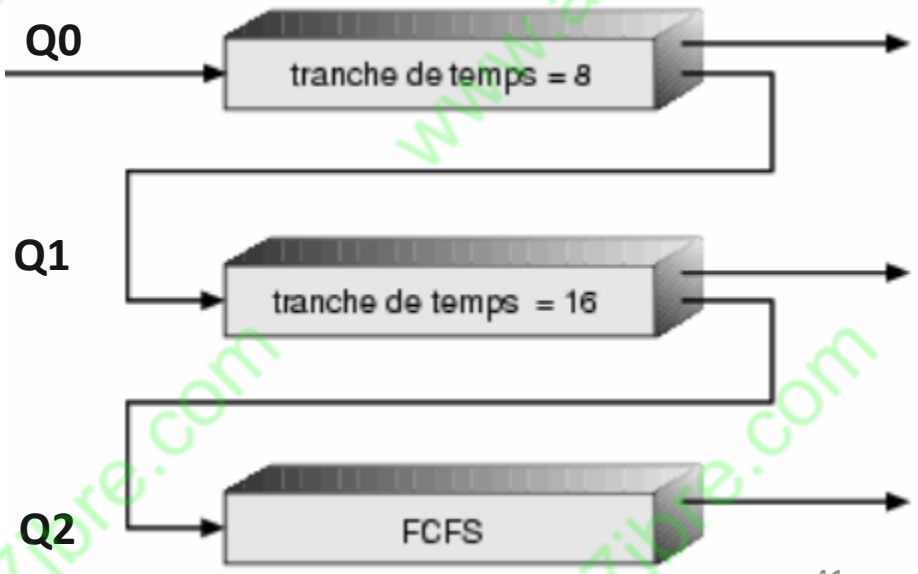
↓**Inconvénients:**

- Le plus complexe à mettre en place. Ajustement délicat des paramètres.
- Risque de famine.
- Nombreux changements de contexte.

II.4 Algorithmes d'ordonnancement

9) Files multiples avec feedback - exemple

- Trois files: **Q0:** RR q= 8 ms | **Q1**: RR q= 16 ms |**Q2**: FCFS
- Exemple de Schéma d'ordonnancement:
  - Un nouveau processus entre dans Q0, il reçoit 8 ms d'UCT
  - S'il ne finit pas dans les 8 ms, il est mis dans Q1, il reçoit 16 ms additionnels
  - S'il ne finit pas encore, il est interrompu et mis dans Q2

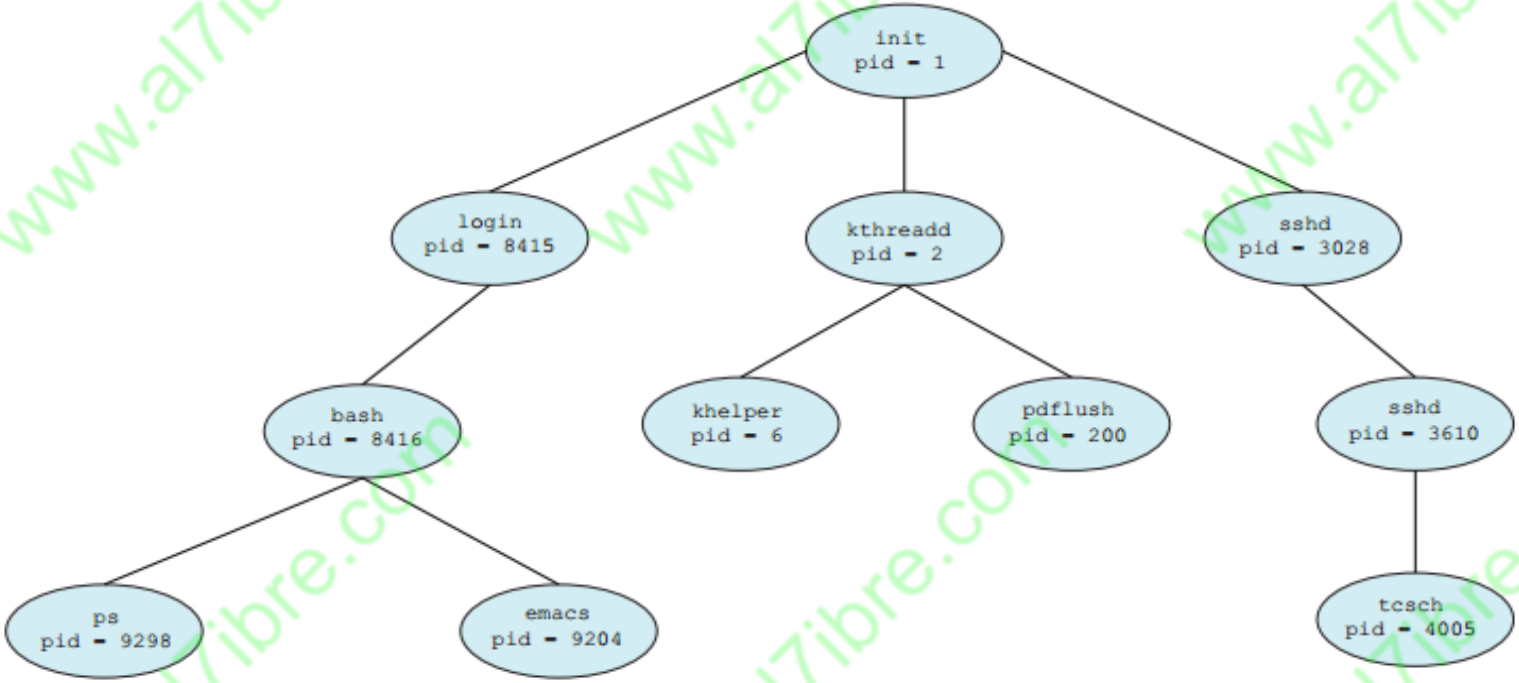


1. Introduction et Définitions
2. Niveaux d'ordonnancement des processus
3. Etats des processus
4. Algorithmes d'ordonnancement
5. Superviseur des processus
6. Création de processus

II.5 Superviseur des processus

1) Création de processus

- Pendant son exécution, un processus (**père**) peut créer de nouveaux processus (**fils**), qui peuvent à leur tour créer des fils, formant ainsi une ***arborescence*** de processus.
- La majorité des SE identifient les processus par un **numéro unique** (process id ou **PID**) et font référence au père par son PID noté **PPID**.
- Pour le SE Linux par exemple, lorsque le SE démarre, il crée un **processus initial** nommé **init** (avec **pid=1**). Si un utilisateur souhaite se connecter, **init** crée le processus login(pid 8415).



II.5 Superviseur des processus

1) Création de processus

La majorité des SE proposent des mécanismes de création de processus fils à partir du processus père:

- **Fork():**
  - Le SE crée un **nouveau PCB** pour le fils et y copie **les mêmes éléments** du PCB du père.
  - Père et fils **continuent** leur exécution à partir de l'instruction suivant le fork(), puisque les deux processus ont **les mêmes valeurs des registres** dans leur PCB.
  - La **valeur de retour de fork()** est **0 pour le fils**, alors que pour **le père**, elle correspond au **PID du fils**. Avec un code qui teste cette valeur, le père et le fils peuvent être dirigés vers différents segments du code.

- **Exec():**
  - Typiquement, après l'appel système fork(), le fils utilise l'appel système exec() pour **remplacer les éléments de son PCB** par un nouveau programme.
  - De cette manière, les deux processus peuvent communiquer facilement et effectuer chacun leur exécution.
  - Le père peut se placer en attente jusqu'à la terminaison de son fils

II.5 Superviseur des processus

2) Terminaison de processus

3 appels systèmes sont liés à la terminaison des processus:

- **wait()** : Permet à un processus **père d'attendre jusqu'à ce que son fils** termine. Il retourne l'identifiant du processus fils et son état de terminaison.
- **exit() :** Permet au processus de **finir volontairement** son exécution car il a terminé ses instructions et retourne son état de terminaison.
- **kill():** Permet de **forcer l'arrêt** d'un autre processus. Habituellement, un tel appel ne peut être invoqué que par le père du processus qui doit être terminé.
    - Un père peut mettre fin à l'exécution de l'un de ses enfants pour diverses raisons:
        - Le fils a dépassé son utilisation de certaines des ressources allouées.
        - La tâche assignée au fils n'est plus nécessaire.
        - Le parent sort et le SE ne permet pas au fils de continuer si son parent se termine.

Remarque: Sur certains SE, lorsqu'un père prend fin, tous ses fils se terminent automatiquement. Puis, les fils de ces derniers s'achèvent à leur tour, ce mécanisme est connu sous le nom de **terminaison en cascade**.

II.5 Superviseur des processus

2) Terminaison de processus – le processus zombie

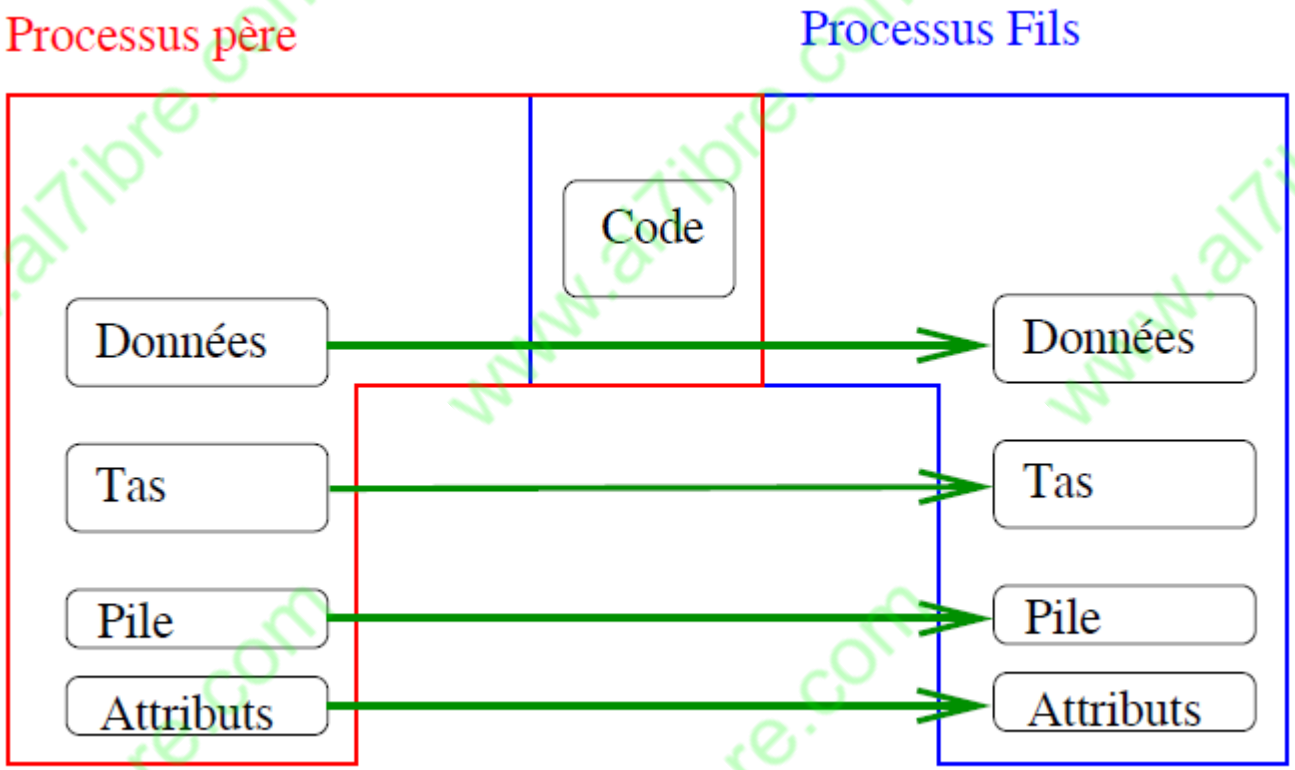
- Lorsqu'un processus se termine, ses ressources sont désaffectées par le SE. Cependant, **son PCB doit rester dans la table des processus** jusqu'à ce que son père appelle **wait().**
- Un processus qui s'est terminé, mais dont le père n'a pas encore appelé **wait(),** est connu sous le nom de **processus zombie**.
- Tous les processus passent à cet état lorsqu'ils terminent, mais généralement ils n'y restent que brièvement. Une fois que le père appelle **wait(),** le PID du zombie et son PCB sont libérés.
- Si un père n'a pas appelé **wait ()** et a terminé, ses processus fils vont devenir **orphelins.**
- Linux et UNIX attribuent le processus d'initialisation (init) en tant que nouveau père des processus orphelins. Init invoque périodiquement **wait()**, ce qui permet de **collecter** le statut de sortie de tout processus orphelin et de libérer son PID et son PCB.

1. Introduction et Définitions
2. Niveaux d'ordonnancement des processus
3. Etats des processus
4. Algorithmes d'ordonnancement
5. Superviseur des processus
6. Création de processus

II.6 Création de processus

1) La primitive système fork()

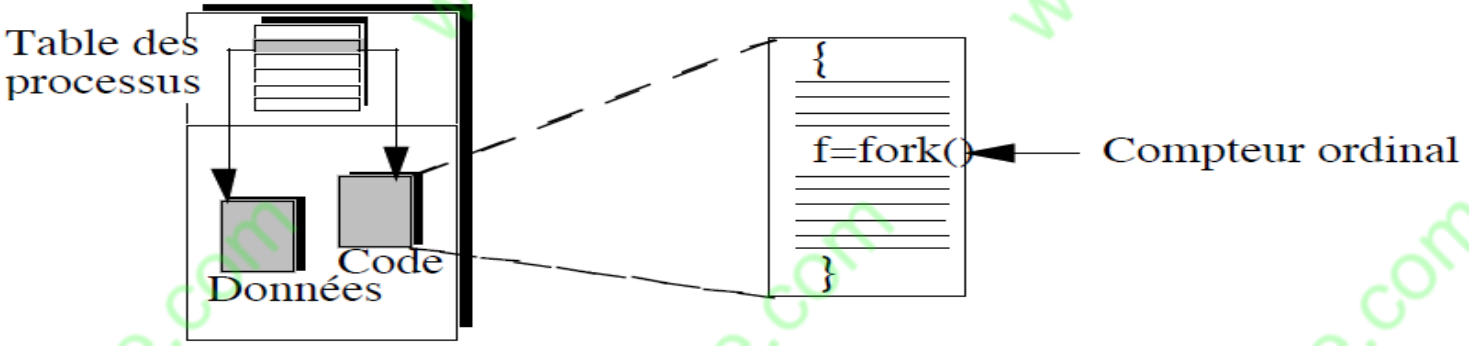
- Recopie les données et les attributs du processus père vers son processus fils auquel il attribue un nouveau pid.
- Le fils continue son exécution à partir de cette primitive



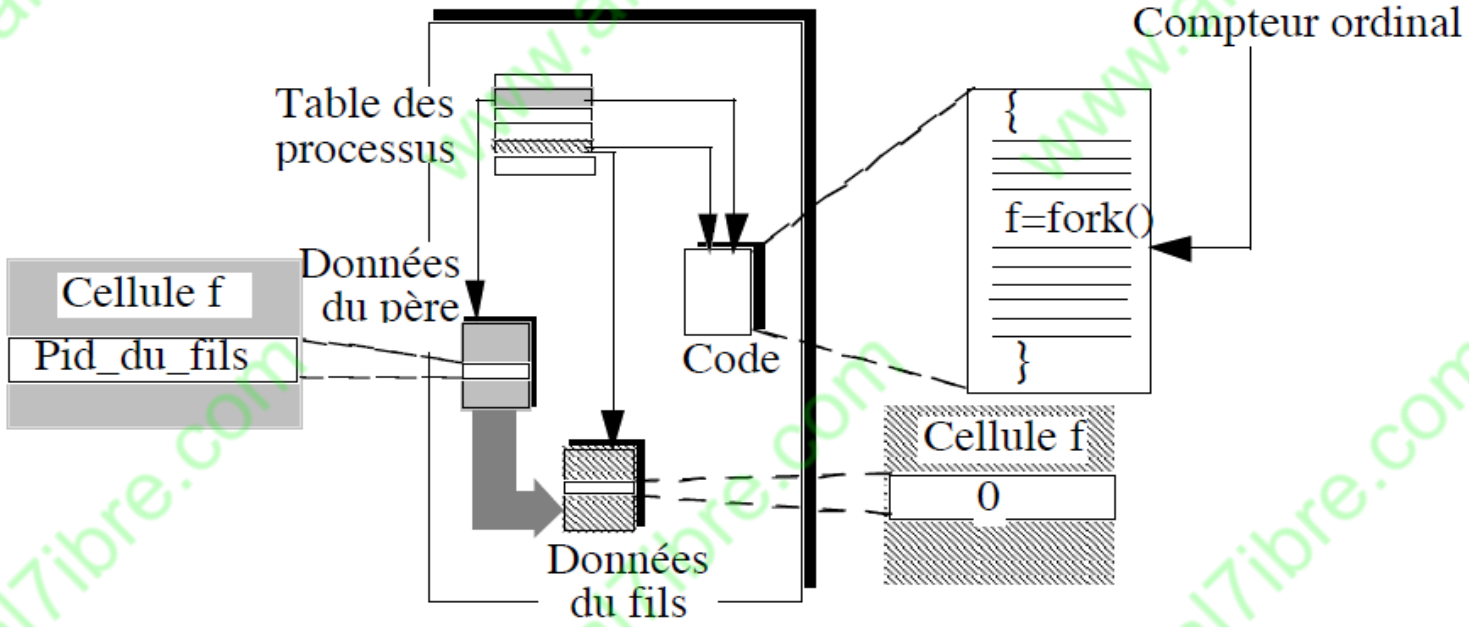
II.6 Création de processus

1) La primitive système fork() – avant vs après

Avant fork()



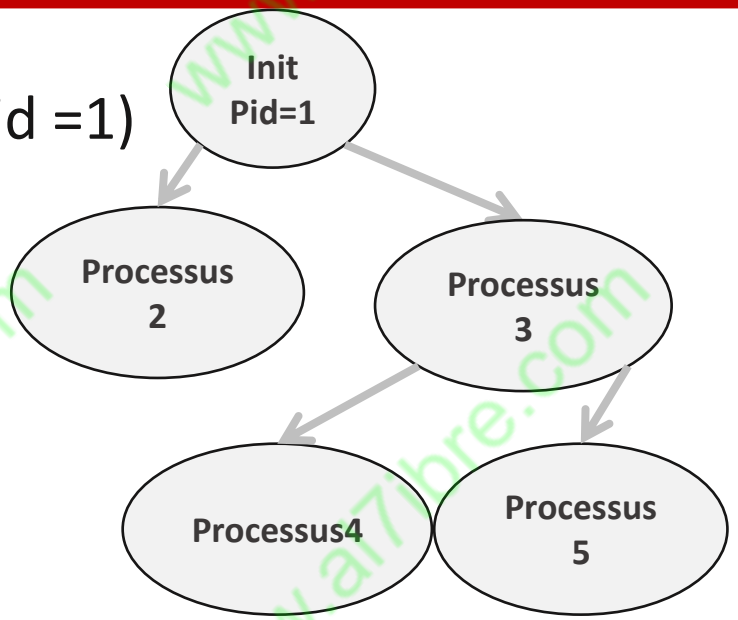
Après fork()



II.6 Création de processus

1) La primitive système fork() – Arborescence

- L'itération de fork() conduit à une arborescence à partir du processus init (pid =1)



- Différencier le père du fils : Code de retour du fork()
  - Dans le père : le fork retourne le pid du processus fils
  - Dans le fils : le fork retourne 0

- Pourquoi ?
  - Le fils peut connaitre le pid de son père avec getppid().
  - Alors que pour le père, le seul moyen pour connaitre le pid du processus fils est le retour du fork.

II.6 Création de processus

1) La primitive système fork() – porté du code

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main () {
int f;
f = fork();
printf (" Valeur retournee par la fonction fork: %d\n", (int)f);
printf (" Je suis le processus numero %d\n", (int)getpid());
return 0;
}
```

```
Valeur retournee par la fonction fork: 3952
Je suis le processus numero 3951
Valeur retournee par la fonction fork: 0
Je suis le processus numero 3952
```

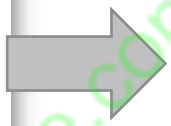
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
int k ;
printf("Je suis seul au monde\n");
k=fork();
if (k==-1) { printf("Erreur fork()");
}

if (k== 0) {
printf("Je suis le processus fils\n");
}
else
printf("Je suis le processus pere\n");
printf("Et la qui suis-je %d\n",getpid());
}
```

```
Je suis seul au monde
Je suis le processus pere
Et la qui suis-je 2850
Je suis le processus fils
Et la qui suis-je 2851
```

II.6 Création de processus

1) La primitive système fork() - porté des variables

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
int i,j,k ;
i=5; j=2;
if ((k=fork()) == -1) {
printf("Erreur fork()");
}
if (k== 0) {
/* code du fils */
printf("fils %d \n",getpid());
j--;
}
else
{
printf("pere %d\n",getpid());
i++;
}
printf("Pid:%d i:%d j:%d\n", getpid(),i,j);
}
```



```
pere 2906
Pid:2906 i:6 j:2
fils 2907
Pid:2907 i:5 j:1
```

- Les passage des variables se fait par valeur!!
- Le changement d'une variable par l'un des processus n'affecte pas la valeur de la variable dans l'autre processus

II.6 Création de processus

1) La primitive système fork() - relation père/fils

- Le père est toujours prévenu de la fin d'un fils
- Le fils est toujours prévenu de la fin du père
- Mais il faut que le père soit en attente
- Si la fin d'un fils n'est pas traitée par le père ce processus devient un processus **zombie**.

II.6 Création de processus

2) Les appels système wait et exit

Void exit (int)

- La Valeur du *int* est transmise au père, c'est le code de retour
- Fin du processus fils après exit
- Par convention (défaut) une fin correcte donne un code de retour nul.

int wait (int *)

- Entier retourné : pid du fils qui s'est terminé depuis l'invocation du wait
- Si aucun fils susceptible de se terminer alors renvoi de −1
- L'entier pointé enregistre l'état du fils lorsqu'il se finit (valeur en paramètre dans exit)

II.6 Création de processus

2) Les appels système wait et exit

```c
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <stdio.h>
int main() { int r,s,w;
if ( (r=fork())==0 ) {
printf("Fils %d\n",getpid());
// Traitement long
exit(14);
} else {
// P`ere doit attendre la
// mort de son fils.
w=wait(&s);
printf("w:%d s:%d\n",w,s);}
}
```

```
Fils 3156
w:3156 s:3584
```

La valeur de s=la valeur du paramètre de wait * 256

II.6 Création de processus

3) Mise en sommeil d'un processus

- Suspend l'exécution du processus appelant pour une durée de n secondes : **Int sleep(int n)**

```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <stdio.h>
main( )
{int PID, status ;
 if (fork( ) == 0)
{
printf("processus fils %d \n ", getpid());
exit(10);
}
PID=wait(&status);
printf("processus père %d\n ", getpid());
printf("sortie du wait \n") ;
sleep(15);
printf("PID = %d status= %d \n",PID, status);
exit(0);
}
```

```
processus fils 3559
 processus père 3558
sortie du wait
PID = 3559 status= 2560
```

Attente de 15s avant l'affichage de la dernière line

II.6 Création de processus

4) L'appel système exec et execl

- Un processus peut changer de code par un appel système à **exec** ou **execl**.
  - Code et données remplacés
  - Pointeur d'instruction réinitialisé

SMI - S4

Chapitre 3: Communication interprocessus & Synchronisation

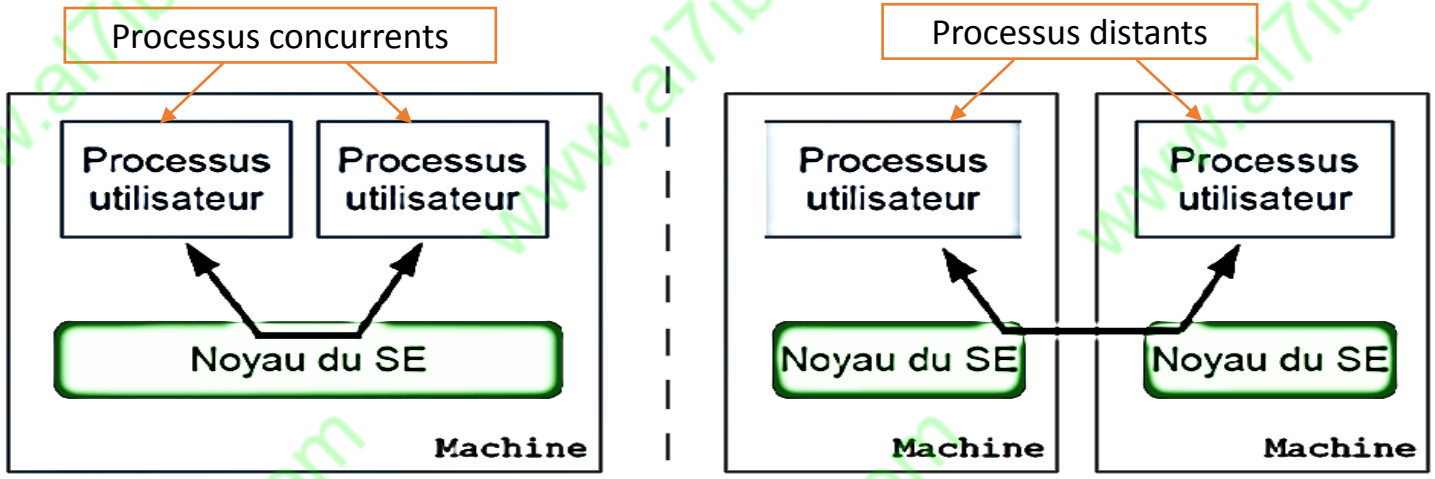
Cours donné par:
Pr. N. ALIOUA

Année universitaire:
2019-2020

1. Notions de base
2. Mécanismes de communication
3. Mécanismes de synchronisation

1. Notions

- Les processus, **concurrents** ou **distants**, sont amenés à **communiquer** et à **synchroniser** durant leur cycle de vie.
- Processus concurrents: sont en compétition pour le partage de **ressources**.
  - Coopérants: qui **partagent des données**, se trouvant en mémoire principale ou en mémoire secondaire, avec d'autres processus, et peuvent **s'affecter mutuellement** en cours d'exécution .
  - Indépendants: **ne partagent pas de données** avec d'autres processus et sont **ordonnancés indépendamment** les uns des autres.



Communications intra-système

Communications inter-système

1. Notions: Ressource

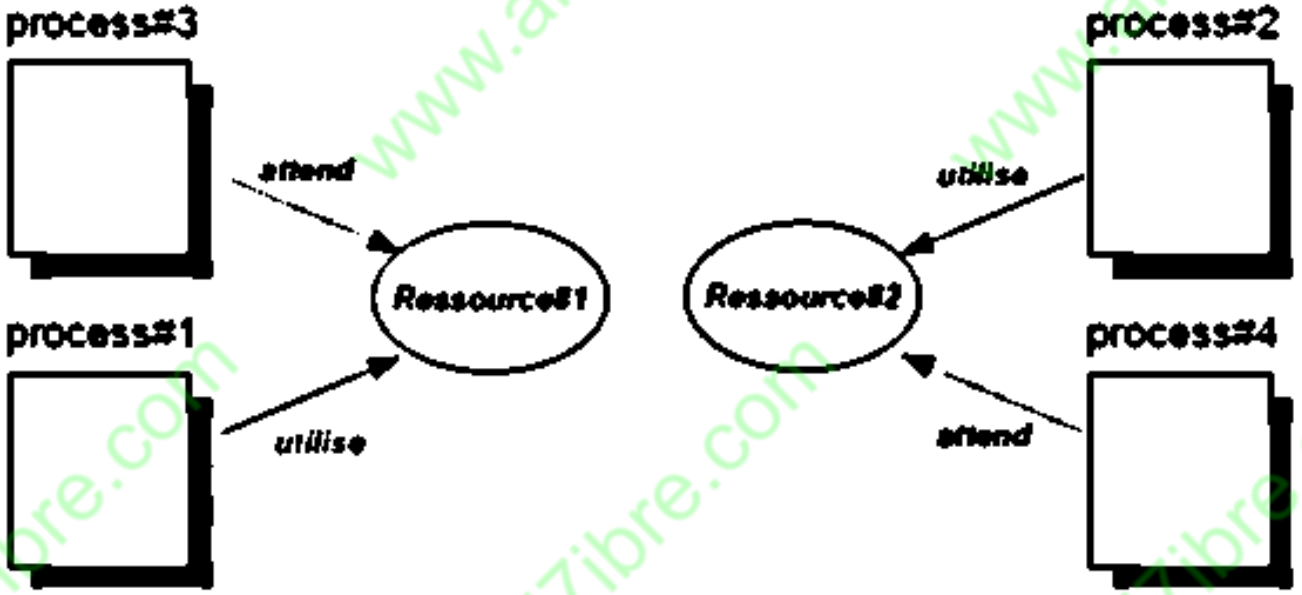
- Ressource: toute entité dont a besoin un processus pour qu'il puisse évoluer
  - Matérielle: mémoire, UCT, périphériques
  - Logicielle: données, variables
- Ressource locale à un processus:
  - Ne peut être utilisée que par ce processus
  - Doit obligatoirement disparaitre à la destruction de ce processus puisqu'elle n'est plus utilisable.
- Ressource commune: n'est locale à aucun processus.
- Une ressource commune partageable avec n points d'accès (n >= 1): une ressource qui peut être attribuée, au même instant, à n processus au plus.
- Une ressource critique: partageable à un point d'accès (n=1 ou non partageable).
  - Exemple: L'UCT est une ressource à un seul point d'accès.
- Le mode d'accès à une ressource peut évoluer dynamiquement:
  - Un fichier est une ressource à **n points d'accès** quand il est **ouvert en lecture**, **critique** quand il est **ouvert en écriture**.
- Section Critique: Soit une ressource critique ***c***, **la section critique** d'un processus ***p***, pour la ressource ***c***, est une phase du processus ***p*** pendant laquelle il utilise ***c***, qui devient donc inaccessible aux autres processus.

1. Notions: Ressource

Les processus coopérants sont confrontés à deux grands problèmes : la famine et l'inter-blocage(deadlock).

- Famine: Monopolisation d'une ressource. Si un processus émet un flux constant de requêtes (de lecture par exemple) et si toutes ses requêtes sont satisfaites en premier, il pourrait arriver que les requêtes d'autres processus ne soient jamais satisfaites.

Attente indéfinie



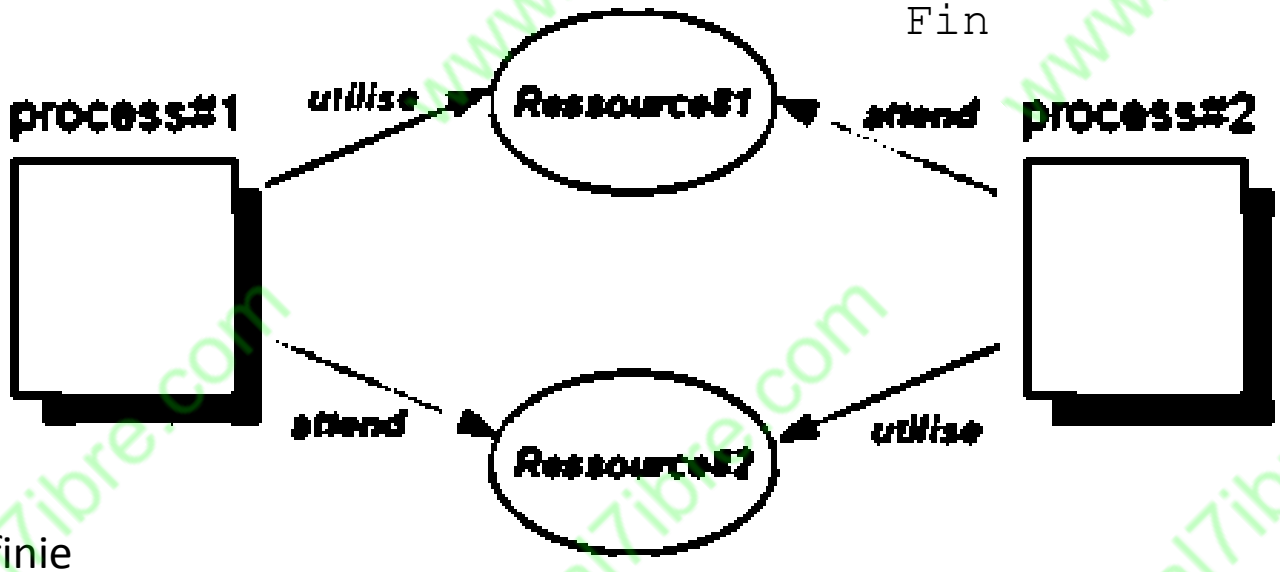
1. Notions: Ressource

- Inter-blocage: survient lorsque deux ou plusieurs processus demandent à obtenir des ressources en même temps, et que les ressources requises par les uns sont occupées par les autres et vice versa.

On considère deux processus P_1 et P_2 utilisant deux ressources critiques R_1 et R_2 comme suit :

```
Processus P1
Début
acquérir R1
acquérir R2
utiliser R1 et R2
Fin
```

```
Processus P2
Début
acquérir R2
acquérir R1
utiliser R2 et R1
Fin
```

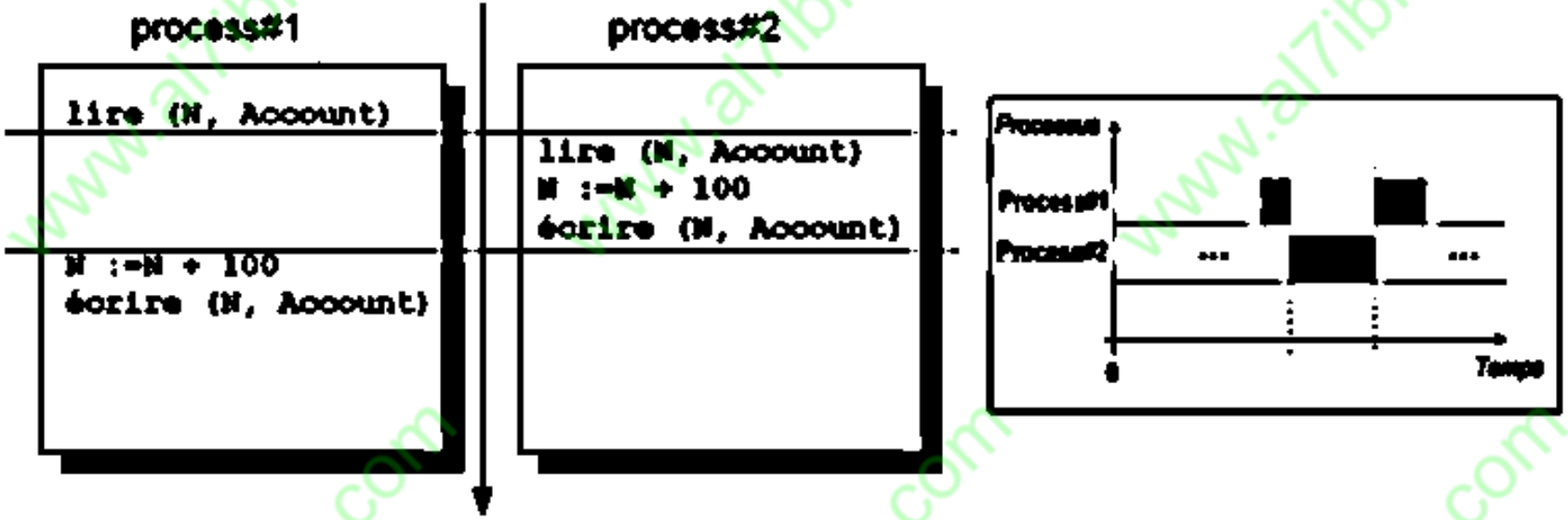


Attente infinie

1. Notions: Ressource

- Incohérence de données : problème de la synchronisation relative à l'exécution des processus

- **Exemple:** incrémenter un compte client de 100 depuis deux opérations bancaires simultanées.
- **N:** solde initial, **Account:** numéro du compte client.



Une solution?

1. Notions: Exclusion mutuelle

- Soit deux processus ***p*** et ***q*** qui produisent des données devant être imprimées sur une imprimante unique. L'emploi de cette imprimante par ***p exclut son emploi par q tant que l'impression pour p n'est pas terminée***.
- **Un mécanisme d'exclusion mutuelle** sert à assurer l'atomicité des sections critiques relatives à une ressource critique.
- Autrement: s'assurer que les ressources non partageables ne soient attribuées qu'à un seul processus à la fois.
- Un processus désirant entrer dans une section critique doit être **mis en attente** si la ressource relative à la section critique n'est pas libre.
- Mais Comment peut-on attendre?
  - **Active :** procédure *entrer_Section_Critique* matérialisée *par* 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.

1. Notions: Exclusion mutuelle

- Un mécanisme d'exclusion mutuelle doit satisfaire les conditions suivantes:

  1. **Exclusion mutuelle:** Si le processus Pi s'exécute dans sa section critique, alors aucun autre processus peut s'exécuter dans sa section critique.

  2. **Interblocage:** aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres processus d'entrer en section critique.

  3. Attente bornée: aucun processus ne doit attendre indéfiniment avant d'entrer en section critique.

  4. Aucune hypothèse ne doit être faite sur les vitesses relatives des processus.

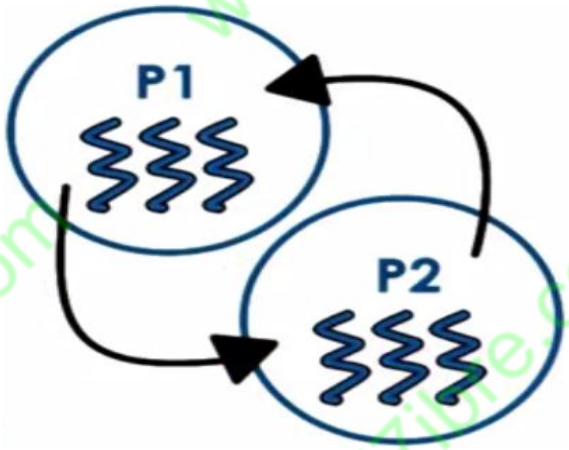
1. Notions: Accès concurrents aux ressources

- Comment gérer les accès concurrents aux ressources ?
- Mécanismes de:
  - Communication: Echange de données entre processus, tout en maintenant la protection ainsi que l'isolation entre processus communicants.
  - Synchronisation: La relation de dépendance logique entre processus qui cadence leur évolution et fixe l'ordre de leur exécution dans le temps (*i.e.* s'affecter mutuellement).

1. Notions de base
2. Mécanismes de communication
3. Mécanismes de synchronisation

2. Mécanismes de Communications

- La communication interprocessus (IPC: interprocess communication) comparée à la communication entre ouvriers.



- **Les ouvriers** partagent un espace de travail: un dépôt d'outils et de pièces nécessaires pour la confection de voitures.
- **Les ouvriers** appellent les uns et les autres: expliciter les demandes et les réponses.
- Besoin de synchronisation: l'un commence sa tâche après la fin de celle de l'autre.

- **Les processus** partagent un espace de travail: une mémoire partagée.
- **Les processus** appellent les uns et les autres: passage de messages (message passing).
- Besoin de synchronisation: mécanismes de synchronisation.

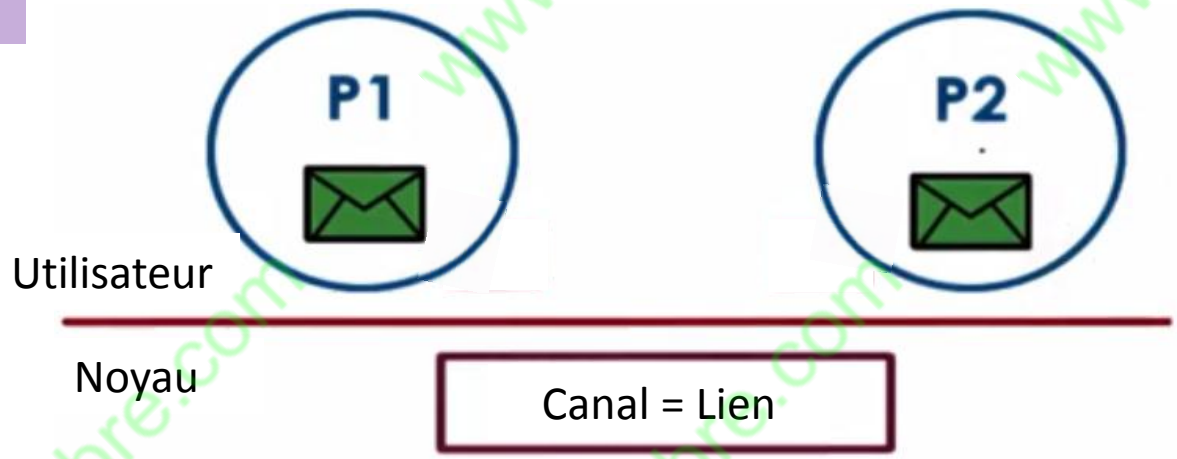
2. Mécanismes de Communications

- **IPC:** un ensemble de mécanismes que l'OS supporte pour permettre aux processus d'interagir entre eux (coordiner, communiquer).
- Les mécanismes IPC sont catégorisés comme suit:
  - Mécanismes à passage de messages (messages passing)
  - Mécanismes à mémoire partagée (shared memory)
- On verra que les IPC comprennent la notion de synchronisation.

2. Mécanismes de Communications

1)Message Passing

1.1)Principes

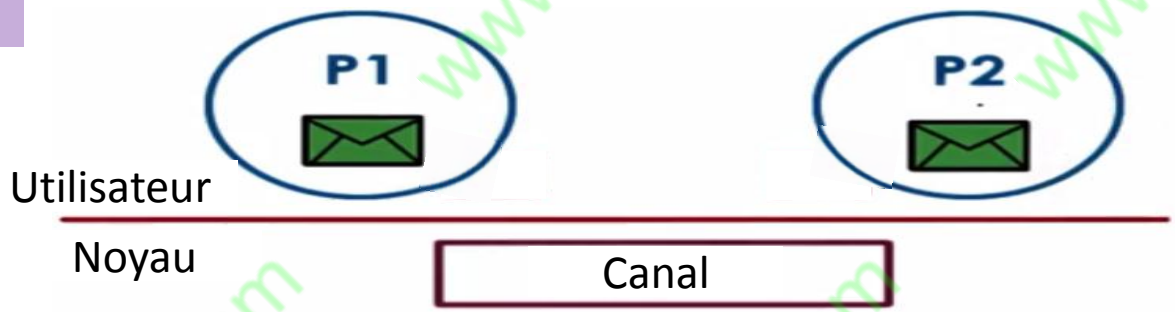


- Les processus créent des messages, puis les **envoient (écrire)** ou **les reçoivent (lire)**.
- Le noyau OS **établi et maintient le canal** qui sera utilisé pour transmettre les messages entre les processus et est requis pour **effectuer toutes les opérations IPC**.
- Le canal (ou le lien), qui peut être implémenté sous forme d'une file d'attente FIFO par exemple, est responsable de **transmettre le message** d'un processus à un autre.
- Ce lien peut être unidirectionnel ou bidirectionnel.

2. Mécanismes de Communications

1)Message Passing

1.1)Principes



- Coût des opérations:
  - L'envoi: appel système + copie du message depuis l'espace adresse du processus vers le canal.
  - La réception: appel système +copie du message depuis le canal vers l'espace adresse du processus de réception.
  - 1 communication= 2 copies de données + 2 passages utilisateur/noyau.
- Inconvénients:
  - coût généré (overhead) dû aux multiples copies de données en entrée et en sortie depuis ou vers le noyau + les passages multiples utilisateur/noyau.

✓Avantages:

- Le noyau du SE prend en charge toutes les opérations, concernant la gestion des canaux.
- La synchronisation: le noyau s'assurera que les données ne sont pas écrasées ou corrompues d'une façon ou d'une autre, quand les processus tentent d'envoyer ou de recevoir en même temps.

2. Mécanismes de Communications

1)Message Passing

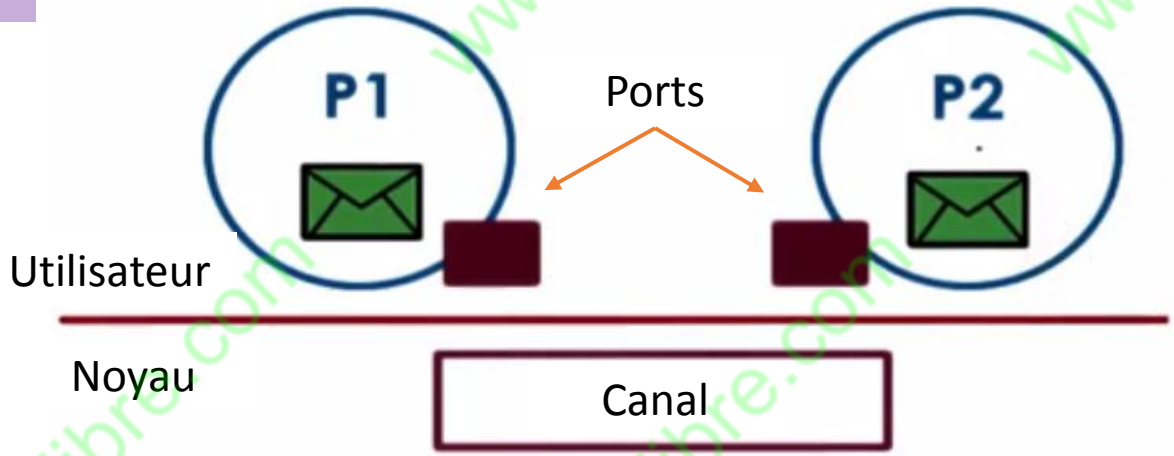
1.1)Principes

- Le Message passing peut être bloquant ou non-bloquant.
- Bloquant c'est à dire synchrone
  - Envoie bloquant: l'expéditeur est bloqué jusqu'à la réception du message.
  - Réception bloquante: le récepteur est bloqué jusqu'à ce qu'un message soit disponible.
- Non-bloquant c'est à dire asynchrone
  - Envoi non-bloquant: l'expéditeur envoie le message et continue.
  - Réception non-bloquante: le récepteur reçoit sans attendre.
- Si à la fois envoyer et recevoir bloquent, nous avons besoin d'un **rendez-vous**.

2. Mécanismes de Communications

1)Message Passing

1.1)Principes



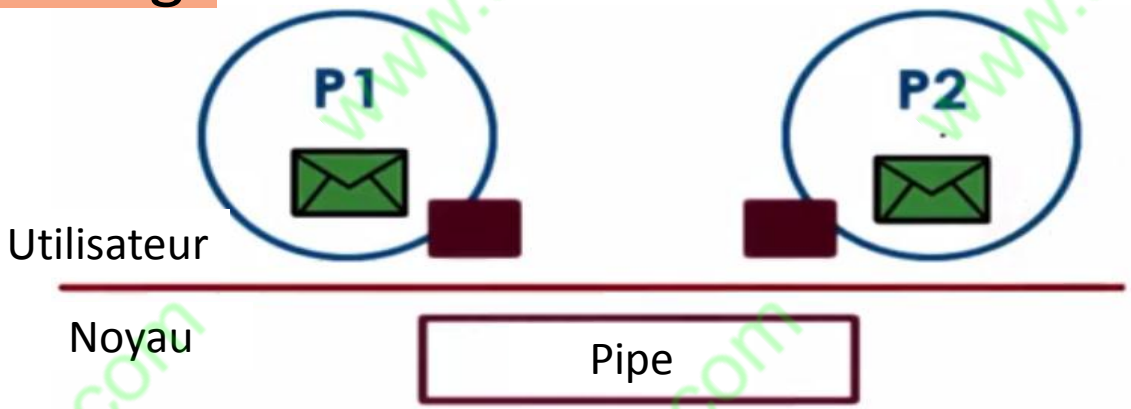
- Le message passing implémente la notion de port.
- Le port est une interface par laquelle un processus peut, entre autres, envoyer ou recevoir un message.

2. Mécanismes de Communications

1)Message Passing

1.2)Les pipes

a) Pipe ordinaire



- **Les pipes ordinaires** permettent à deux processus de communiquer de la manière standard producteur-consommateur: le producteur écrit à une extrémité du tuyau (WRITE_END) et le consommateur lit à l'autre extrémité (READ_END).
- Il n'y a pas de message en soi mais plutôt un flux d'octets poussés dans le pipe depuis un processus puis reçu dans un autre.
- Les pipes ordinaires sont unidirectionnels.
- Les pipes peuvent être accédées en utilisant les appels système read () et write ().
- Un pipe ordinaire n'est pas accessible depuis l'extérieur du processus qui l'a créé:
  - Le processus parent crée un pipe et l'utilise pour communiquer avec un processus fils qu'il crée via fork ().
- Les pipes ordinaires peuvent être utilisés uniquement pour la communication entre les processus sur la même machine.
- Une fois que les processus ont fini de communiquer et sont terminés, le pipe ordinaire cesse d'exister.

2. Mécanismes de Communications

1)Message Passing

1.2)Les pipes

b) Pipe nommé

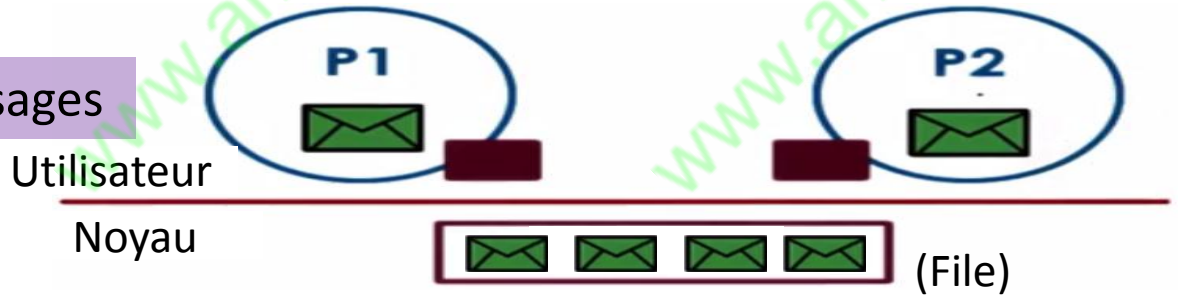
- La communication pour **les pipes nommés** peut être bidirectionnelle et aucune relation parent-enfant n'est requise.
- Une fois qu'un pipe nommé est établi, plusieurs processus peuvent l'utiliser pour la communication.
- Les pipes nommés continuent d'exister après la fin des processus communicants jusqu'à ce qu'ils soient explicitement supprimés du système de fichiers.
- **UNIX:**
  - Seule la transmission half-duplex est autorisée.
  - Si les données doivent passer dans les deux sens simultanément, deux pipes sont généralement utilisés.
  - Les processus de communication doivent résider sur la même machine.
  - Si une communication inter-machine est requise, les **sockets** doivent être utilisées.
- **WINDOWS:**
  - La communication en full-duplex est autorisée.
  - Les processus en communication peuvent résider sur la même machine ou des machines différentes.

2. Mécanismes de Communications

1)Message Passing

1.3)Les files d'attente de messages

a)Principe



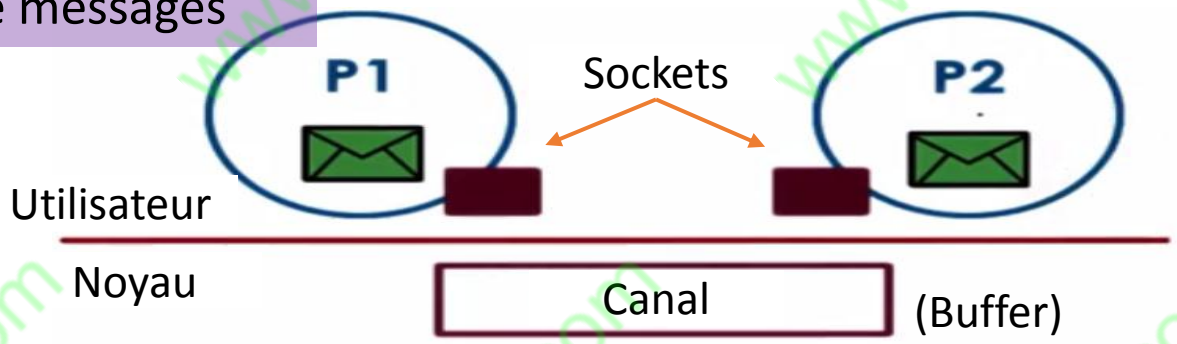
- Un processus émetteur doit envoyer un message correctement formaté au canal, puis le canal fournira un message correctement formaté au processus destinataire, selon le protocole de communication établi entre ces deux processus.
- Plusieurs types de files de messages possibles:
  - **File d'attente sans mémoire:**
    - La file ne stock pas les messages.
    - Ce que l'émetteur dépose dans la file doit immédiatement être retiré par le destinataire. L'émetteur doit bloquer jusqu'à ce que le destinataire reçoive le message.
  - **File d'attente à longueur finie n:**
    - Si la file d'attente n'est pas pleine lorsqu'un nouveau message est envoyé, le message est placé dans la file d'attente et l'émetteur peut continuer l'exécution sans attendre.
    - Si elle est pleine, l'émetteur doit bloquer jusqu'à ce qu'un espace soit disponible dans la file d'attente.
  - **File d'attente à capacité illimitée:**
    - La longueur de la file est potentiellement infinie.
    - L'émetteur ne bloque jamais.
- Le SE fournit des mécanismes aux niveaux des files d'attente de messages pour intégrer également la notion de priorités des messages ou la planification des envoies des messages.

2. Mécanismes de Communications

1)Message Passing

1.3)Les files d'attente de messages

b)Les sockets

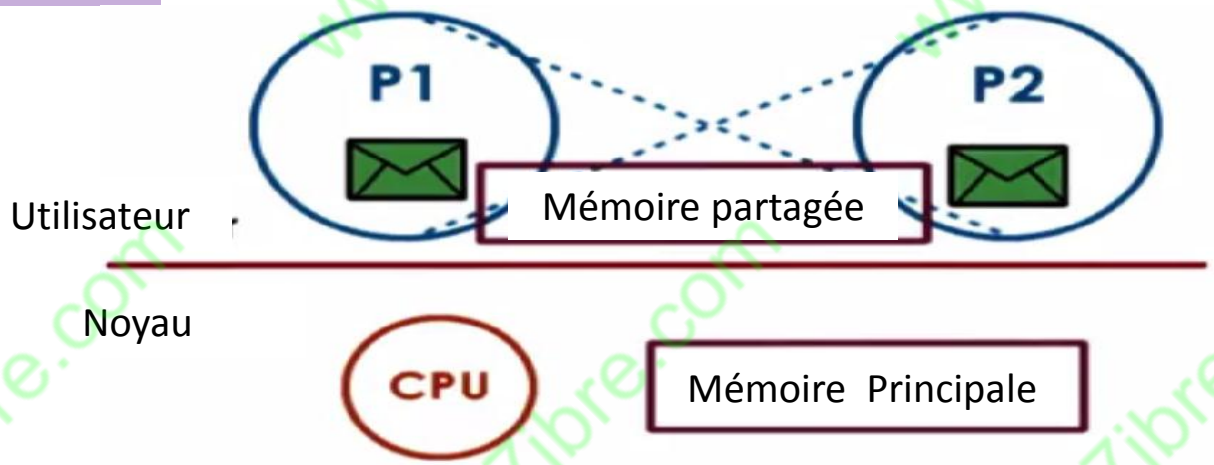


- Les sockets permettent aux processus d'envoyer des messages à l'intérieur et à l'extérieur du buffer de communication dans le noyau.
- Un socket est identifié par une adresse IP concaténée à un numéro de port.
- En général, les sockets utilisent une architecture client-serveur.
- L'appel socket():
  - Crée une mémoire buffer au niveau du noyau.
  - Associe tout le traitement nécessaire au niveau du noyau pour la transmission du message.
- Le socket peut être un socket TCP/IP, ce qui signifie que l'ensemble de la pile de protocoles TCP/IP est associé au mouvement des données dans le noyau.
- Le SE est suffisamment intelligent pour comprendre que si deux processus sont sur la même machine, il n'a pas vraiment besoin d'exécuter la pile de protocoles complète pour envoyer les données sur le réseau, puis de le recevoir et le passer au processus.

2. Mécanismes de Communications

2)Mémoire partagée

2.1)Principe

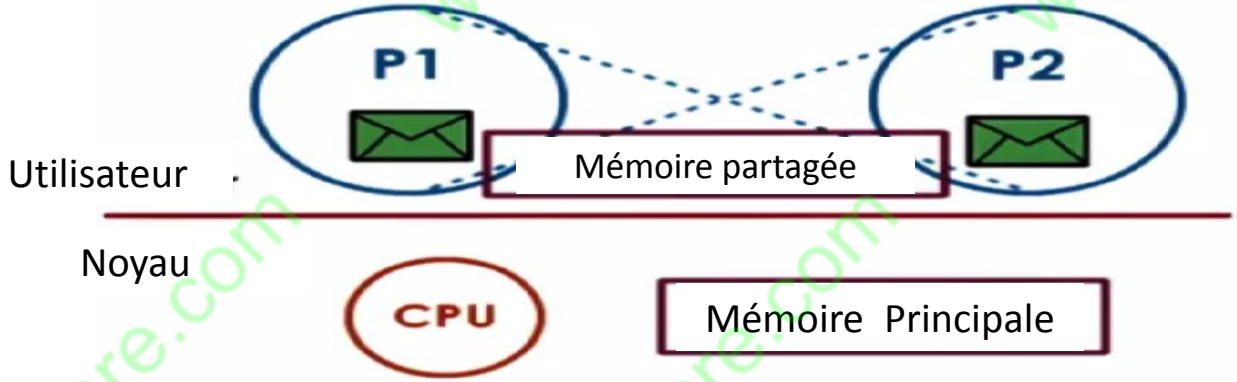


- Les processus **envoient (écrire)** ou **reçoivent (lire)** les messages dans une région partagée de la mémoire.
- Le noyau SE **établi** la mémoire partagée entre les processus.
- Un **même espace de la mémoire physique** peut être accessible par ces processus. C'est à dire qu'une adresse logique de P1 et une autre de P2 vont correspondre à la même adresse physique dans la mémoire principale.
- ***Normalement***, le SE **empêche un processus d'accéder à la mémoire des autres processus**. Lorsque le mécanisme de mémoire partagée est utilisé, cela nécessite que les processus communicants **<u>suppriment cette restriction</u>**.

2. Mécanismes de Communications

2)Mémoire partagée

2.1)Principe

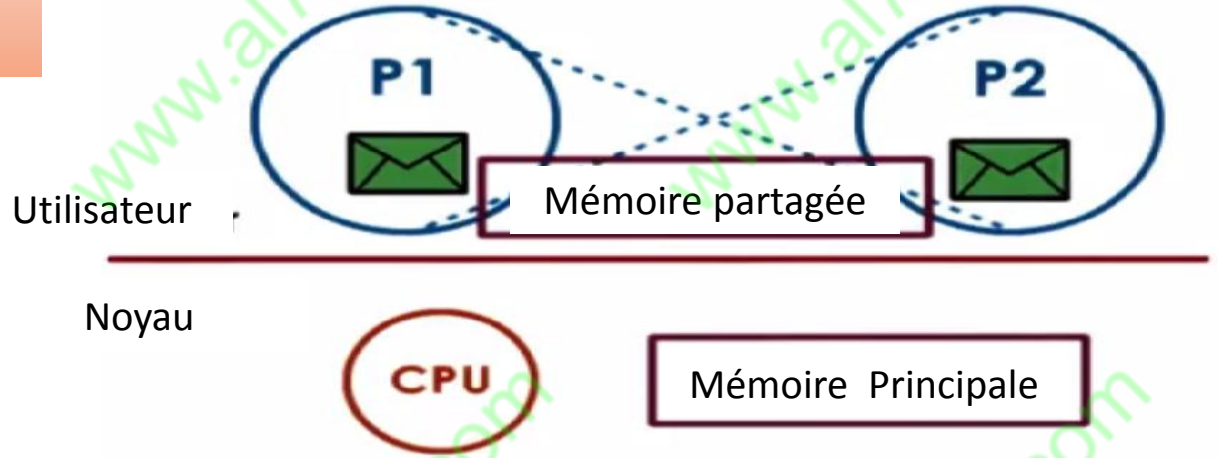


- Une mémoire partagée réside dans **l'espace d'adressage du processus** qui l'a créé. Les autres processus qui souhaitent communiquer à l'aide de cette région doivent la **joindre à leur espace d'adressage**.
- Les processus peuvent ensuite échanger des informations en **lisant et écrivant** des données dans la région partagée.
- Les processus sont également responsables de s'assurer qu'ils n'écrivent pas au même endroit **simultanément**. Cela est géré par les mécanismes de **synchronisation**.

2. Mécanismes de Communications

2)Mémoire partagée

2.1)Principe



✓Avantages:

- **Plus rapide** que le message passing car les appels système sont requis uniquement pour établir la région de la mémoire partagée.
- **Réduction du nombre de copies** de données. Un processus peut utiliser une donnée dans la mémoire partagée sans avoir besoin de la copier.

- Inconvénient:
  - **C'est au programmeur de gérer** les accès et l'organisation de la mémoire partagée. La difficulté majeur est de gérer la **synchronisation**: les processus doivent synchroniser explicitement leurs accès à la mémoire partagée.

2. Mécanismes de Communications

2)Mémoire partagée

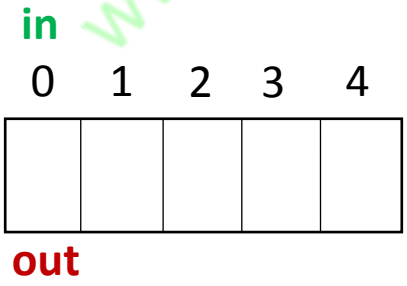
2.2) Problème du producteur/consommateur

- Le problème majeur de la mémoire partagée est la gestion de la synchronisation.
- **La synchronisation** permet de gérer les accès à la mémoire partagée grâce à plusieurs mécanismes.
- Un mécanisme simple est d'utiliser le concept du **problème du producteur/consommateur:**
  - Le processus producteur (**P**) ne peut que produire (**écrire**) des informations.
  - Le processus consommateur (**C**) ne peut que consommer (**lire**) ces informations.
- **Déroulement:**
  - Pour permettre l'exécution simultanée de P et C, un **buffer** est rempli par P et est vidé par C. Le buffer réside dans la région mémoire partagée par P et C.
  - P peut produire une information pendant que C consomme **une autre**.
  - P et C doivent être **synchronisés**, de sorte que C n'essaie pas de consommer une information qui **n'a pas encore été produite**.
  - **Buffer:** tableau circulaire avec deux pointeurs **in** (modifié par P et indique la prochaine case vide) et **out** (modifié par C et indique la première case pleine).

2. Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



1) Etat initial:
- buffer vide: **in=out**
- Si P ne produit rien, C ne peut rien consommer

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

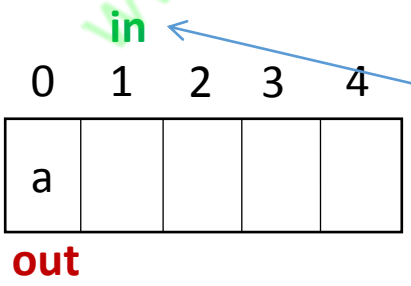
```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

2. Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



2) P produit l'item a dans next_prod:
- next_prod est mis dans la case 0
- **in** est incrémenté par 1

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

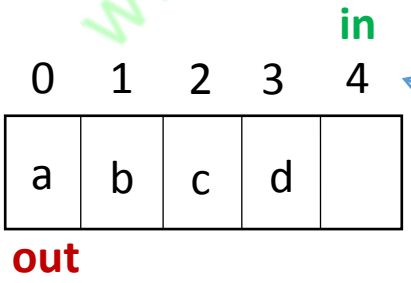
```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

2. Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



3) P produit 3 autres items:
- A chaque itération, next_prod est mis dans la case **in**
- **in** est incrémentée et atteint case 4

4) P ne pourra par produire un 5ème item: après cela, il devra pointer sur la case 0 (= 5 mod 5) contenant un item pas encore lu par C (pointé par **out**).

```
//mémoire partagée
#define B_S 5 /* taille du buffer
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

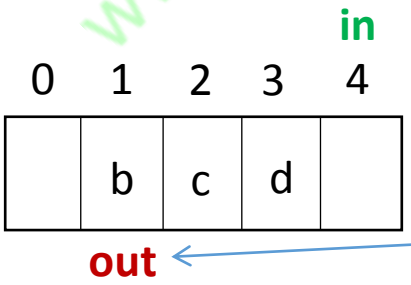
```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

2. Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



5) C consomme un item:
- L'item à la case pointée par **out** (=0) est mis dans next_cons
- **out** est incrémentée

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

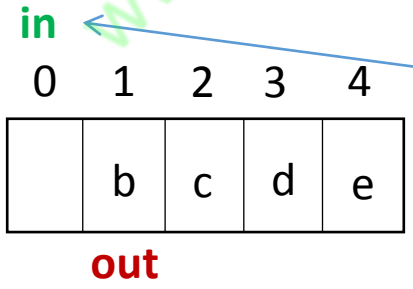
```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

2. Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



4-bis) P pourra produire un 5ème item: le contenu de la case 0 est consommé, elle est libre d'accès (car **in ≠ out**)

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

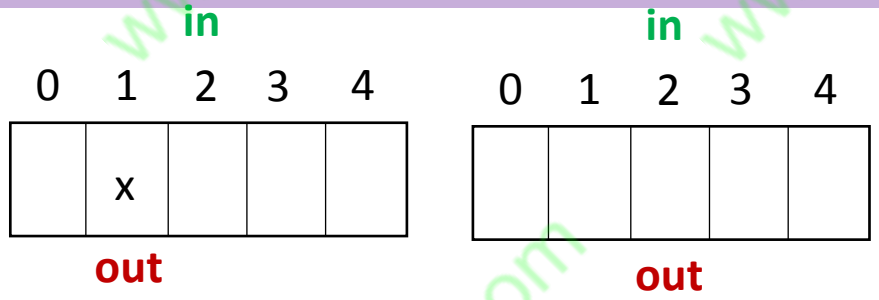
```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

2. Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



6) Supposant la configuration à gauche (**out** précède **in** de 1) et C consomme le contenu de la case 1:
- out pointe alors sur case 2 (= **in**).
-C ne pourra plus consommer d'item (car **in**=**out**)

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

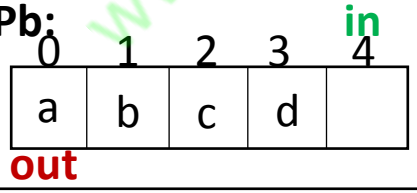
```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

2. Mécanismes de Communications

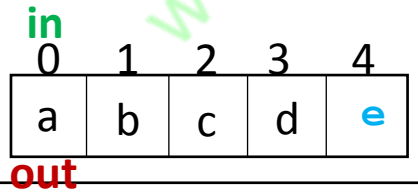
2)Mémoire partagée

2.2) Problème du producteur/consommateur

Pb:



Sol:



Problème de l'algo: Si B_S = 5, P ne peut déposer que **4 items à la fois** (étape 4).

Solution: Introduire une variable globale (**counter**):

- initialisée à 0
- utilisée dans les boucles while.
- incrémentée à chaque production
- décrémentée à chaque consommation.

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
int counter=0;
```

```
//Producteur P
item next_prod;
while (true) {
/* produire un item dans  next_prod */
While( counter == B_S ); //buffer
plein, ne rien faire
buffer[in] = next_prod;
in = (in + 1) % B_S;
counter++; }
```

```
//Consommateur C
item next_cons;
while (true) {
while ( counter == 0 );//buffer
vide, ne rien faire
next_cons = buffer[out];
out = (out + 1) % B_S;
counter--;
/* consommer un item dans
next_cons */ }
```

1. Notions de base
2. Mécanismes de communication
3. Mécanismes de synchronisation

3. Mécanismes de synchronisation

1) Introduction

- **Synchronisation:** elle se présente comme un ensemble de mécanismes qui permettent aux processus d'accéder à leur section critique en garantissant l'exclusion mutuelle.
- Les mécanismes de synchronisation sont catégorisés comme suit:
  - Mécanismes logiciels
  - Mécanismes matériels
  - Combinaison des deux

On introduit la synchronisation à travers la solution proposée (i.e. variable `counter`) au problème rencontré dans l'algorithme du producteur/consommateur.

3. Mécanismes de synchronisation

1) Introduction

1.1) Problème de l'algo du producteur/consommateur

- L'incrémentation et décrémentation de counter est exécutée sous forme de 3 instructions en assembleur:

counter++ :
- reg1=counter
- reg1 = reg1+1
- counter = reg1

counter-- :
- reg2 = counter
- reg2 = reg2-1
- counter = reg2

- On se positionne dans le cas où les processus s'exécutent en parallèle et que P exécute counter++ et C counter--. Considérons que counter = 5 et que les instructions s'exécutent dans cet ordre:
  - Inst0: P exécute reg1 = counter {reg1 = 5}
  - Inst1: P exécute reg1 = reg1 + 1 {reg1 = 6}
  - Inst2: C exécute reg2 = counter {reg2 = 5}
  - Inst3: C exécute reg2 = reg2 – 1 {reg2 = 4}
  - Inst4: P exécute counter = reg1 {counter = 6 }
  - Inst5: C exécute counter = reg2 {counter = 4} **Cette valeur doit être = 5**

- **Problème:** les processus P et C **accèdent en même temps** à une variable partagée counter
- **Solution:** introduire une **section critique** pour l'accès à counter

3. Mécanismes de synchronisation

1) Introduction

1.2) Section critique

- La structure générale d'un programme utilisant une section critique :

```
do{
        Entrée section critique

          // code section critique

        Sortie section critique

          // code non critique

}while{true}
```

- **Rappel:** Lorsqu'on introduit une section critique, il faut s'assurer de satisfaire les conditions d'exclusion mutuelle:
  1. Exclusion mutuelle
  2. Pas d'Interblocage
  3. Attente bornée
  4. Pas d'hypothèse sur les vitesses relatives des processus.

3. Mécanismes de synchronisation

2) Mécanisme logiciel

2.1) Introduction

- **Synchronisation par mécanisme logiciel:** l'accès à la section critique est contrôlé par un algorithme uniquement et n'a pas besoin de circuit spécial.
- **La Solution de Peterson** en est un exemple:
  - Elle est adaptée au cas de deux processus
  - Les processus partagent deux variables :
    - Int **turn**: indique l'indice du processus (1 ou 2) qui entre dans la section critique
    - Boolean **flag[2]**: flag[k]==True indique que le processus « k+1 » souhaite entrer dans la section critique
  - Remarque: *i* est l'indice du processus courant, *j* est l'indice de l'autre processus

3. Mécanismes de synchronisation

2) Mécanisme logiciel

2.2) Solution de Peterson

```
do{// le processus
    flag[i-1] = true; // inst 1
    turn = j; // inst 2
    while(flag[j-1] && turn = = j);  // inst 3
// code section critique
    flag[i-1] = false; // inst 4
// code non critique
}while{true};
```

Les conditions de l'exclusion mutuelle sont satisfaites:

1) Pas d'accès simultané à la SC
2) Pas d'interblocage
3) Les processus ont accédé à leur SC.
4) Pas de supposition sur la vitesse des processus

- 2 processus P1 et P2 s'exécutent en même temps
- Initialement: `turn=0 | flag=[F,F]`
- P1 puis P2 →inst 1 `turn=0 | flag=[T,T]`
- P1 → inst 2: `turn=2 | flag = [T,T]`
- P1 → inst 3: While(T), P1 boucle et ne peut pas accéder à sa SC
- P2 → inst 2: `turn=1 | flag = [T,T]`
- P1 → inst 3: While(F), P1 entre en SC
- P2 → inst 3: While(T), P2 boucle et ne peut pas accéder à sa SC
- P1 termine SC → inst 4: `turn=1 | flag=[F,T]`
- P2 → inst 3: While(F), P2 entre en SC
- P1 entre en section non critique, puis termine
- P2 termine SC → inst 4: `turn=1 | flag=[F,F]`
- P2 entre en section non critique, puis termine

3. Mécanismes de synchronisation

2) Mécanisme logiciel

2.2) Solution de Peterson

- La Solution de Peterson peut **poser un problème** lorsqu'on dispose d'un processeur superscalaire (comporte plusieurs unités de calcul).
- Ce processeur peut exécuter plusieurs instructions simultanément parmi une suite d'instructions. Soit les instructions suivantes traitées par processeur qui exécute 2 instructions à la fois :

```
1. Mov eax,0
2. Mov ebx,1
3. Mov edx,2
4. Inc edx
5. Mov ecx,3
```

```
1. Mov eax,0
2. Mov ebx,1
3. Mov edx,2
5. Mov ecx,3
4. Inc edx
```

- Le processeur choisit à chaque fois 2 instructions qui n'agissent pas sur les mêmes registres.
- Il peut alors exécuter : (1,2); (3,5); (4)
- Cela produit un changement dans l'ordre des instructions (**memory reordering**).

3. Mécanismes de synchronisation

2) Mécanisme logiciel

2.2) Solution de Peterson

- Si la solution de Peterson rencontre un changement d'ordre des instructions. On peut imaginer le scénario suivant:

```
do{// le processus
   flag[i-1] = true; // inst 1
   turn = j; // inst 2
   while(flag[j-1] && turn = = j)
   ; // inst 3
// code section critique
   flag[i-1] = false; // inst
   4
// code non critique
}while{true}
```

- Initialement: `turn=0 | flag = [F,F]`
- P1 →inst 2 puis P1 → inst 3:

`turn=2 | flag=[F,F] et P1 entre en SC`

- P2 → inst 2 puis P2 → inst 3:

`turn=1 | flag=[F,F] et P2 entre en SC`

- P1 → inst 1 `turn=1 | flag = [T,F]`
- P2 → inst 1: `turn=1 | flag = [T,T]`
- …..

Une condition de l'exclusion mutuelle n'est pas satisfaite:

1) accès simultané à la SC par les deux processus

Solution: Mécanisme matériel

3. Mécanismes de synchronisation

2) Mécanisme matériel

- **Synchronisation par mécanisme matériel:** l'accès à la section critique est contrôlé par un circuit spécial atomique qui ne peut pas être interrompu pendant son exécution.
- Le principe commun des mécanismes matériels est d'utiliser un **verrou** (lock) pour bloquer/débloquer l'accès à la section critique.

```
do{
        Acquire Lock (verrouillage)

            // code section critique

        Release Lock (déverrouillage)

            // code non critique

}while{true}
```

- Il existe plusieurs mécanismes matériels, parmi eux:
  - La solution test_and_set()
  - Mutex (API)

- **Note:** les algorithmes des solutions matérielles présentés ci-après décrivent le comportement du circuit atomique utilisé. Ce circuit ne peut jamais être utilisé par deux ou plusieurs processus à la fois.

3. Mécanismes de synchronisation

2) Mécanisme matériel

2.1) Solution test_and_set()

```
boolean test_and_set(boolean
*oldLock) // circuit atomique
{
  boolean newLock=*oldLock;
  *oldLock=true;
  return newLock;
}
```

- lock est une variable globale et son passage se fait par référence (&lock).

```
boolean lock = false; //initialisation
```

```
do{// le processus
  while(test_and_set(&lock)) ;//inst 1

// code section critique
    lock = false; // inst 2
// code non critique
}while{true}
```

Cas de deux processus:

- P1 →inst 1 `lock=T | while(F)` P1 dans SC.
- P2 →inst 1 `lock=T | while(T)` P2 attend.
- P1 →inst 2 `lock=F` P1 dans SNC
- P2 →inst 1 `lock=T | while(F)` P2 dans SC
- P2 →inst 2 `lock=F` P2 dans SNC

Les conditions de l'exclusion mutuelle sont satisfaites:

1) Pas d'accès simultané à la SC
2) Pas d'interblocage
3) Les processus ont accédé à leur SC.
4) Pas de supposition sur la vitesse des processus

3. Mécanismes de synchronisation

2) Mécanisme matériel

2.1) Solution test_and_set()

```
boolean test_and_set(boolean *oldLock)
// circuit atomique
{
  boolean newLock=*oldLock;
  *oldLock=true;
  return newLock;
}
```

```
boolean lock = false; //initialisation
```

```
do{// le processus
  while(test_and_set(&lock)) ;//inst 1
// code section critique
    lock = false; // inst 2
// code non critique
}while{true}
```

Cas de trois processus:

- P1 →inst 1 `lock=T | while(F)` P1 dans SC.
- P2 →inst 1 `lock=T | while(T)` P2 attend.
- P1 →inst 2 `lock=F` P1 dans SNC.
- P2 →inst 1 `lock=T | while(F)` P2 dans SC
- P3→inst 1 `lock=T | while(T)` P3 attend.
- P1→inst 1 `lock=T | while(T)` P1 attend.
- P2 →inst 2 `lock=F` P2 dans SNC.
- P1→inst 1 `lock=T | while(F)` P1 dans SC.
- ……

Une condition de l'exclusion mutuelle n'est pas satisfaite:
3) P3 n'a pas accédé à sa SC, alors que P1 y a accédé deux fois.

Solution:
Réécrire le code en ajoutant d'autres variables pour que les processus puissent accéder autant de fois les uns et les autres

3. Mécanismes de synchronisation

2) Mécanisme matériel

2.2) Mutex

- Pour simplifier l'utilisation des solutions matérielles aux programmeurs, les concepteurs des SE ont proposé des outils logiciels (API) permettant l'appel de ses solutions.
- Le verrou mutex en fait partie. Il permet d'appeler un mécanisme de synchronisation matériel tel que test_and_set():
  - La fonction acquire() permet d'établir le verrou sur la section critique
  - La fonction release() libère le verrou.

```
do{
        acquire ()

        // code section critique

        release ()

        // code non critique

}while(true);
```

```
acquire()
{
while(test_and_set(&lock));
// busy wait (attente active)
}
```

```
release()
{
  lock=true;
}
```

3. Mécanismes de synchronisation

2) Mécanisme matériel

2.3) Problème du busy wait

- Lorsqu'un processus entre dans la section critique, les autres processus sont occupés à attendre (**busy wait**) sa libération:
  - Dans le cas de la solution `test_and_set()`, la boucle `while(test_and_set(&lock));` occupe le processeur tous le temps d'attente de la libération de la section critique
- Inconvénient: gaspille des cycles processeur.

✓ Avantage:

- Pas de changement de contexte.
- Lorsque la section critique est de courte durée, cela peut être plus avantageux que de céder le processeur à un autre processus et gaspiller du temps en changement de contexte.

- Le problème du busy wait est résolu par **les sémaphores**

3. Mécanismes de synchronisation

3) Les sémaphores

3.1) Définition

- Dans le domaine ferroviaire, un sémaphore est un signal permettant de déterminer si l'accès à la voie est libre (sémaphore ouvert) ou non (sémaphore fermé).
- Pour résoudre le problème du busy wait, on utilise un **sémaphore** représenté par une variable entière qui garantit l'exclusion mutuelle, accompagnée d'une liste de processus en attente pour accéder à la SC.
- En dehors de l'initialisation, le sémaphore n'est accessible que par deux opérations `wait()` et `signal()` (aussi notées dans la littérature `P()` et `V()`, `down()` et `up()`).

Fermé

Ouvert

3. Mécanismes de synchronisation

3) Les sémaphores

3.2) Solution au busy wait

```
Semaphore S | S.value=1 | S.list=Ø // initialisation
RQ_UCT=Ø // liste attente prêt de l' UCT
```

```
while(true){//Processus
  wait(&S); //inst 1
 //section critique
  signal(&S); //inst 2
//section non critique
}
```

```
typedef struct{
int value;
struct process *list;
} semaphore;
```

```
wait(semaphore *S){
 S.value--;
 if (S.value < 0){
 add_Proc(S.list);
 block();}}
```

```
signal(semaphore *S){
 S.value++;
 if (S.value <= 0){
Proc=remove_Proc(S.list);
 wakeup(Proc);}}
```

- P1 →inst 1 `S.value=0` P1 dans SC.
- P3 →inst 1 `S.value=-1 | S.list=P3` P3 est block.
- P2 →inst 1 `S.value=-2 | S.list=P3,P2` P2 est block.
- P1 →inst 2 `S.value=-1 | S.list=P2` 1er proc (P3) de S.list wakeup et `RQ_UCT=P3` . P1 accède à SNC et termine.
- P3 accède à l'UCT `RQ_UCT= Ø` et entre en SC.
- P3 →inst 2 `S.value=0 | S.list=Ø` P2 wakeup `RQ_UCT=P2` et P3 accède à SNC et termine.
- P2 accède à l'UCT `RQ_UCT= Ø` et entre en SC.
- P2 →inst 2 `S.value=1` P2 accède à SNC et termine.

3. Mécanismes de synchronisation

3) Les sémaphores

3.2) Solution au busy wait

- **Le busy wait est annulé:**
  - Lorsque un processus est placé dans S.list, cela implique un changement de contexte.
  - L'appel de la fonction block() empêche le processus de demander l'accès à l'UCT.
- **Les conditions de l'exclusion mutuelle sont satisfaites:**

  1. Pas d'accès simultané à la SC
  2. Pas d'interblocage
  3. Les processus ont accédé à leur SC: S.list permet d'ordonner l'accès à la file d'attente prêt de l'UCT (pas comme dans test_and_set() cas 3 processus).
  4. Pas de supposition sur la vitesse des processus
- Le sémaphore est **une solution logicielle** pour le problème du busy wait, il y a donc un risque de **changement d'ordre des instructions.**

3. Mécanismes de synchronisation

4) Solution combinée: Sémaphore & mutex

```
while(true){//Processus
 acquire();
  wait(&S); //inst 1
 release();
 //section critique
  signal(&S); //inst 2
//section non critique
}
```

```
typedef struct{
int value;
struct process *list;
} semaphore;
```

```
wait(semaphore *S){
 S.value--;
 if (S.value < 0){
 add_Proc(S.list);
 block();}}
```

```
signal(semaphore *S){
 S.value++;
 if (S.value <= 0){
Proc=remove_Proc(S.list);
 wakeup(Proc);}}
```

```
Semaphore S | S.value=1 | S.list=Ø //initialisation
```

- Pour résoudre le problème du memory reordering, on **combine le sémaphore (solution logicielle) avec une solution matérielle.**
- On inclut donc un mutex autour de l'appel du wait(&S)
- Rappelons que nous avons introduit le sémaphore pour **résoudre le problème du busy wait** provoqué par mutex et les solutions matérielles en général.
- En réalité, établir mutex autour de wait(&S) permet de **réduire le degré du busy wait** (la fonction n'est composée que de 3 instructions), comparé à celui engendré par mutex autour de la section critique (généralement beaucoup plus longue).

3. Mécanismes de synchronisation

5) Problème d'interblocage

- **Interblocage:** deux ou plusieurs processus attendent indéfiniment l'arrivée d'un événement qui ne peut être engendré que par un des processus en attente.
- **Exemple:** Soit P0 et P1 deux processus partageant deux sections critiques, l'une contrôlée par le sémaphore S et l'autre par le sémaphore Q:

  - P0 exécute wait(S), puis P1 éxécute wait(Q).
  - P0 exécute wait(Q) mais il est block(), puis P1 exécute wait(S) et il est block().
  - Aucun processus ne peut avancer dans son exécution, car il a besoin que l'autre processus le débloque par signal(). c'est **l'interblocage**
- Il est donc nécessaire de bien programmer les mécanismes de synchronisation pour éviter ce problème.