**Université de Bouira**
**Faculté des Sciences et des Sciences appliquées**
**Département d'Informatique**
**Licence SI (Systèmes Informatiques)**
**Semestre 3**

# Systèmes d'exploitation 2

**A. ABBAS**
**abbasakli@gmail.com**

# Préambule

- Pré-requis: Cours (algorithmique, SE1, 2ème année Licence).
- Volume horaire hebdomadaire: 1.5H Cours + 1.5H TD + 1.5H TP.
- Évaluation: continu + Examen
- Coefficient : 2, Crédit: 4

# Objectifs de l'enseignement :

Introduire la problématique du **parallélisme** dans les systèmes d'exploitation et étudier la mise en œuvre des mécanismes de **synchronisation**, de **communication** et d'**Interblocage** dans l'environnement centralisé

# CONTENU DU COURS

**1. Introduction**

Introduction aux Systèmes d'exploitation

Concepts du processus

Concepts du threads

**2. Synchronisation**

- Problème de l'exclusion mutuelle
- Synchronisation
  - Evénements, Verrous
  - Sémaphores
  - Moniteurs
  - Régions critiques.
  - Expressions de chemins

**3. Communication**

- Partage de variables (modèles : producteur/ consommateur, lecteurs/ rédacteurs)
- Boites aux lettres
- Echange de messages (modèle du client/ serveur)
- Communication dans les langages évolués (CSP, ADA, JAVA..)

# CONTENU DU COURS

**4. Interblocage**

- Modèles
- Prévention
- Evitement
- Détection/ Guérison
- Approche combinée

**5. Etude de cas : système Unix**

- Principes de conception
- Interfaces (programmeur, utilisateur)
- Gestion de processus, de mémoire, des fichiers et des entrées/sorties
- Synchronisation et Communication entre processus.

# Chapitre 1:

# Introduction

# Introduction aux S.E

**Le système d'exploitation et le système informatique :**

➢Le **matériel** (hardware) d'un système informatique est composé :

de **processeurs** qui exécutent les instructions, de **mémoire centrale** qui contient les données et les instructions à exécuter (en binaire), de **mémoire secondaire** qui sauvegarde les informations et de **périphériques d'Entrées/Sorties** (clavier, souris, écran, modem, etc.) pour introduire ou récupérer des informations.

Les **logiciels** (software), d'un système informatique, sont à leur tour divisés en **programmes système** qui fait fonctionner l'ordinateur : **le système d'exploitation** et les **utilitaires** (compilateurs, éditeurs, interpréteurs de commandes, etc.) et en **programmes d'application** qui résolvent des problèmes spécifiques des utilisateurs.

# Introduction aux S.E

## Définition

- Un Système d'Exploitation (noté SE ou OS, abréviation du terme anglais Operating System) peut être défini comme un logiciel qui , dans un système informatique, pilote les dispositifs matériels et reçoit des instructions de l'utilisateur ou d'autres logiciels (ou applications).

- Le SE  constitue donc une interface entre l'utilisateur, et la machine physique.

# Introduction aux S.E

**Les composants du S.E:**

- **Le noyau** (en anglais kernel) : représentant les fonctions fondamentales du système d'exploitation telles que la gestion de la mémoire, des processus, des fichiers, des entrées-sorties principales, et des fonctionnalités de communication.

- **L'interpréteur de commande** (en anglais shell) permettant la communication avec le système d'exploitation par l'intermédiaire d'un langage de commandes.

- **Le système de fichiers** (en anglais «file system», noté FS), permettant d'enregistrer les fichiers dans le disque sous forme d'une arborescence.
- **Les pilotes :** Les pilotes sont fournis par l'auteur du système d'exploitation ou le fabricant du périphérique.

# Introduction aux S.E

**Rôle du système d'exploitation**

- **Gestion des processeurs:** Gérer l'allocation des processeurs entre les différents programmes grâce à un algorithme d'ordonnancement.
- **Gestion des processus**(programmes en cours d'exécution) : Gérer l'exécution des processus en leur affectant les ressources nécessaires à leur bon fonctionnement.
- **Gestion des mémoires:** Gérer l'espace mémoire alloué à chaque processus.
- **Gestion des fichiers:** Gère l'organisation du disque dur et du système de fichiers
- Donne l'illusion du **multitâche.**

# Concepts du processus

## Définition

- Un processus est l'entité dynamique représentant l'exécution d'un programme sur un processeur

- Un processus est l'activité résultant de l'exécution d'un programme séquentiel, avec ses données, par un processeur.

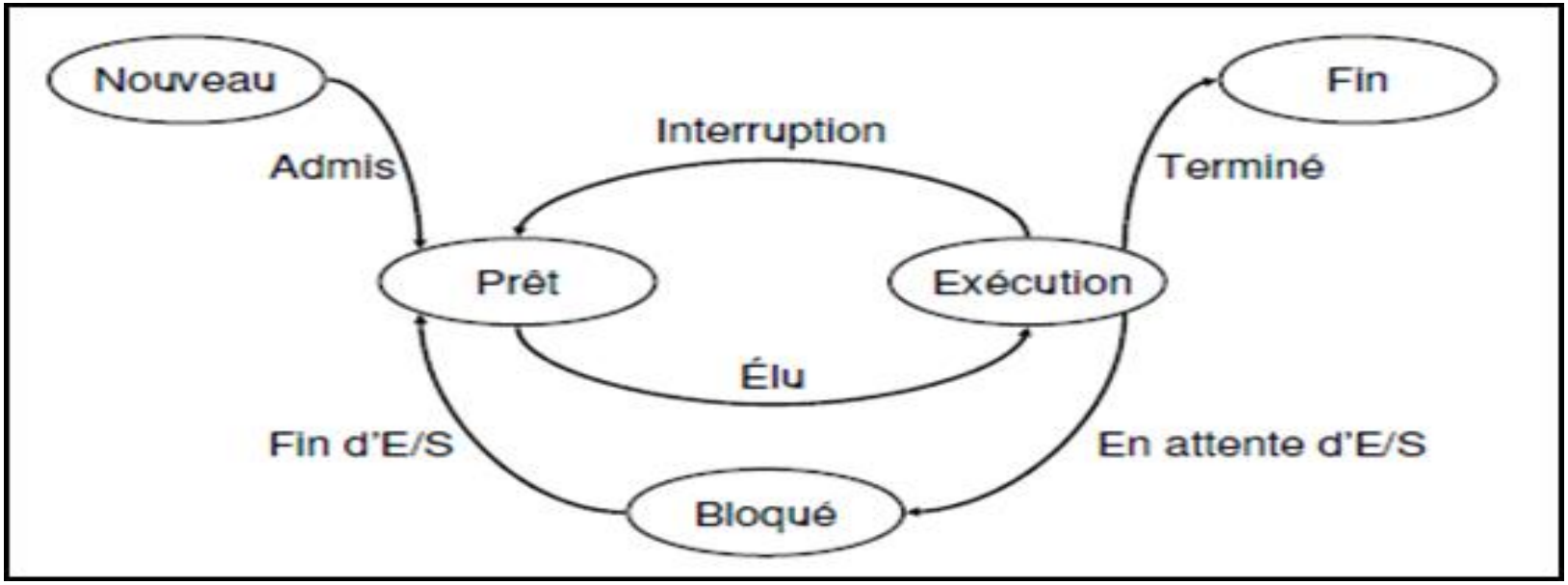
## Un processus est caractérisé par:

1. Un numéro d'identification unique (PID);
2. Un espace d'adressage (code, données, piles d'exécution);
3. Un état principal (prêt, en cours d'exécution (élu), bloqué, …);
4. Les valeurs des registres lors de la dernière suspension (CO, sommet de Pile...);
5. Une priorité;
6. Les ressources allouées (fichiers ouverts, mémoires, périphériques …);
7. Les signaux à capter, à masquer, à ignorer, en attente et les actions associées;
8. Autres informations indiquant le processus père, les processus fils, le groupe, les variables d'environnement, les statistiques et les limites d'utilisation des ressources....

# Concepts du processus

## États d'un processus

Plusieurs processus peuvent se trouver simultanément en cours d'exécution (multiprogrammation et temps partagé), si un système informatique ne comporte qu'un seul processeur, alors, à un instant donné, un seul processus aura accès à ce processeur. En conséquence, un programme en exécution peut avoir plusieurs états. Ces états peuvent être résumés comme suit :



# Concepts du processus

## États d'un processus

**Nouveau** : création d'un processus dans le système

**Prêt** : le processus est placé dans la file d'attente des processus prêts, en attente d'affectation du processeur.

**En exécution :** Le processus est en cours d'exécution.

**Bloqué** : Le processus attend qu'un événement se produise, comme l'achèvement d'une opération d'E/S ou la réception d'un signal.

**Fin**: terminaison de l'exécution

# Concepts du processus

## Le bloc de contrôle et le contexte d'un processus

Pour suivre son évolution, le SE maintient pour chaque processus une structure de données particulière appelée bloc de contrôle de processus (PCB : Process Control Bloc) et dont le rôle est de reconstituer le contexte du processus.

Le contexte d'un processus est l'ensemble des informations dynamiques qui représente l'état d' exécution d'un processus

Le PCB contient aussi des informations sur l'ordonnancement du processus (priorité du processus, les pointeurs sur les files d'attente)

| Numéro de processus |
| --- |
| Etat du processus |
| Compteur d'instruction |
| Registres |
| Limites de la mémoire |
| Liste des fichiers ouverts |
| ... |

# Ordonnancement des processus

**Définition**

L'ordonnanceur (scheduler en anglais) est un programme du SE qui s'occupe de choisir, selon une politique (ou algorithme) d'ordonnancement donnée, un processus parmi les processus prêts pour lui affecter le processeur.

**Les critères d'évaluation entre les algorithmes d'ordonnancement sont:**

**Utilisation du processeur** : Un bon algorithme d'ordonnancement sera celui qui maintiendra le processeur aussi occupe que possible.

**Capacité de traitement** : C'est le nombre de processus terminés par unité de temps.

**Temps d'attente :** C'est le temps passé à attendre dans la file d'attente des processus prêts.

**Temps de réponse :** C'est le temps passé depuis l'admission (état prêt) jusqu'à la terminaison (état fin).

# Ordonnancement des processus

## Les algorithmes d'ordonnancement

1. L'algorithme du Premier Arrive Premier Servi (FCFS)
2. L'algorithme du Plus Court d'abord (SJF)
3. Ordonnancement avec priorité
4. L'algorithme de Round Robin (Tourniquet)
5. Ordonnancement avec les d'attente multiniveaux

# La création de processus

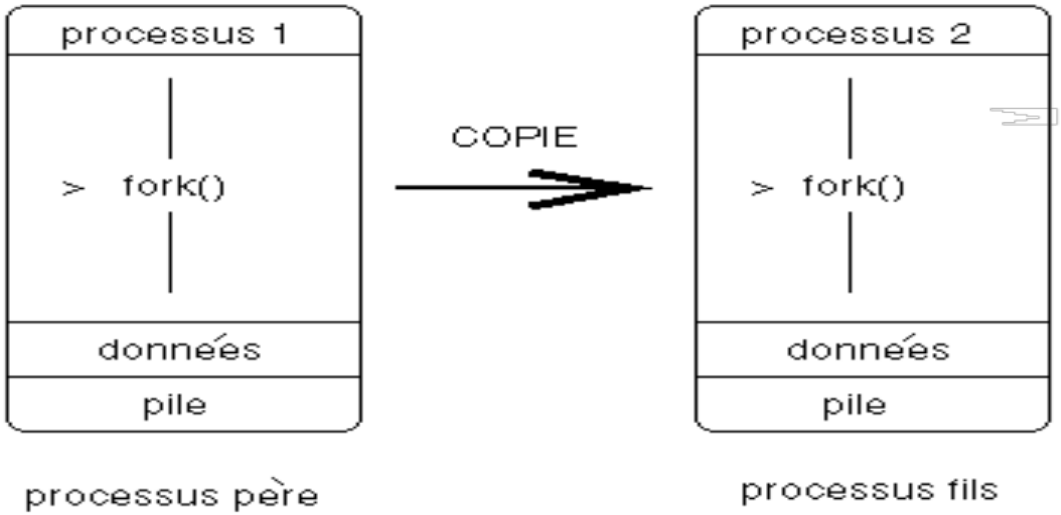
L'appel système ***fork()*** est une façon qui permet de créer des processus aussi appelés **processus lourds**.

Exemple 1 : Instruction de création :

```
#include <unistd.h>
int fork();
```

fork() est le moyen de créer des processus, par duplication d'un processus existant.

L'appel système fork() crée une copie exacte du processus original.



# La création de processus

Pour distinguer le processus père du processus fils on regarde la valeur de retour de fork(), qui peut être:

- La valeur 0 dans le processus fils.
- Positive pour le processus père et qui correspond au PID du processus fils
- Négative si la création de processus a échoué ;

Lors du démarrage de Unix, deux processus sont crées :

1. le Swapper (pid = 0) qui gère la mémoire;
2. le Init (pid = 1) qui crée tous les autres processus.

# La création de processus

**Exemple 2** :L'exécution de Pgfork.c le père et le fils montrent leur PID.

```
//programme Pgfork.c : appel système fork()
#include <sys/types.h> / * typedef pid_t * /
#include <unistd.h> / * fork() * /
#include <stdio.h> / * pourperror, printf * /
int a = 20;
int main() {
pid_t x;
// création d'un fils
switch (x = fork()) {
case −1 : /* le fork a échoué */
printf("le fork a échoué !");
break;
case 0: /* seul le processus fils exécute ce ( case )*/
printf("ici processus fils, le PID %d.\n ", getpid());
a+ = 10;
break;
default: /* seul le processus père exécute cette instruction*/
printf("ici processus père, le PID %d.\n", getpid());
a += 100;
}
// les deux processus exécutent ce qui suit
printf("Fin du Process %d avec a = %d.\n", getpid(), a);
return 0;
}
```

# La hiérarchie des processus

Le système d'exploitation fournit un ensemble d'appels système qui permettent la création, la destruction, la communication et la synchronisation des processus.

Un processus fils peut partager certaines ressources (mémoire, fichiers) avec son processus père ou avoir ses propres ressources. Le processus père peut contrôler l'usage des ressources partagées.

Le processus père peut avoir une certaine autorité sur ses processus fils. Il peut les suspendre, les détruire (appel système **kill**), attendre leur terminaison (appel système **wait**) mais ne peut pas les renier.

# Concept de threads

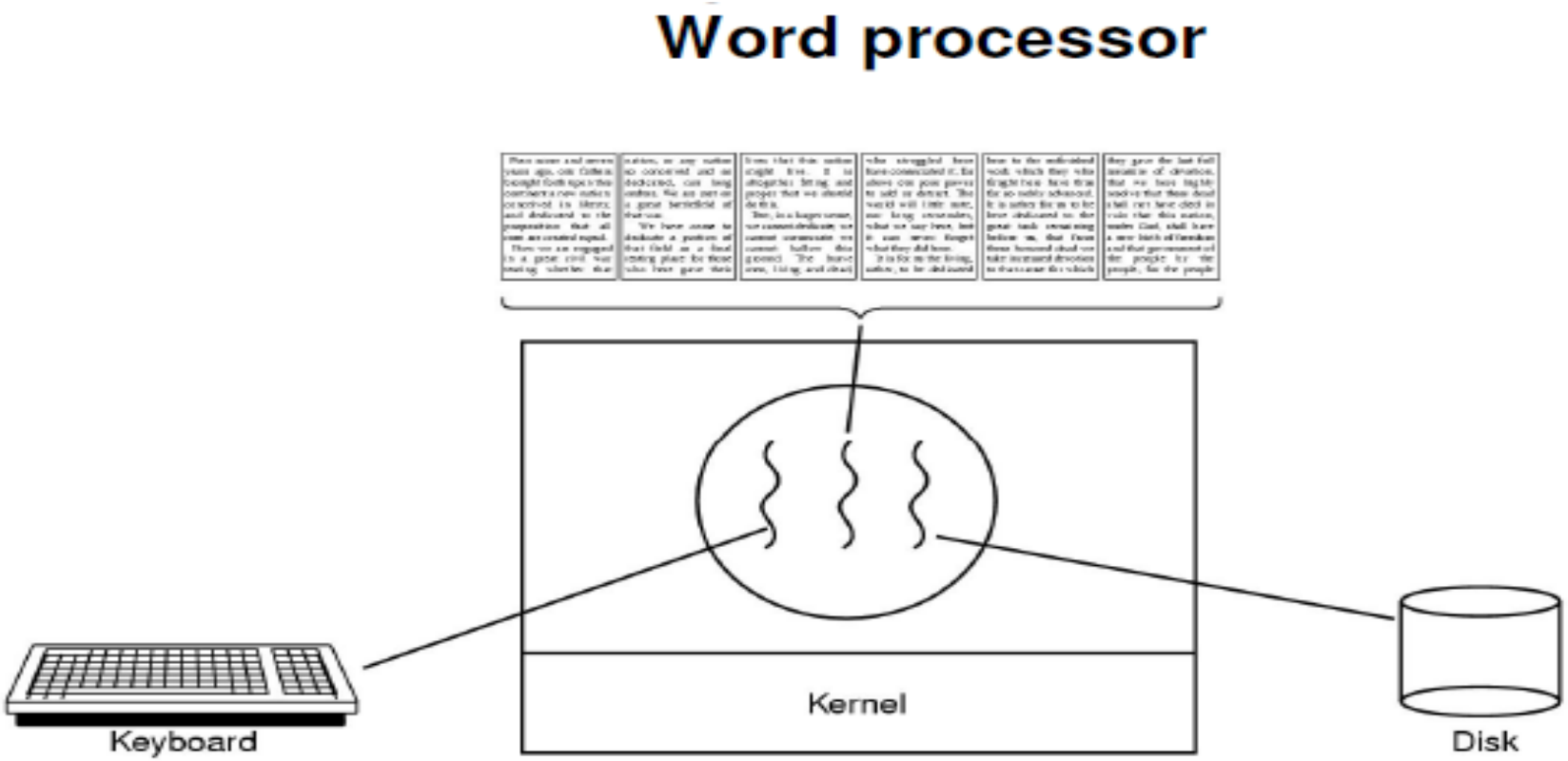
- Les threads appelés aussi processus légers, comme les processus, est un mécanisme permettant a un programme de faire plus d'une chose a la fois (c.-a-d. exécution simultanée des parties d'un programme).

- Un thread est une unité d'exécution rattachée a un processus permettent son exécution.

- Comme les processus, les threads semblent s'exécuter en parallèle; le noyau du SE les ordonnance, interrompant chaque thread pour donner aux autres une chance de s'exécuter.

- Lorsqu'un programme crée un nouveau thread, dans ce cas, rien n'est copie. Le thread créateur et le thread crée partagent tous deux le même espace mémoire, les mêmes descripteurs de fichiers et autres ressources.

# Cas d'utilisation de threads

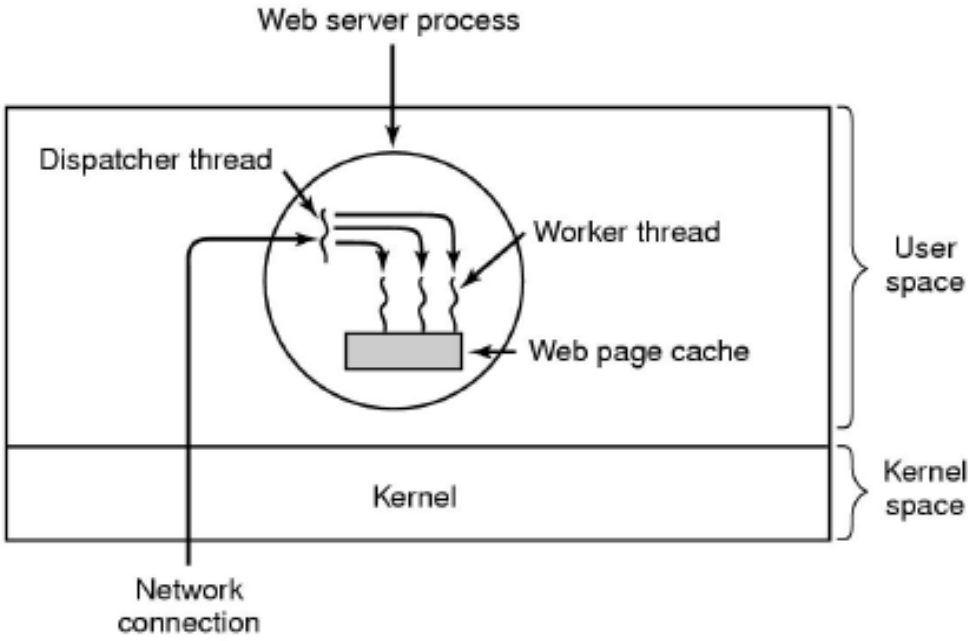



- Un thread pour interagir avec l'utilisateur,
- Un thread pour reformater en arrière plan,
- Un thread pour sauvegarder périodiquement le document

# Cas d'utilisation de threads

## Serveur Web

- Le Dispatcher thread se charge de réceptionner les requêtes.
- Il choisit, pour chaque requête reçue, un thread libre (en attente d'une requête). Ce thread va se charger d'interpréter la requête.




```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

# Création de Threads

La fonction **pthread_create** crée un nouveau thread.

**Syntaxe:**

```
int pthread_create ( pthread_t *thread ,
                     pthread_attr_t *attr,
                     void *nomFonct,
                     void *arg   );
```

**Paramètres du thread :**

- Un pointeur (*thread) vers une variable *pthread_t*, dans laquelle l'identifiant (TID) du thread sera stocke;
- Un pointeur (*attr) vers un objet d'attribut de thread `(taille de la pile, priorité….). Par défaut, il prend NULL
- Un pointeur (* nomFonct) vers la fonction de thread. Il s'agit d'un pointeur de fonction ordinaire de type: void* (*funct) (void*);
- Une valeur d'argument de thread de type void*.

L'appel renvoie 0 s'il réussit, sinon il renvoie une valeur non nulle identifiant l'erreur qui s'est produite

# Création de Threads

**Fonctions pthread_join(), pthread_self() et pthread_exit()**

**void pthread_join( pthread_t tid, void * *status);**

• Attend la fin d'un thread. L'équivalent de waitpid des processus sauf qu'on doit spécifier le tid du thread à attendre.

• status sert à récupérer la valeur de retour et l'état de terminaison.

**pthread_t pthread_self(void);**

• Retourne le TID du thread.

**void pthread_exit( void * status);**

• Termine l'exécution du thread

# Création de Threads

```
#include <pthread.h>
#include <stdio.h>
void* A (void *data){     /* Affiche Hello world */
        printf ("Hello world \n");
        return 0
}
int main (){              /* Le programme principal. Processus*/
        pthread_t thread_id;
/* Cree un nouveau thread. Le nouveau thread exécutera la fonction A */
        Pthread_create (&thread_id, NULL, &A, NULL);
/* Attend la fin du thread.
        pthread_join (thread_id, NULL);
        return 0;
}
```

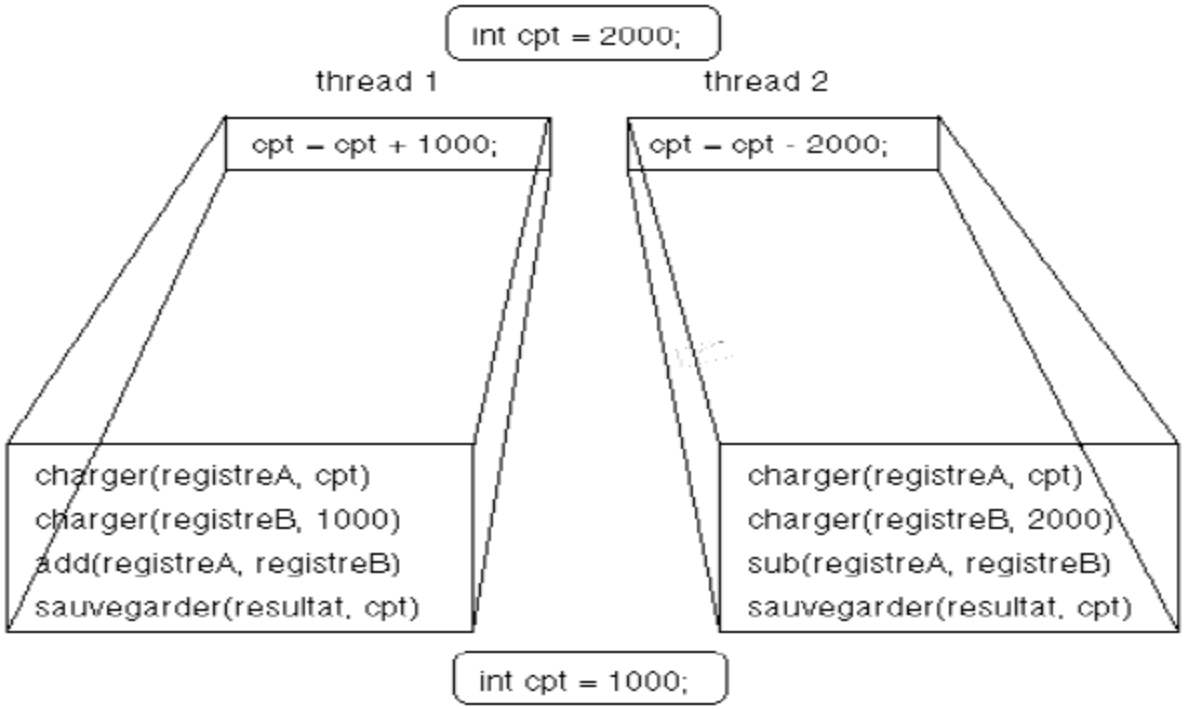
# Problématique

Lorsque plusieurs processus (ou threads) qui s'exécutent sur processeur sont amenés a partager des ressources comme : les périphériques d'entrées-sorties (écrans, imprimantes,..), les moyens de mémorisation (mémoire centrale,...) ou des moyens logiciels (fichiers, base de données, ...) soit **volontairement** s'ils coopèrent pour traiter un même problème, soit **involontairement** parce qu'ils sont obliges de se partager ces ressources vue leurs nombre limite.

Dans ce cas, les processus vont se trouver en situation de concurrence d'accès vis-à-vis de ces ressources offertes par le système d'exploitation.

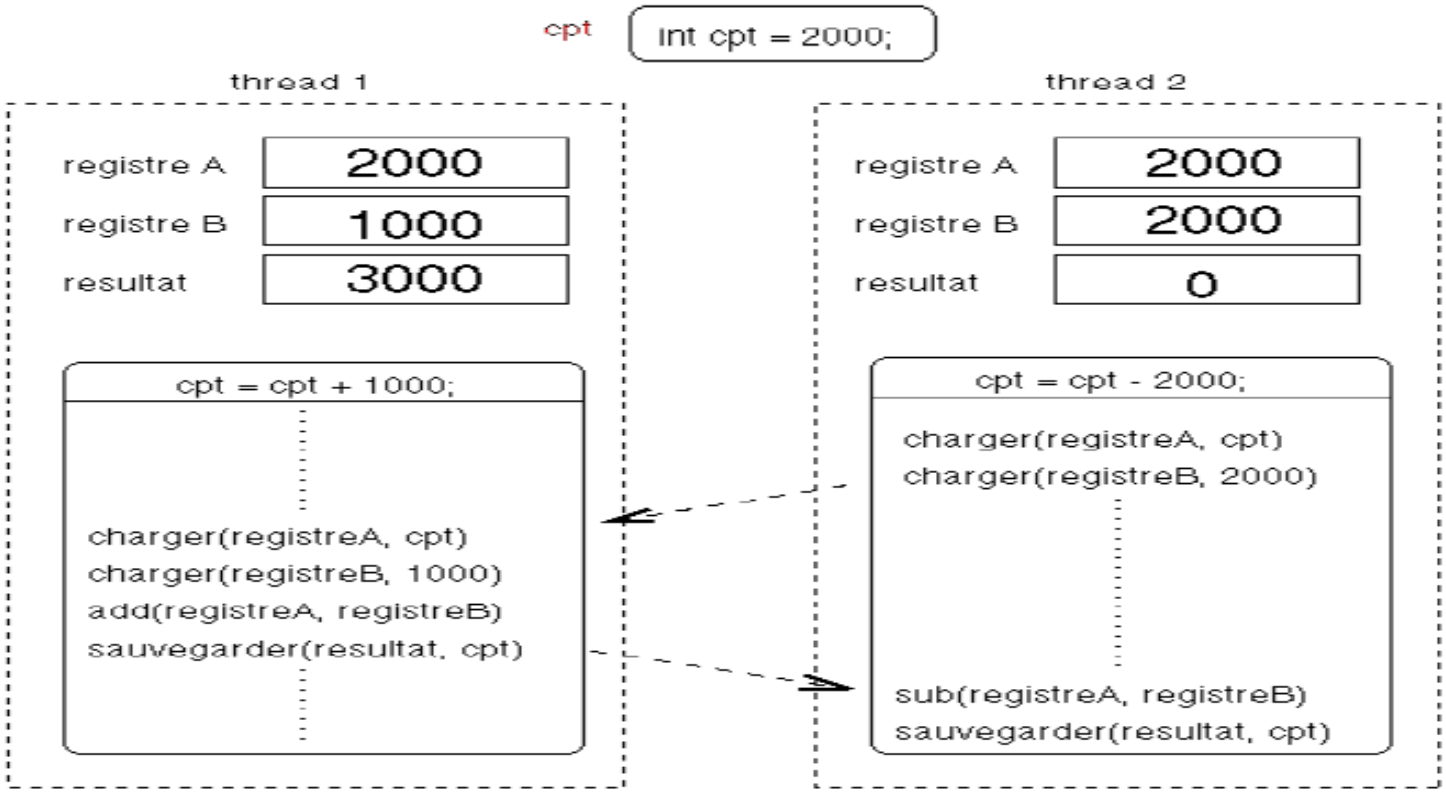
Malheureusement, le partage des ressources sans précaution particulière peut conduire à des résultats imprévisibles. L'état final des données dépend de l'ordonnancement des processus.

# Problématique




La mise a jour concurrente des données peut se dérouler sans problème, où la variable cpt, initialisée à 2000, aura comme valeur finale 1000 (2000 + 1000 - 2000).

# Problématique




Malheureusement, la mise a jour concurrente peut générer des incohérences, où la variable cpt, aura comme valeur finale 0 après l'exécution du même code. On voit que l'exécution de plusieurs threads peut conduire a des résultats incohérents.

La cause de cette incohérences est due a l'utilisation simultanée de la ressource partagée (la zones mémoire de la variable cpt) par plusieurs programmes.

# Chapitre 2:

## Synchronisation

**Université de Bouira**
**Faculté des Sciences et des Sciences appliquées**
**Département d'Informatique**
**Licence SI (Systèmes Informatiques)**
**Semestre 5**

# Systèmes d'exploitation 2

**A. ABBAS**
**abbasakli@gmail.com**

# CONTENU DU COURS

**1. Introduction**

Introduction aux Systèmes d'exploitation

Concepts du processus

Concepts du threads

**2. Synchronisation**

- Problème de l'exclusion mutuelle
- Synchronisation
  - Evénements, Verrous
  - Sémaphores
  - Moniteurs
  - Régions critiques.
  - Expressions de chemins

**3. Communication**

- Partage de variables (modèles : producteur/ consommateur, lecteurs/ rédacteurs)
- Boites aux lettres
- Echange de messages (modèle du client/ serveur)
- Communication dans les langages évolués (CSP, ADA, JAVA..)

# CONTENU DU COURS

**4. Interblocage**

- Modèles
- Prévention
- Evitement
- Détection/ Guérison
- Approche combinée

**5. Etude de cas : système Unix**

- Principes de conception
- Interfaces (programmeur, utilisateur)
- Gestion de processus, de mémoire, des fichiers et des entrées/sorties
- Synchronisation et Communication entre processus.

# Chapitre 1:

# Synchronisation

# Synchronisation

## 1- Introduction

Lorsque plusieurs processus s'exécutent dans un système d'exploitation multitâches, en pseudo-parallèle ou en parallèle et en temps partage, ils partagent des ressources (mémoires, imprimantes, etc.). Cependant, le partage d'objets sans précaution particulière peut conduire a des résultats incohérents. La solution a ce problème s'appelle synchronisation des processus.

# Synchronisation

## 2- Définitions

1. **Ressources critiques:**

a. Les ressources partagée, comme des zones mémoire, cartes d'entrées/sorties, etc., sont dites critiques si elles ne peuvent être utilisées **simultanément** que par un seul processus.

b. On appelle **ressource critique** tout objet : variable, table, chier, périphérique, ... qui peut faire l'objet d'un accès concurrent (ou simultané) par plusieurs processus.

2. **Section critique**

a. Une section critique (SC) est un ensemble d'instruction d'un programme qui peuvent engendrer des résultats imprévisibles (ou incohérents) lorsqu'elles sont exécutées simultanément par des processus différents.

b. Une SC est une suite d'instructions qui opèrent sur un ou plusieurs ressources partagées (critiques) et qui nécessitent une utilisation **exclusive** de ces ressources.

## 2- Définitions

### 3. Exclusion mutuelle

Les problème d'incohérences des résultats, posé par les accès concurrents, montrent que la solution consiste à exécuter les sections critiques en exclusion mutuelle. C'est à dire qu'une section critique (SC) ne peut être entamée (ou exécutée) que si aucune autre SC du même ensemble n'est en exécution.

Le principe général d'une solution garantissant que l'exécution simultanée de plusieurs processus ne conduirait pas à des résultats imprévisibles est : avant d'exécuter une SC, un processus doit s'assurer qu'aucun autre processus n'est en train d'exécuter une SC du même ensemble. Dans le cas contraire, il ne devra pas avancer tant que l'autre processus n'aura pas termine sa SC.

## 2- Définitions

### 3. Exclusion mutuelle

Par exemple, avant d'entrer en SC, le processus doit exécuter un protocole d'entrée. Le but de ce protocole est de vérifier si la SC n'est occupée par aucun autre processus. A la sortie de la SC, le processus doit exécuter un protocole de sortie de la SC. Le but de ce protocole est d'avertir les autres processus en attente que la SC est devenue libre.

Le pseudo-code suivante résume ce fonctionnement :

```
Processus Pi
Début
...<Instructions hors de la section critique>
Protocole d'entrée en SC
|| SC
Protocole de sortie de SC
...<Instructions hors de la section critique>
Fin.
```

# Synchronisation

## 2- Conditions nécessaires pour réaliser une exclusion mutuelle

Pour réaliser une exclusion mutuelle utile on admet que certaines conditions doivent être respectées :

1. **Le déroulement :** Le fait qu'un processus qui ne demande pas à entrer en section critique ne doit pas empêcher un autre processus d'y entrer. En plus, aucun processus suspendu en dehors de sa section critique ne doit bloquer les autres processus.

2. **L'attente infinie :** Si plusieurs processus sont en compétition pour entrer en SC, le choix de l'un d'eux ne doit pas être repoussé indéfiniment. Autrement dit, la solution proposée doit garantir que tout processus n'attend pas indéfiniment.

**3.** Deux processus ne peuvent être en même temps dans leurs sections critiques.

**4.** Tous les processus doivent être égaux vis a vis de l'entrée en SC.

# Synchronisation

## 3- Solutions pour réaliser une exclusion mutuelle

### 1- Masquage des interruptions

Avant d'entrer dans une section critique, le processus masque les interruptions.

Il les restaure (non masque) a la fin de la section critique.

Il ne peut être suspendu alors durant l'exécution de la section critique.

Exemple : Masque des interruptions

```
Local_irq_disable(); /* Les interruptions sont masquées .. */
Section critique ();
Local_irq_enable();
```

**Problèmes:**

1- Si le processus ne restaure pas les interruptions a la sortie de la section critique, ce serait le bug du système.

2- Elle assure l'exclusion mutuelle, si le système est monoprocesseur puisque le masquage des interruptions concerne le processeur qui a demandé ce masquage. Les autres processus exécutés par un autre processeur pourront donc accéder aux ressources partagées.

## 3- Solutions pour réaliser une exclusion mutuelle

### 2- Attente active

Utiliser une variable de verrouillage partagée ***verrou***, unique, initialisée a 0. Pour rentrer en section critique, un processus doit tester la valeur de ***verrou***

- Si elle est égale à 0, le processus modifie la valeur du verrou ←1 et exécute sa section critique. A la fin de la section critique, il remet le verrou a 0.
- Sinon, il attend (par une attente active) que le verrou devienne egal à 0, c.-a-d. :

while(verrou ≠0);

```
while (verrou ≠ 0) do
; // Attente active
end while
verrou ⟵ 1;
Section_critique() ;
verrou ⟵ 0;
```

## 3- Solutions pour réaliser une exclusion mutuelle

### 2- Attente active

**Problèmes**: Cette méthode n'assure pas l'exclusion mutuelle

- Si un processus P1 est suspendu juste après avoir lu la valeur du verrou qui est égal à 0 (par exemple: mov verrou).
- Ensuite, si un autre processus P2 est élu et il teste le verrou qui est toujours égal a 0, met verrou ←1 et entre dans sa section critique.
- Si P2 est suspendu avant de quitter la section critique et que P1 est réactivé et entre dans sa section critique et met le verrou←1

**Résultats:** Les deux processus P1 et P2 sont en même temps en section critique.

Susceptible de consommer du temps en bouclant inutilement.

## 3- Solutions pour réaliser une exclusion mutuelle

### 3 - Solution de Peterson

- Cette solution se base sur deux fonctions **entrer_region** et **quitter_region**.

- Chaque processus doit, avant d'entrer dans sa section critique, appeler la fonction **entrer_region** en lui fournissant en paramètre son numéro.

- Cet appel le fera attendre si nécessaire jusqu'a ce qu'il n'y ait plus de risque.

- A la fin de la section critique, il doit appeler **quitter_region** pour indiquer qu'il quitte sa section critique et pour autoriser l'accès aux autres processus.

Processus $P_i$

```
while (1){
/*attente active*/
entrer_region(i) ;
section_critique_Pi() ;
quitter_region(i);
...}
```

## 3- Solutions pour réaliser une exclusion mutuelle

### 3 - Solution de Peterson

Le pseudo code des deux fonction est donné ci-dessous:

```
void entrer_region(int process)
{
int autre;
if(process==1)
autre = 2;
else
autre=1; //l'autre processus
interesse[process] = TRUE; //indiquer qu'on est intéressé
tour = process; //la course pour entrer se gagne ici
while (tour == process and interesse [autre] == TRUE);
}
```

```
void quitter_region (int process )
{// Processus quitte la région critique
interesse[process] = FALSE;
}
```

**Note:** Les variables *tour* et *interesse* sont globales.

- La généralisation de cette solutions aux cas de plusieurs processus est bien complexe.
- Susceptible de consommer du temps en bouclant inutilement.

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 - Sémaphore

Pour contrôler les accès a un objet partage, Dijkstra (en 1965) a suggéré l'utilisation d'un nouveau type de variables appelées les sémaphores.

**Définition :** Un sémaphore S est un ensemble de deux variables :

1. Une valeur (ou compteur) entier notée value désigne le nombre d'autorisations d'accès à une section critique. Cette valeur est manipulable au moyen des opérations P (ou wait) et V (ou signal);
2. Une file F des processus en attente.

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 - Sémaphore

L'opération P(S) décrémente la valeur du sémaphore S. Puis, si cette dernière est inferieure à 0 alors le processus appelant est mis en attente. Sinon le processus appelant accède a la section critique.

```
P(S)
{
        S.value = S.value - 1;
        if (S.value < 0)
          {
            état(Pro i) = bloqué   ; // Pro i = processus appelant
            mettre Pro i dans S.F   ;
          }
}
```

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 - Sémaphore

L'opération V(S) incrémente la valeur du sémaphore S. Puis si cette valeur est supérieure ou égale à 0 alors l'un des processus bloqués par l'opération P(S) sera choisi et redeviendra prêt.

```
V(S)
{
        S.value = S.value + 1;
        if (S.value ≥ 0)
        {
            < sortir un processus Pro j de S.F >;
            état(Pro j) = prêt
        }
}
```

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 - Sémaphore

L'existence d'un mécanisme de file d'attente (FIFO ou LIFO), permettant de mémoriser le nombre et les demandes d'opération P(S) non satisfaites et de réveiller les processus en attente.

**Remarque 1** : Le test du sémaphore, le changement de sa valeur et la mise en attente éventuelle sont effectués en une seule opération atomique indivisible.

**Remarque 2** : La valeur initiale du champ value d'un sémaphore doit être un nombre non négatif. La valeur initiale d'un sémaphore est le nombre d'unités de ressource.

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 - Sémaphore

Ainsi, on peut proposer un schéma de synchronisation de n processus voulant entrer simultanément en SC, en utilisant les deux opérations P(S) et V(S).

En effet, il suffit de faire partager les ***n*** processus un sémaphore S , initialise a 1, appelé sémaphore d'exclusion mutuelle.

Chaque processus Pi a la structure suivante :

```
Processus Pi
{
        ...
      P(S) ;

        || Sc();

      V(S);
        ...
}
```

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 – Sémaphore - Exemple 1

Considérons deux processus P1 et P2 exécutent respectivement deux instructions

S1 = A+2 B; et S2 = S1+10.

Si nous souhaitons que S2 ne doit s'exécuter qu'après l'exécution de S1, nous pouvons implémenter ce schéma en faisant partager P1 et P2 un sémaphore commun S, initialise a 0 et en insérant les primitives P(S) et V(S) de la façon suivante:

| Processus P1 | Processus P2 |
|---|---|
| S1;<br>V(S); | P(S);<br>S2; |

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 – Sémaphore - Exemple 2: Lecteurs / Rédacteurs

Considérons un objet (une base de donnée par exemple) qui n'est accessible que par deux catégories d'opérations : les lectures et les écritures. Plusieurs lectures (consultations) peuvent avoir lieu simultanément ; par contre les écritures (mises a jour) doivent se faire en exclusion mutuelle.

On appellera *lecteur* un processus faisant des lectures et *rédacteur* un processus faisant des écritures.

Il s'agit donc de réaliser la synchronisation entre lecteurs et rédacteurs en respectant les contraintes suivantes :

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 – Sémaphore - Exemple 2: Lecteurs / Rédacteurs

**Exclusion mutuelle entre lecteurs et rédacteurs** : si un lecteur demande a lire et qu'il y a une écriture en cours , la demande est mise en attente. De même que si un rédacteur demande à écrire et qu'il y a au moins une lecture en cours , la demande est mise en attente.

**Exclusion mutuelle entre rédacteurs :** si un rédacteur demande a écrire et qu'il y a une écriture en cours , la demande est mise en attente.

Pour satisfaire ces contraintes ci-dessus , on peut procéder comme suit :

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 – Sémaphore - Exemple 2: Lecteurs / Rédacteurs

```
var S1,S2 : sémaphore init 1,1 ;
nl : entier init 0 ;
```

| Processus Lecteur | Processus Rédacteur |
|---|---|
| Début Début | Début |
| . | |
| . | |
| . | |
| P(S2) | P(S1) |
| nl := nl + 1 | Ecriture |
| si nl=1 alors P(S1) finsi | V(S1) |
| V(S2) | . |
| Lecture | . |
| P(S2) | . |
| nl := nl – 1 | Fin |
| si nl=0 alors V(S1) finsi | |
| V(S2) | |
| . | |
| . | |
| . | |
| Fin | |

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 – Sémaphore - Services Posix sur les semaphores

Le système d'exploitation Linux permet de créer et d'utiliser les sémaphores définis par le standard Posix.

Les services Posix de manipulation des sémaphores se trouvent dans la librairie <semaphore.h>.

Le type sémaphore est désigné par le type sem_t.

**Initialisation d'un sémaphore**

**int sem_init(sem_t *sem, int pshared, unsigned int valeur)**

- ***sem*** **:** est un pointeur sur le sémaphore à initialiser;
- **value :** est la valeur initiale du sémaphore ;
- **pshared :** indique si le sémaphore est local au processus (pshared=0) ou partage entre le père et le fils pshared ≠0.

# Synchronisation

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 – Sémaphore - Services Posix sur les sémaphores

**int sem_wait(sem_t *sem) :** est équivalente à l'opération P (S) et retourne toujours 0.

**int sem_post(sem t *sem):** est équivalente a l'opération V(S) : retourne 0 en cas de succès ou -1 autrement.

**int sem trywait(sem_t *sem) :** décrémente la valeur du sémaphore sem si sa valeur est supérieure à 0 ; sinon elle retourne une erreur. C'est une opération atomique.

**int sem_getvalue (sem_t * sem, int * sval):** récupérer la valeur d'un sémaphore : il retourne toujours 0.

**int sem destroy (sem t* sem):** détruire un sémaphore. Retourne -1 s'il y a encore des waits.

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 – Sémaphore - Exemple 3: le problème du producteur-consommateur

Le problème de Producteur-Consommateur est un problème de synchronisation classique qui permet de représenter une classe de situations où un processus, appelé **Producteur**, délivre des messages (informations) à un processus **Consommateur** dans un tampon (par exemple, un programme qui envoie des données sur le spool de l'imprimante).

Le **Producteur** produit un message dans la *ZoneP*, puis le dépose dans le buffer. Le **Consommateur** prélève un message du Buffer et le place dans la ZoneC où il peut le consommer.

On considérera que le buffer est de N cases, de 0 à N-1, et organisé de façon circulaire. Le Producteur dépose les messages par un bout du buffer alors que le consommateur les consomme au fur et à mesure par l'autre bout.

# Synchronisation

## 3- Solutions pour réaliser une exclusion mutuelle

**4 – Sémaphore - Exemple 3:** le problème du producteur-consommateur

n producteurs produisent des messages et les déposent dans un tampon de taille illimitée. n consommateurs peuvent récupérer les messages déposés dans le tampon.

```
Semaphore Mutex = 1 ;
Message tampon[ ];
Producteur ( ){
Message m ;
Tantque Vrai faire
        m = creermessage() ;
        Mutex.P() ;
                EcritureTampon(m);
        Mutex.V() ;
FinTantque
}
```

```
Consommateur( ){
Message m ;
Tantque Vrai faire
        Mutex.P() ;
                m = LectureTampon();
        Mutex.V() ;
Fin Tantque
}
```

Adaptez cette solution pour que l'échange des message soit synchronisé (consommateurs bloqués si le tampon est vide).

## 3- Solutions pour réaliser une exclusion mutuelle

**4 – Sémaphore - Exemple 3:** le problème du producteur-consommateur

```
Semaphore Mutex = 1, Plein = 0 ;
Message tampon[];
Producteur ( ){
Message m ;
Tantque Vrai faire
        m = creermessage() ;
        Mutex.P() ;
                EcritureTampon(m);
        Mutex.V() ;
        Plein.V();
FinTantque
}
```

```
Consommateur( )
{
Message m ;
Tantque Vrai faire
        Plein.P();
        Mutex.P() ;
                m = LectureTampon();
        Mutex.V() ;
Fin Tantque
}
```

## 3- Solutions pour réaliser une exclusion mutuelle

### 4 – Sémaphore -Critiques

Les programmes utilisant les sémaphores sont généralement difficile à réaliser et à "débuguer" car les erreurs, (oublie de V ou utiliser P à la place de V), sont dues à des conditions de concurrence, des inter-blocages ou d'autres formes de comportement imprévisibles et/ou difficiles à reproduire.

D'autre part, les sémaphores sont des objets globaux et doivent être connus de tous les participants. De plus, la synchronisation par les sémaphores nécessite l'étude de tout un programme concurrent pour comprendre l'aspect synchronisation qu'il renferme.

- Pour palier à ces points faibles, le concept de ***moniteur*** a vu le jour
- Ce concept a été implémenté dans des langages de programmation concurrente : C# ainsi que Java.

## 3- Solutions pour réaliser une exclusion mutuelle

### 5 – Les Moniteurs -Définition

L'idée des moniteurs est de regrouper dans un module spécial, appelé moniteur, toutes les sections critiques d'un même problème.

Un moniteur est un ensemble de **procédures**, de **variables** et de **structures de données**.

Les processus peuvent appeler (ou utiliser) les procédures du moniteur mais ils ne peuvent pas accéder aux variables et aux structures de données interne du moniteur à partir des procédures externes.

## 3- Solutions pour réaliser une exclusion mutuelle

### 5 – Les Moniteurs -Exclusion mutuelle par des moniteurs

Pour assurer l'accès aux ressources critiques en exclusion mutuelle, il suffit qu'il y ait à tout instant pas plus d'un processus actif dans le moniteur, c'est-a-dire que les procédures du moniteur ne peuvent être exécutées que par un seul processus a la fois. C'est le compilateur qui effectue de cette tâche (assurer l'accès aux ressources critiques).

Pour cela, le compilateur rajoute, au début de chaque procédure du moniteur un code qui réalise ce qui suit:

- S'il y a un processus P1 actif dans le moniteur, alors le processus P2 appelant est suspendu jusqu'a ce que le processus actif dans le moniteur se bloque en exécutant un **wait** sur une variable conditionnelle (wait(c)) ou quitte le moniteur.

- Le processus bloqué est réveillé par un autre processus en lui envoyant un signal sur la variable conditionnelle (signal(c)).

## 3- Solutions pour réaliser une exclusion mutuelle

### 5 – Les Moniteurs

Une variable de condition (des moniteurs) est une condition manipulée au moyen de deux opérations wait et signal :

**wait(x) :**

- suspend l'exécution du processus (thread) appelant (le met en attente de x);
- autorise un autre processus en attente du moniteur a y entrer.

**signal(x)** : débloqué un processus en attente de la condition x.

## 3- Solutions pour réaliser une exclusion mutuelle

### 5 – Les Moniteurs -Implémentation des moniteurs

L'utilisation de moniteurs est plus simple que les sémaphores puisque le programmeur n'a pas a se préoccuper de contrôler les accès aux sections critiques.

Mais, malheureusement, a l'exception de JAVA (avec le mot clé synchronized) et C#, la majorité des compilateurs utilisés actuellement ne supportent pas les moniteurs. Par contre, on peut simuler les moniteur avec d'autres langages à partir des mutex comme suit:

- Le moniteur contient un sémaphore binaire (par exemple: mutex)
- Toutes les procédures commencent par l'acquisition du mutex et finissent par sa libération.

## 3- Solutions pour réaliser une exclusion mutuelle

### 5 – **Les Moniteurs** –Exemple 1 : Producteur/consommateur

- Les sections critiques du problème du producteur et du consommateur sont les opérations de dépôt et de retrait dans le tampon partagé.
- Le consommateur est actif dans le moniteur et le tampon est vide => il devrait se mettre en attente et laisser place au producteur.
- Le producteur est actif dans le moniteur et le tampon est plein => il devrait se mettre en attente et laisser place au consommateur.

| Processus producteur | Processus consommateur |
|---|---|
| {<br>int p ;<br>while (1){<br>Produire (p) ;<br>ProducteurConsommateur.mettre (p);<br>}<br>} | {<br>int c ;<br>while (1){<br>ProducteurConsommateur.retirer(c);<br>Consommer (c);<br>}<br>} |

## 3- Solutions pour réaliser une exclusion mutuelle

### 5 – Les Moniteurs –Exemple 1 : Producteur/consommateur

```
Moniteur ProducteurConsommateur (){
bool nplein, nvide ; //variable de condition pour non
plein et non vide
int compteur =0, ic=0, ip=0, N ;

void mettre (int objet) // section critique pour le dépôt
{
  if (compteur==N) wait(nplein) ; //attendre
                jusqu'a ce que le tampon soit non plein

                tampon[ip] = objet ;
                ip = (ip+1)%N ;
                compteur++ ;

                // si le tampon était vide, envoyer un
                signal pour réveiller le consommateur.

        if (compteur==1) signal(nvide) ;
}

void retirer (int* objet) //section critique
pour le retrait
{               if (compteur ==0) wait(nvide) ;
                objet = tampon[ic] ;
                ic = (ic+1)%N ;
                compteur - ;

// si le tampon était plein, envoyer un
signal pour réveiller le producteur.

if(compteur==N-1) signal(nplein) ;
}
}
```

## 3- Solutions pour réaliser une exclusion mutuelle

### 5 – Les Moniteurs - Règles d'utilisation des moniteurs

An d'éviter que tous les processus réveillés se retrouvent au même temps dans le moniteur, différentes règles ont été établies pour définir ce qui se passe a l'issue d'un signal.

1. **Règle de Hoare** : ne laisser entrer dans le moniteur que le processus qui a été suspendu le moins longtemps;

**2. Regle de Brinch Hansen** : exiger du processus qui fait **Signal** de sortir immédiatement du moniteur. Il laisse ainsi la place a tous ceux qui étaient en attente. L'ordonnanceur choisira un parmi ceux ci.

**3. 2ème règle de Brinch Hansen :** si un Signal est réalise sur une variable conditionnelle et qu'aucun processus ne l'attend, alors ce signal est perdu.

## 3- Solutions pour réaliser une exclusion mutuelle

### 5 – Les Moniteurs - Réalisation sous C/POSIX

Pour garantir qu'une seule procédure du moniteur soit activée à instant donne, il suffit de protéger l'exécution de toutes les procédures par le même sémaphore d'exclusion mutuelle.

**Exemple Moniteur: probleme Producteur/consommateur**

```
pthread_mutex_t mutex ;
pthread_cond_t est_vide, est_plein;
char *buffer;
void *mettre (char msg) {// section critique pour le dépôt
pthread_mutex_lock ( & mutex);
        while (buffer ! = NULL)
         pthread_cond _wait ( & est_vide, & mutex);
        // buer = NULL
        buffer = strdup(msg);
        Pthread_cond_signal ( & est_plein);
pthread_mutex _unlock ( & mutex);
}
```

## 3- Solutions pour réaliser une exclusion mutuelle

### 5 – **Les Moniteurs -** Réalisation sous C/POSIX

```
char *retirer () {//section critique pour le retrait
char *result;
    pthread_mutex_lock ( & mutex);
while (buffer == NULL)
            pthread_cond _wait ( & est_plein, & mutex);
    result = buer;
    buffer = NULL;
    pthread_cond_signal ( & est_vide);
    pthread_mutex_unlock ( & mutex);
    return result;
}
```

```
void main (void){
        pthread_t ta; pthread_t tb;
        pthread_mutex_init (&mutex, NULL);
        pthread_cond_init (& est_vide, NULL);
        Pthread_cond_init (& est_plein, NULL);
        buffer = NULL;
}
```

## 3- Solutions pour réaliser une exclusion mutuelle

**5 – Les Moniteurs -** Résumé des types et des fonctions utilisés

| Types et fonctions | Description |
|---|---|
| pthread_t | Contexte de tâche |
| pthread_create() | Création d'une tâche |
| pthread_join() | Attente de fin de tâche |
| pthread_mutex_t | Sémaphore d'exclusion mutuelle |
| pthread_mutex_lock() | Début de zone critique |
| pthread_mutex_unlock() | Fin de zone critique |
| pthread_cond_t | Condition |
| pthread_cond_wait() | Mise en attente |
| pthread_cond_signal() | Déclenchement |

## 3- Solutions pour réaliser une exclusion mutuelle

### 5 – **Les Moniteurs -** Réalisation avec JAVA

Le concept de moniteur est implémente dans la MVJ de la manière suivante :

- Les données du moniteur doivent être déclarées avec le mot clé **privates** pour que seules les méthodes du moniteur accèdent à ces données,

- Les méthodes (ou procédures d'entrée) du moniteur doivent être déclarées avec le mot clé **synchronized** pour qu'elles puissent s'exécuter en exclusion mutuelle,

- La classe **Object** fournit les méthodes *wait()* et *notify()* pour la synchronisation des threads.

**Université de Bouira**
**Faculté des Sciences et Sciences Appliquées**
**Département d'Informatique**
**Semestre 5 (3ème année SI)**

# Partie 3

## Communication Interprocessus

**Par : ABBAS**

**Disponible sur :**

**https://sites.google.com/a/esi.dz/a-abbas**

# Introduction :

La communication interprocessus consiste à transférer des données entre les processus.

**Exemple:** un navigateur Internet peut demander une page à un serveur, qui envoie alors les données HTML.

La communication interprocessus peut se faire de différentes manières :

1. les segments de mémoire partagée;
2. les files de messages ;
3. les sémaphores,
4. les signaux
5. les tubes
6. Socket : communication (bidirectionnel) entre processus à travers un réseau

# Introduction :

La communication interprocessus consiste à transférer des données entre les processus.

**Exemple:** un navigateur Internet peut demander une page à un serveur, qui envoie alors les données HTML.

**Deux familles**

- Les outils gérés avec le SGF :
    1. les tubes
    2. les sockets
- Les outils IPC gérés dans des tables du systèmes et repérés par une clef :
    1. les files de messages
    2. **les segments de mémoire partagée,**
    3. les sémaphores

# Partage de mémoire

- Dans le cas de **processus légers** (*thread*), l'espace mémoire des processus est partagé, la mémoire peut donc être utilisée directement.

- Dans le cas de **processus lourds** (*process*), les espaces mémoires des processus ne sont pas partagés. il faut passer par un moyen externe (IPC = Inter Process Communication) pour communiquer.

- Ce type de communication pose le problème des sections critiques

# Partage de mémoire

**Construction du segment de mémoire partagée :**

int shmget (key_t  key,  size_t size,  int shmflg);

**Identification externe du segment**

**Taille de la mémoire en octet**

**Constantes IPC_CREAT, IPC_EXCL et droits d'accès**

Cet appel système retourne l'identificateur du segment créer et −1 en cas d'erreur ;

**Création d'une clé:**

Key_t ftok (char * pathname, int proj) ;

pour créer une clé relativement unique à partir du nom d'un fichier et des 8 premier bits de projet

**Exemple:**

key_t key= ftok("/tmp/fichier", 3);

# Partage de mémoire

**Attachement : shmat()** permet à un processus de s'attacher ce segment

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

**Identification externe de la mémoire**

Adresse de la région (0 ou valeur)

shmflg précise l'opération

**Détachement : shmdt()** permet à un processus de se détacher du segment :

```
int shmdt(const void *shmaddr);
```

**Contrôle: shmctl()** permet d'intervenir et de contrôler segment

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

# Segment de mémoire partagée

**processus créateur du segment et écrivain**

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE          256

main()
{
int shmid;
char *mem ;

/* création du segment de mémoire partagée avec la clé CLE */
shmid=shmget((key_t)CLE,1000,0750 |IPC_CREAT | IPC_EXCL);

/* attachement */
if((mem=shmat(shmid,NULL,0))==(char *)-1)
{
   perror("shmat");
   exit(2);
}

/* écriture dans le segment */
strcpy(mem,"voici une écriture dans le segment");
exit(0);
}
```

# Segment de mémoire partagée

**processus lecteur et destructeur du segment.**

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE          256

main()
{
/* récupération du segment de mémoire */
shmid=shmget((key_t)CLE,0,0);

/* attachement */
mem=shmat(shmid,NULL,0);

/* lecture dans le segment */
printf("lu: %s\n",mem);

/* détachement du processus */
shmdt(mem);

/* destruction du segment */
shmctl(shmid,IPC_RMID,NULL);

exit(0)
}
```

# Les files de messages

- **Création de file de message**

***int msgget(key_t key, int flg)*** :

Permet de créer une nouvelle file de message ou de récupérer l'identificateur d'une file déjà existante à partir d'une clé.

flg combinaison des constantes IPC_CREAT, IPC_EXCL et de droits d'accès

- **modifier ou supprimer une file**

**msgctl(...)**

- **Structure des messages:**

```
struct msgbuf {
        long mtype; // type de message ( >0 )
        char mtext[1]; // contenu du message
        …
};
```

# Les files de messages

- **Envoi du message**

***int msgsnd(int id, struct msgbuf * msgp, size_t sz, int flg)***

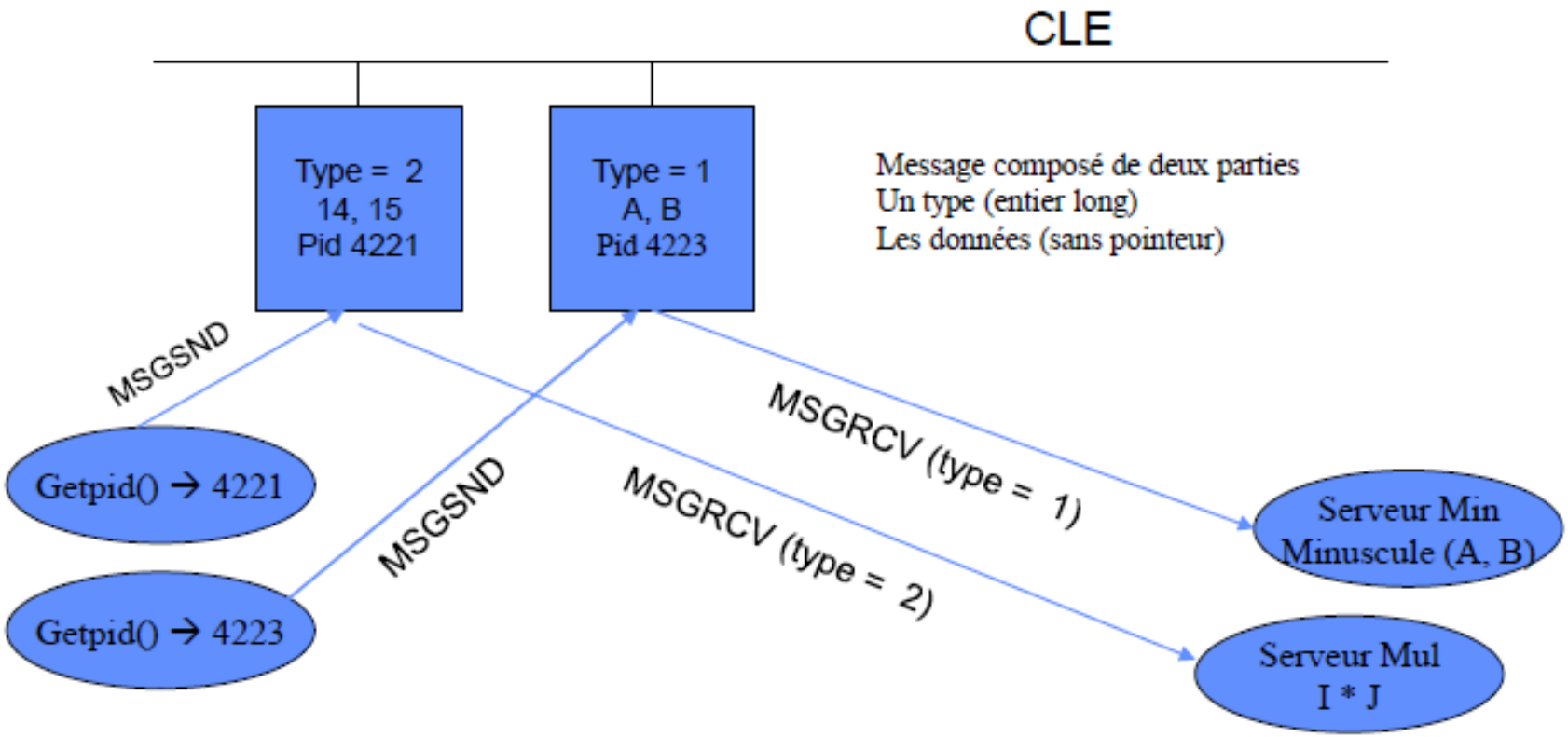
pour envoyer un message constitué d'un type (un entier long positif) et d'une suite d'octets de longueur **sz** sur une file

- **Récupération de messages**

**ssize msgrcv(int id, struct msgbuf * msgp, size_t sz, long typ, int flg)**

pour récupérer le premier message d'un type donné (ou de tous les types si typ = 0)

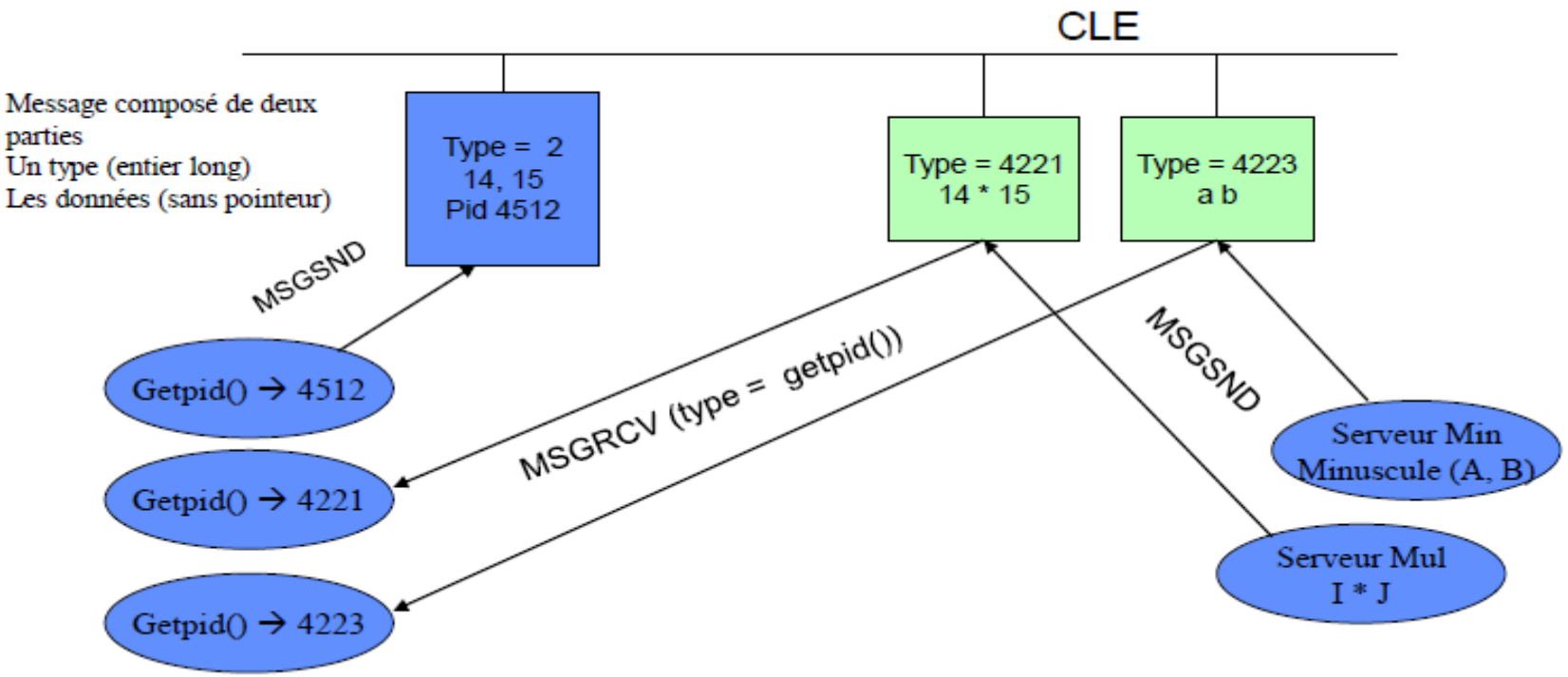
# Les files de messages – Exemple illustratif




Le type dans le message permet de désigner le destinataire.

La primitive MSGRCV spécifie quel type de message prélever.
**Côté client** : type à utiliser pour l'identifier de façon unique : son pid
**Côté serveur** : de petits numéros qui ne peuvent pas correspondre à des pid de clients.

# Les files de messages – Exemple illustratif




Le type dans le message permet de désigner le destinataire.

La primitive MSGRCV spécifie quel type de message prélever.
**Côté client** : type à utiliser pour l'identifier de façon unique : son pid
**Côté serveur** : de petits numéros qui ne peuvent pas correspondre à des pid de clients.

# Les files de messages – Exemple d'un producteur de messages

```
#include ...
struct mymsgbuf {
    long  mtype;        /* type du message ( > 0 ) */
    char  mtext[100];   /* contenu du message      */
} message;
#define CLE 100012

int main(int argc, char* argv[]) {
    int i;
    int id = msgget( CLE, IPC_CREAT | 0777);
    if (id == -1) { perror("Création de la file : ");  exit(-1); }
    for(i = 1; i < argc ; i++) {
        printf("[%d] envoie du message %d : <%s>\n",
                                          getpid(), i, argv[i]);
        message.mtype = i;
        strncpy(message.mtext, argv[i], 99);
        msgsnd(id, (struct msgbuf *) &message, 100, 0);
        printf("[%d] fin de l'envoie du message %d : <%s>\n",
                                          getpid(), i, argv[i]);
        sleep(2);
    }
}
```

# Les files de messages – Exemple d'un récupérateur msg Type 1

```
#include ...
struct mymsgbuf {
    long  mtype;        /* type du message ( > 0 ) */
    char  mtext[100];   /* contenu du message      */
} message;
#define CLE 100012

int main(int argc, char* argv[]) {
    int i;
    int id = msgget( CLE, IPC_CREAT | 0777);
    if (id == -1) { perror("Création de la file : ");  exit(-1); }
    printf("[%d] réception des messages de type 1\n", getpid());
    while(1) {
        int i = msgrcv(id, (struct msgbuf *) &message, 100, 1, 0);
        if (i == -1)
            perror("Erreur de msgrcv :");
        else {
            printf("[%d] réception du message %d : <%s>\n",
                  getpid(), message.mtype, message.mtext);
        }
    }
}
```

# Les files de messages – Exemple d'un récupérateur de tous msg

```
#include ...
struct mymsgbuf {
    long  mtype;       /* type du message ( > 0 ) */
    char  mtext[100];  /* contenu du message      */
} message;
#define CLE 100012

int main(int argc, char* argv[]) {
    int i;
    int id = msgget( CLE, IPC_CREAT | 0777);
    if (id == -1) { perror("Création de la file : ");  exit(-1); }
    printf("[%d] réception de tous les messages\n", getpid());
    while(1) {
        int i = msgrcv(id, (struct msgbuf *) &message, 100, 0, 0);
        if (i == -1)
            perror("Erreur de msgrcv :");
        else {
            printf("[%d] : réception du message %d : <%s>\n",
                        getpid(), message.mtype, message.mtext);
        }
```

# Communication par signaux

Dans le cas du système Unix/Linux, un processus utilisateur peut également envoyer un signal à un autre processus du même propriétaire.

Un signal, qui est une interruption logicielle, sont des événements externes qui changent le déroulement d'un programme, et qui a pour but de l'informer de l'arrivée d'un événement. **Il ne véhicule pas d'information**.

Ce mécanisme de communication permet à un processus de réagir à un événement sans être obligé de tester en permanence l'arrivée.

# Communication par signaux

## Envoi d'un signal

Un signal peut être envoyé :

1. lors de la constations d'une anomalie matérielle;
2. suite à la frappe d'une combinaison de touches;
3. par un autre processus utilisant la commande kill ou l'appel système kill.

Le système d'exploitation associe à chaque signal un traitement par défaut (gestionnaire par défaut du signal) :

- **abort :** génération d'un fichier core et arrêt du processus;
- **exit:** terminaison du processus sans génération d'un fichier core;
- **ignore**: le signal est ignoré;
- **stop**: suspension du processus;
- **continue** : reprendre l'exécution si le processus est suspendu sinon le signal est ignoré.

Un processus peut changer son comportement par défaut lors de la réception d'un signal en déroutant le signal, c'est-à-dire en indiquant la fonction à exécuter lors de la réception du signal.

# Communication par signaux

Le système d'exploitation gère un ensemble de signaux. Chaque signal
a un nom, un numéro, un gestionnaire (handler) et est associé à un type d'événement.

**Signaux Linux**

| Numéro | Nom | Action | Description |
|---|---|---|---|
| 1 | SIGHUP | exit | Terminaison de la liason avec le terminal |
| 2 | SIGINT | exit | Interruption en provenance du terminal (<ctrl C>) |
| 3 | SIGQUIT | core | Quitter en provenance du terminal (<ctrl \>) |
| 4 | SIGILL | core | Instruction illégale |
| 5 | SIGTRAP | core | Trace ou trap de point de débogage |
| 6 | SIGIOT | | |
| 7 | SIGBUS | core | Erreur de bus (adresse mal alignée) |
| 8 | SIGFPE | core | Exception d'arithmétique point flottant |
| 9 | SIGKILL | exit | Élimination (pas d'attrapage et masquage) |
| 10 | SIGUSR1 | | Signal1 défini par l'usager |
| 11 | SIGSEGV | core | Faute de segmentation (référence mémoire) |
| 12 | SIGUSR2 | exit | Signal défini par l'usager |
| 13 | SIGPIPE | exit | Tube sans processus récepteur |
| 14 | SIGALRM | exit | Déclenchement de l'alarme (setitimer, alarm) |
| 15 | SIGTERM | exit | Terminaison envoyée par la commande kill |
| 17 | SIGCHLD | ignore | Chargement d'état du processus enfant |
| 18 | SIGCONT | | Reprise |
| 19 | SIGSTOP | stop | Arrêt (pas d'attrapage et masquage) |
| 20 | SIGTSTP | stop | Arrêt (<ctrl Z>) |
| 21 | SIGTTIN | stop | Lecture au terminal par processus en arrière plan (arrêt) |

# Communication par signaux

Le système d'exploitation gère un ensemble de signaux. Chaque signal
a un nom, un numéro, un gestionnaire (handler) et est associé à un type d'événement.

**Signaux Unix**

| Numéro | Nom | Action | Description |
|---|---|---|---|
| 1 | SIGHUP | exit | Terminaison de la liason avec le terminal |
| 2 | SIGINT | exit | Interruption en provenance du terminal (<ctrl C>) |
| 3 | SIGQUIT | core | Quitter en provenance du terminal (<ctrl \>) |
| 4 | SIGILL | core | Instruction illégale |
| 5 | SIGTRAP | core | Trace ou trap de point de débogage |
| 6 | SIGABRT | core | Annulation par un appel à abort (...) |
| 7 | SIGEMT | core | Trap d'émulation |
| 8 | SIGFPE | core | Exception d'arithmétique point flottant |
| 9 | SIGKILL | exit | Élimination (pas d'attrapage et masquage) |
| 10 | SIGBUS | core | Erreur de bus (adresse mal alignée) |
| 11 | SIGSEGV | core | Faute de segmentation (référence mémoire) |
| 12 | SIGSYS | core | Appel système erroné |
| 13 | SIGPIPE | exit | Tube sans processus récepteur |
| 14 | SIGALRM | exit | Déclenchement de l'alarme (setitimer, alarm) |
| 15 | SIGTERM | exit | Terminaison envoyée par la commande kill |
| 16 | SIGUSR1 | exit | Signal défini par l'usager |
| 17 | SIGUSR2 | exit | Signal défini par l'usager |
| 18 | SIGCHLD | ignore | Chargement d'état du processus enfant |
| 19 | SIGPWR | ignore | Panne d'alimentation ou réinitialisation |

# Communication par signaux

Dans le cas du système Unix/Linux, un processus utilisateur peut également envoyer un signal à un autre processus du même propriétaire.

## kill()

L'envoi d'un signal peut se faire avec l'appel système **kill()** :

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int signal);
```

envoie un signal:

– pid > 0 : au processus désigné
– pid = 0 : à tous les processus du même groupe que le processus appelant
– pid = -1 : à tous les processus, sauf init
– pid < -1 : aux processus du groupe -pid

Exemple: **kill(415, SIGCONT);**

# Communication par signaux

## Interception d'un signal

on peut intercepter un signal (sauf SIGKILL et SIGSTOP) en installant un gestionnaire (signal handler)

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal ( int signum, sighandler_t handler );
```

– Le premier paramètre est le numéro ou le nom du signal à capturer

– Le second est la fonction gestionnaire à exécuter à l'arrivée du signal.

– Signal retourne le gestionnaire précédent ou SIG_ERR en cas d'erreur.

Exemple : **signal(SIGTERM, SIG_IGN)**

# Communication par signaux

## Pause:

**int pause(void);**

Endort le processus jusqu'à ce qu'un signal ait été reçu et traité par son gestionnaire, s'il existe

- moins consommateur de CPU que l'attente active quand on attend des signaux

```
while (1);
```



```
while (1) {
    pause();
}
```

## Sleep :

L’appel système sleep(v) suspend l’appelant jusqu’au prochain signal ou l’expiration du délai (v secondes).

**#include <unistd.h>**

**int sleep(unsigned int seconds);**

# Les tubes de communication

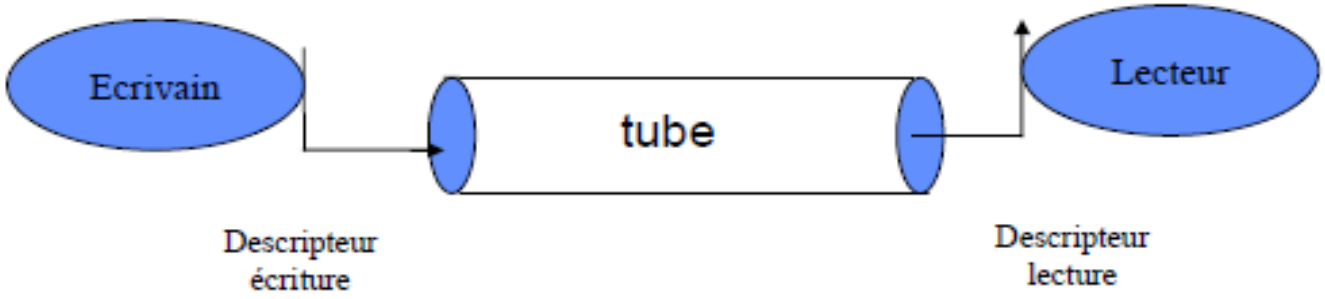
✓ Les tubes de communication permettent à deux ou plusieurs processus s'exécutant sur une même machine d'échanger des informations.

✓ On distingue deux types de tubes :

● Les tubes anonymes (unamed pipe), **entre processus d'une même famille**

● Les tubes nommés (named pipe) qui ont une existence dans le système de fichiers (un chemin d'accès). Il est utilisé par **n'importe quel processus**

✓ Un tube est unidirectionnel: un côté du tube sert à l'écriture, le second pour la lecture



# Les tubes de communication

**Création d'un tube sans nom (Anonymes)**

Fichier sans nom (temporaire), seulement accessible par les deux descripteurs

Seuls les processus ayant connaissance par héritage des descripteurs peuvent utiliser le tube ⇒ processus créateur et ses descendants

Un tube de communication permet de mémoriser des informations et se comporte comme une file FIFO.

Il est caractérisé par deux descripteurs de fichiers (lecture et écriture) et sa taille limitée (PIPE_BUF) est approximativement égale à 4KO.

La lecture dans un tube est destructrice

Lorsque tous les descripteurs du tube sont fermés, le tube est détruit.

```
#include <unistd.h>
int pipe (int p[2]); //  p tableau de deux entiers
```

# Les tubes de communication

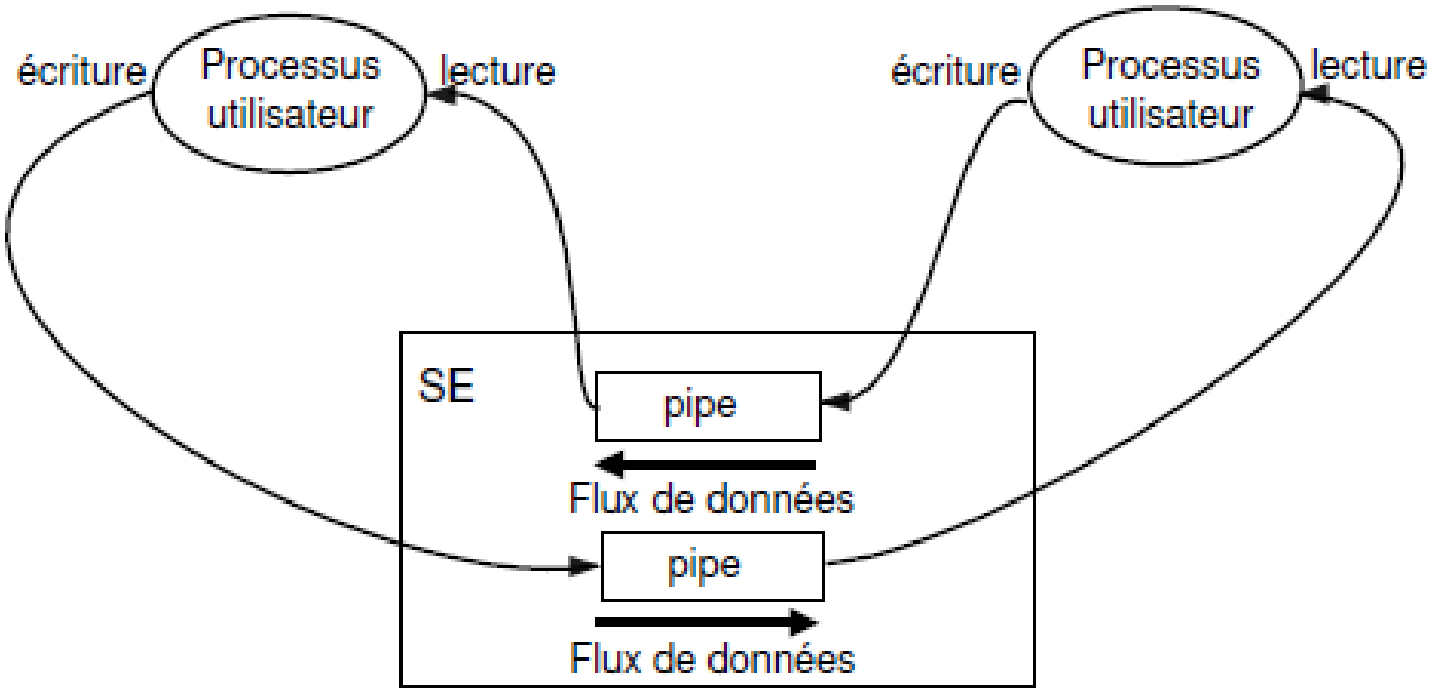
**Exemple d'un tube sans nom (Anonyme)**

```
# include <sys/types . h> // types
# include <unistd . h> // fork , pipe , read , write , close
# include <stdio . h>
main () {
   int p[2], pid_t retour;
   char chaine[7];
   pipe(p);
   retour = fork();
if (retour == 0) {
/* le fils écrit dans le tube */
close (p[0]); /* pas de lecture sur le tube */
write (p[1], "bonjour", 7);
close (p[1]); fermeture du descripteur d'écriture
exit(0); }
else
{ /* le père lit le tube */
/* pas d'écriture sur le tube */
close (p[1]);
read(p[0], chaine, 7);
 close (p[0]);
     wait;
}
```

# Les tubes de communication

**Communication bidirectionnelle avec des tubes sans nom (Anonymes)**

La communication bidirectionnelle entre processus est possible en utilisant deux tubes



# Les tubes de communication

**Les tubes nommés (**supportés par linux**)**

- Ils ont chacun un nom qui existe dans le système de chiers (une entrée dans la Table des fichiers).
- Ils sont considérés comme des fichiers spéciaux.
- Ils peuvent être utilisés par des processus indépendants, à condition qu'ils s'exécutent sur une même machine.
- Ils existeront jusqu'à ce qu'ils soient supprimés explicitement.
- Ils sont de capacité plus grande, d'environ 40 Ko.

# Les tubes de communication

**Ouverture des tubes nommés**

✓L'ouverture d'un tube nommé s'effectue en utilisant la primitive open()

**✓int open (const char *nom, int mode_ouverture);**

✓La primitive renvoie un seul descripteur correspond au mode d'ouverture spécifié (lecture seule, écriture seule, lecture/écriture).

✓La primitive open() appliquée au tube nommé est bloquante.

➢demande d'ouverture en lecture est bloquante tant qu'il n'existe pas d'écrivain sur le tube.

➢demande d'ouverture en écriture est bloquante tant qu'il n'existe pas de lecteur sur le tube.

Ce mécanisme permet à deux processus de se synchroniser et d'établir un rendez-vous en un point particulier de leur exécution.

# Les tubes de communication

**Ouverture des tubes nommés**

✓L'ouverture d'un tube nommé s'effectue en utilisant la primitive open()

**✓int open (const char *nom, int mode_ouverture);**

✓La primitive renvoie un seul descripteur correspond au mode d'ouverture spécifié (lecture seule, écriture seule, lecture/écriture).

✓La primitive open() appliquée au tube nommé est bloquante.

➢demande d'ouverture en lecture est bloquante tant qu'il n'existe pas d'écrivain sur le tube.

➢demande d'ouverture en écriture est bloquante tant qu'il n'existe pas de lecteur sur le tube.

Ce mécanisme permet à deux processus de se synchroniser et d'établir un rendez-vous en un point particulier de leur exécution.

# Les tubes de communication

**Exemple de tubes nommé**

```
/* Processus lecteur sur le tube nommé */
 #include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
main ()
{
        char zone[11];
        int tub;
        tub = open ("fictub", O_RDONLY); /* ouverture du tube */

        read (tub, zone, 10); /* lecture dans le tube */

        printf ("lecteur du tube fictub: j'ai lu %s", zone);

        close(tub); /* fermeture du tube */
}
```

# Les tubes de communication

**Exemple de tubes nommé**

```
/* Processus écrivain sur le tube nommé */
 #include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
main ()
{
        mode_t mode;
        int tub;
        mode = S_IRUSR | S_IWUSR;
        mkfifo ("fictub", mode); /*création du tube nommé */

        tub = open ("fictub", O_WRONLY); /* ouverture du tube */

        write (tub, "0123456789", 10); /* écriture dans le tube */

        close(tub); /* fermeture du tube */
}
```

# Communication Client-Serveur

- Sockets
- Remote Procedure Calls ?
- Remote Method Invocation (Java) ?

# Sockets

- Une socket est définie comme un *endpoint de communication*
- Concatenation d'une adresse IP et d'un numéro de port
- La socket **161.25.19.8:1625** désigne le port **1625** sur l'hôte **161.25.19.8**
- Une communication se fait entre une paire de sockets

# Communication Socket

Un *socket* est un dispositif de communication **bidirectionnel** pouvant être utilisé pour communiquer avec un autre processus sur la **même machine** ou avec un processus s'exécutant sur d'**autres machines**.

Lorsque vous créez un socket, vous devez indiquer trois paramètres:
le style de communication, l'espace de nommage et le protocole.

**Le style de communication** détermine comment sont gérés les paquets et comment ils sont envoyés de l'émetteur vers le destinataire.

**L'espace de nommage** défini l'écriture des *adresses de socket :*
✓local : sont des noms de fichiers ordinaires
✓Internet sont des *adresses IP* d'un hôte connecté au réseau et d'un numéro de port.

**Un protocole** spécifie comment les données sont transmises (TCP/IP).

# Communication par Socket