

# Systemes d'exploitation

## ENSISA 1A

**Maxime Devanne**

[maxime.devanne@uha.fr](mailto:maxime.devanne@uha.fr)

Bureau 3.37

1A IR

1<sup>er</sup> semestre

# Systemes d'exploitation

ENSISA 1A

## Chapitre 2

### Processus

## Objectifs pédagogiques du chapitre

- Connaître les **caractéristiques** des **processus**
- Comprendre les différents méthode **d'ordonnancement** des processus
- Connaître les **caractéristiques** des **threads** et appréhender ce qui les diffère des processus
- Comprendre la **synchronisation** entre processus



Introduction

Les processus

Notions de programmation des processus

L'ordonnancement

**Les threads**

Synchronisation

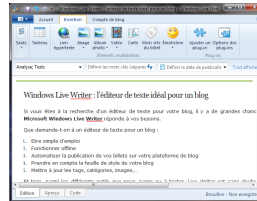
Conclusion

# Les threads

## Introduction

### ● Exemple

- Un programme de traitement de texte avec rendu du document comme à l'impression
- Un utilisateur écrit un livre au clavier
- Le programme permet de sauvegarder automatiquement le document pour ne pas perdre les données



Un programme : un processus

Ex: attente de l'utilisateur lors de la sauvegarde et inversement

Noyau

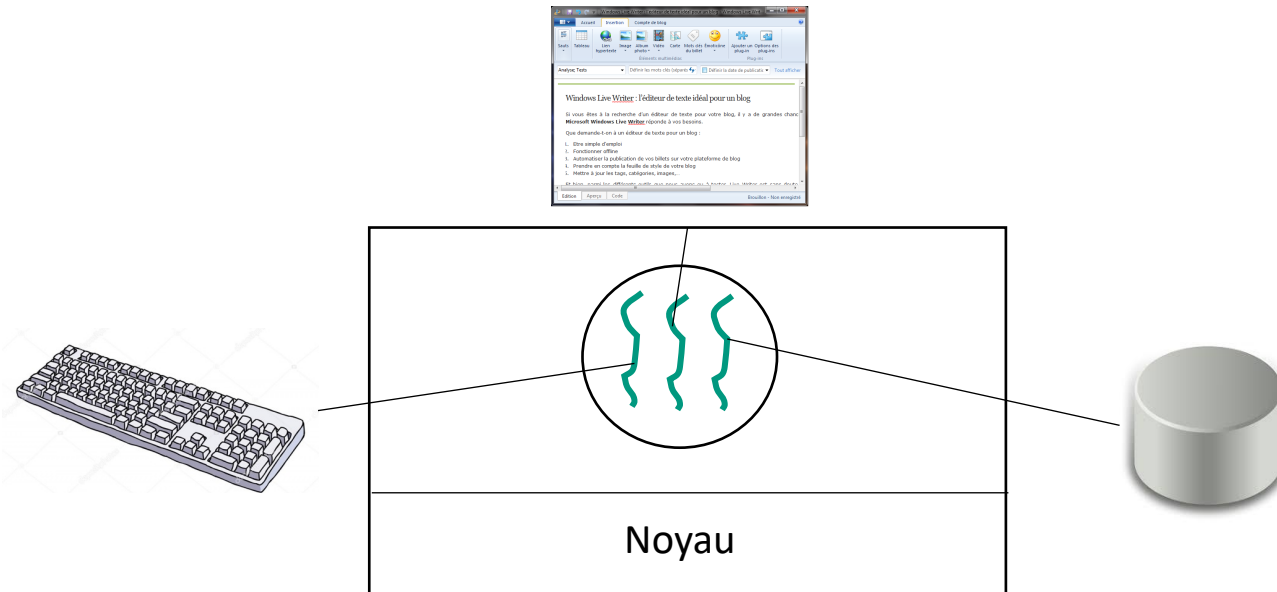


# Les threads

## Introduction

### ● Exemple

- Un programme de traitement de texte avec rendu du document comme à l'impression
- Un utilisateur écrit un livre au clavier
- Le programme permet de sauvegarder automatiquement le document pour ne pas perdre les données

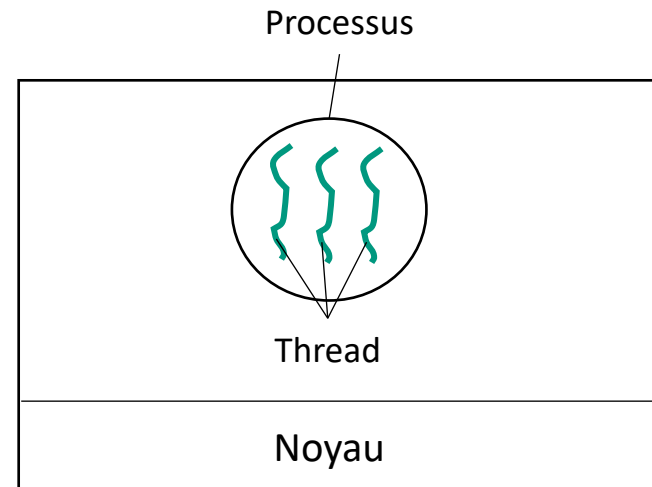
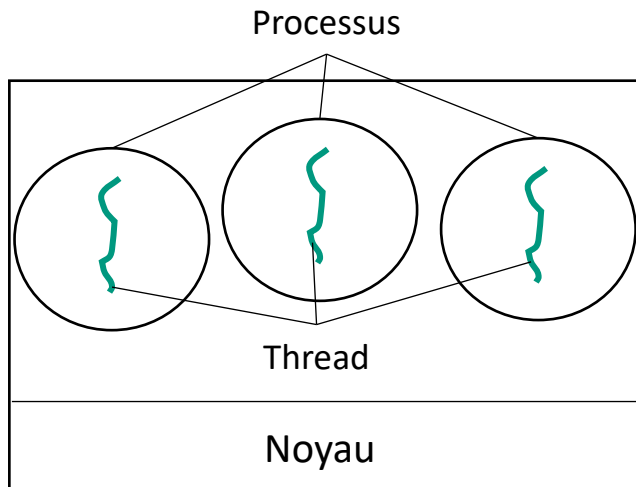


# Les threads

## Introduction

### ● Définition

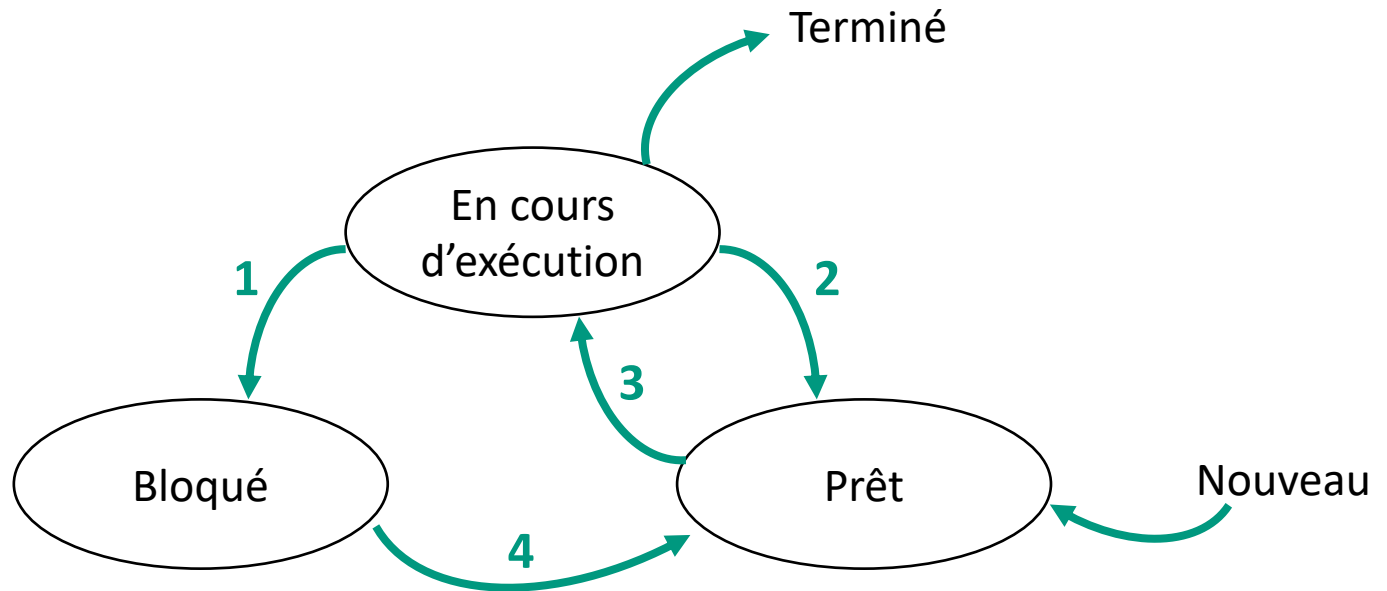
- Un thread est un chemin (fil) d'exécution dans un processus
  - On parle de processus légers
  - Un thread ne peut pas vivre en dehors d'un processus
- Un processus peut avoir plusieurs threads
  - Multithreading (exécution pseudo parallèle)
  - Analogie avec la multiprogrammation pour les processus



# Les threads

## Les états et transitions d'un thread

- Similaires aux processus
  - En cours d'exécution
  - Prêt
  - Bloqué
- Pareil pour les transitions

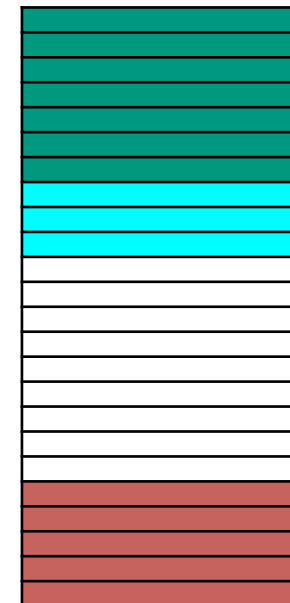




# Les threads

## Thread VS processus

- Thread = processus ?
  - Similaires sur beaucoup d'aspects
    - Facilite leur gestion
  - Une différence majeure :
    - **Les threads partagent le même espace mémoire** (celui du processus)
    - Ex: partage des variable globales
  - Rappel: structure de processus
    - TEXT (instructions)
    - DATA (variables globales)
    - STACK (pile d'exécution)

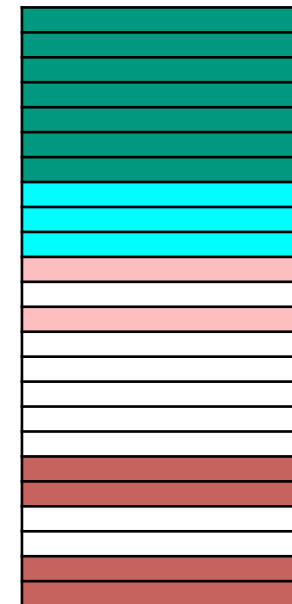


Espace mémoire

# Les threads

## Thread VS processus

- Thread = processus ?
  - Similaires sur beaucoup d'aspects
    - Facilite leur gestion
  - Une différence majeure :
    - **Les threads partagent le même espace mémoire** (celui du processus)
    - Ex: partage des variable globales
  - Thread
    - TEXT (instructions)
    - Variables globales partagées
    - pile d'exécution privée
    - Variables locales privées



Espace mémoire

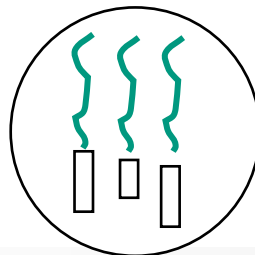
# Les threads

## Partage de mémoire

- Les threads partagent le même espace mémoire
  - Au sein d'un même processus
  - Certains éléments sont propres à chaque thread

| Eléments par processus | Eléments par thread |
|------------------------|---------------------|
| Variables globales     | Compteur ordinal    |
| Fichiers ouverts       | Registres           |
| Processus enfant       | Pile                |
| Alertes en attente     | Etat                |
| Signaux                |                     |

- Un thread peut modifier ou supprimer une variable globale
- Chaque thread possède sa propre pile d'exécution



# Les threads

## Création et terminaison

- Un thread peut créer un autre thread
  - Multithreading
  - Il n'y a pas de notion de hiérarchie
  - Chaque thread est au même niveau
- Lorsque un thread se termine, il disparaît

# Les threads

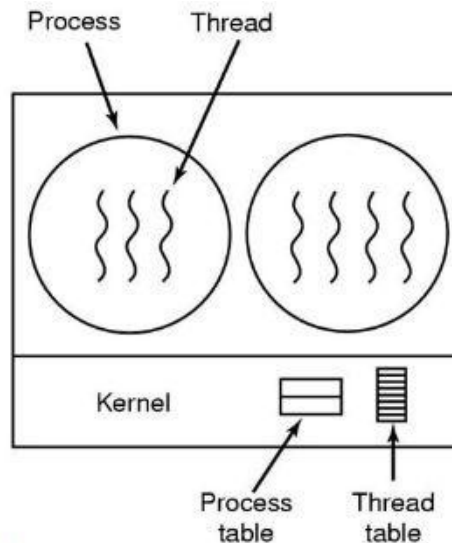
## Ordonnancement des threads

- Pas aussi simple que pour les processus
  - Cas du multithreading
  - Plusieurs processus ont plusieurs threads
  - Plusieurs niveaux de parallélisme
- On distingue deux cas
  - Les threads utilisateurs
    - Le système n'a pas conscience des threads
  - Les threads noyaux
- La plupart des algorithmes vus précédemment peuvent être utilisés
  - Souvent Round robin
    - quantum à définir

# Les threads

## Type de thread

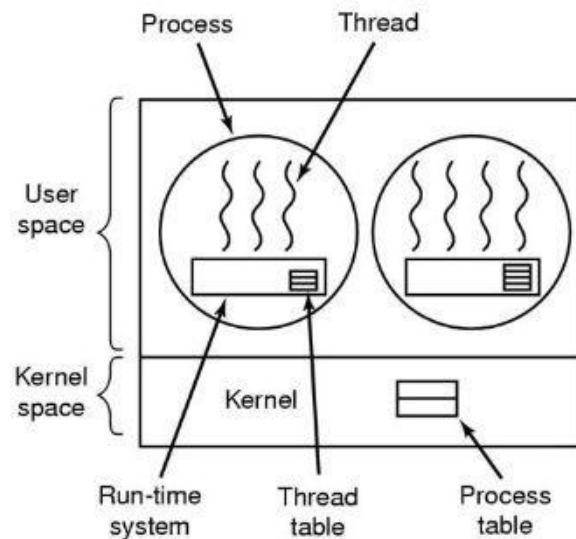
- Il existe deux types de threads
  - Les threads noyaux
    - Ce sont ceux connus par le SE
    - Chaque processus a au moins un thread noyau



# Les threads

## Type de thread

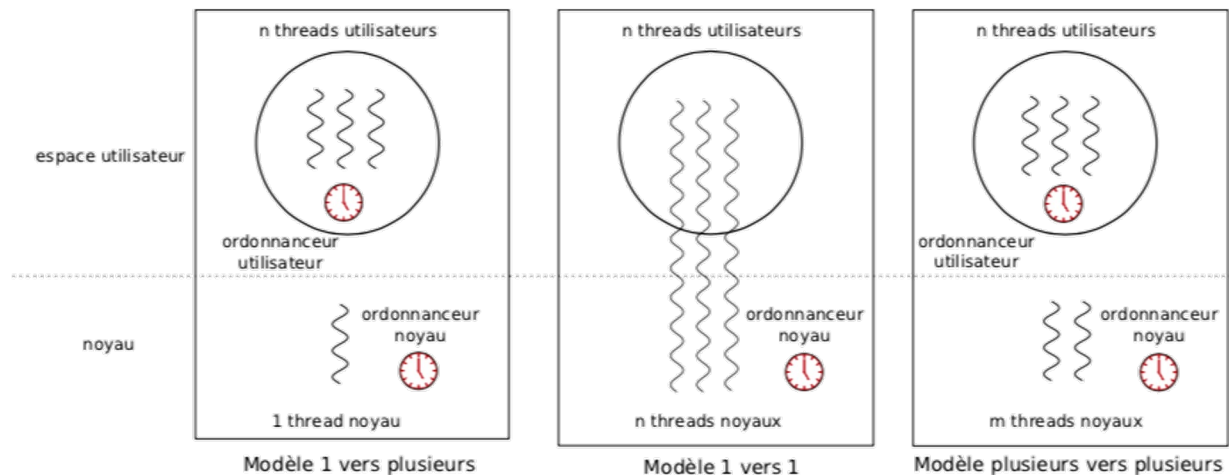
- Il existe deux types de threads
  - Les threads utilisateurs
    - Ils sont implémentés de manière logicielle par le programme utilisateur
    - Le SE n'a pas connaissance de leur existence



# Les threads

## Mode d'exécution des threads

- Il existe trois principaux modes d'exécution des threads
  - Modèle 1 vers plusieurs
  - Modèle 1 vers 1
  - Modèle plusieurs vers plusieurs



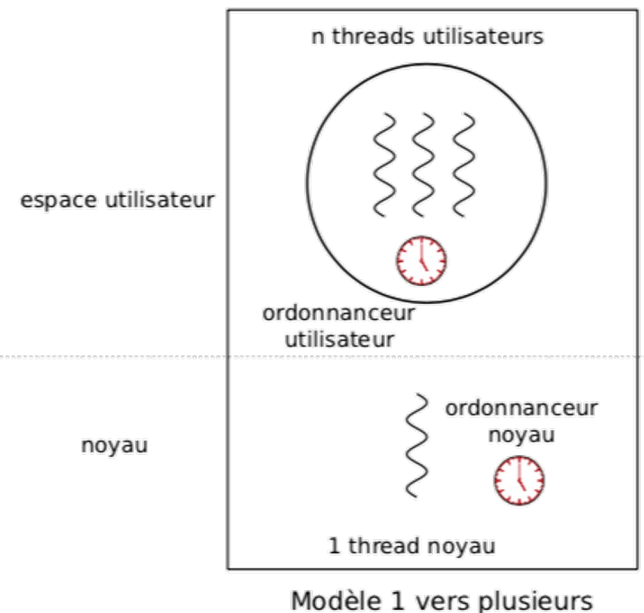


# Les threads

## Modèle 1 vers plusieurs

### ● Mode de fonctionnement

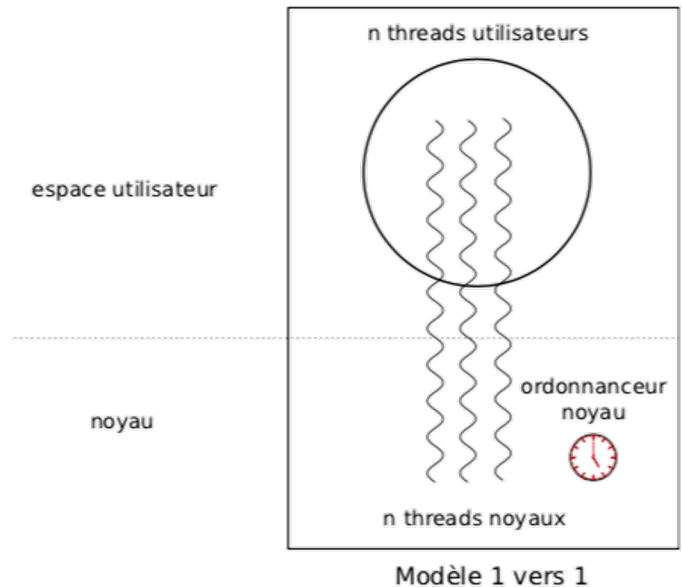
- Dans un modèle 1 vers plusieurs, on ordonnance les threads noyaux
- A chaque thread noyau correspond plusieurs threads utilisateurs
- Les threads utilisateurs possèdent leur propre ordonnanceur dans l'espace utilisateur
- Cette solution est la plus légère
- La gestion des threads ne demande pas d'appels système mais un appel système va bloquer tous les threads
  - Le thread noyau est bloqué
- Pas de possibilités d'exécution sur plusieurs processeurs ou cœurs



# Les threads

## Modèle 1 vers 1

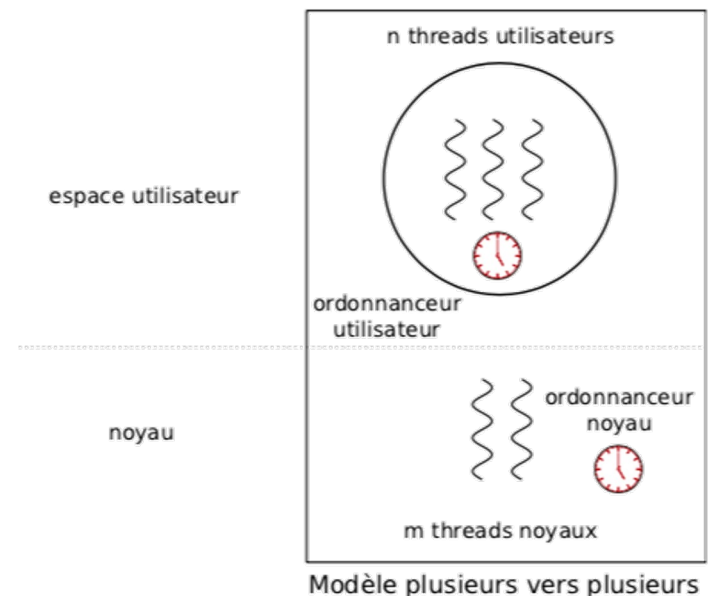
- Mode de fonctionnement
  - Dans un modèle 1 vers 1, chaque thread utilisateur correspond à un thread noyau
  - Cette solution peut poser un problème d'allocation des ressources
  - Les threads étant tous égaux
    - les processus ayant beaucoup de threads sont favorisés
  - Certains systèmes permettent d'ordonnancer des groupes de threads pour résoudre ce problème



# Les threads

## Modèle plusieurs vers plusieurs

- Mode de fonctionnement
  - Dans un modèle plusieurs vers plusieurs, on peut associer un nombre quelconque de threads utilisateurs à un nombre quelconque de threads noyaux
  - Solution théoriquement la plus modulaire
  - Elle permet généralement de réduire le nombre de threads noyaux par rapport au modèle 1 vers 1
  - Mais le risque qu'un thread utilisateur se retrouve en famine de thread noyau peut exister




# Les threads

## Ordonnancement des threads

- Exemple

- Deux processus multithreads **A** et **B** (3 threads par processus)
- Quantum 20 ms
- Chaque thread dure 5ms

- Thread utilisateur

- Résultat : A<sub>1</sub> A<sub>2</sub> A<sub>3</sub> A<sub>1</sub> B<sub>1</sub> B<sub>2</sub> B<sub>3</sub> B<sub>1</sub> A<sub>2</sub>
- L'ordonnanceur de processus n'a pas conscience des threads

- Thread noyau

- Résultat : A<sub>1</sub> B<sub>1</sub> A<sub>2</sub> B<sub>2</sub> A<sub>3</sub> B<sub>3</sub>
- La préemption peut se faire en mode noyau (top d'horloge)

# Les threads

## Ordonnancement des threads

### ● Exercice 1

- Deux processus multithreads **A** et **B** (3 threads par processus)
- Quantum 20 ms
- Chaque thread dure 10ms
- Dérouler l'ordonnancement Round Robin dans les deux cas

### ● Thread utilisateur

- Résultat : A sequence of nine colored boxes representing thread execution: A1 (red), A2 (red), B1 (blue), B2 (blue), A3 (red), A1 (red), B3 (blue), B1 (blue), and A2 (red).

### ● Thread noyau

- Résultat : A sequence of six colored boxes representing thread execution: A1 (red), B1 (blue), A2 (red), B2 (blue), A3 (red), and B3 (blue).

## Ordonnancement des threads

### ● Exercice 2

- Deux processus multithreads **A** et **B** (2 par processus)
- Quantum 5ms
- Chaque thread dure 10ms
- Dérouler l'ordonnancement Round Robin dans les deux cas

### ● Thread utilisateur

- Résultat : A sequence of eight colored boxes: red (A<sub>1</sub>), blue (B<sub>1</sub>), red (A<sub>1</sub>), blue (B<sub>1</sub>), red (A<sub>2</sub>), blue (B<sub>2</sub>), red (A<sub>2</sub>), blue (B<sub>2</sub>).

### ● Thread noyau

- Résultat : A sequence of eight colored boxes: red (A<sub>1</sub>), blue (B<sub>1</sub>), red (A<sub>1</sub>), blue (B<sub>1</sub>), red (A<sub>2</sub>), blue (B<sub>2</sub>), red (A<sub>2</sub>), blue (B<sub>2</sub>).
- La préemption peut se faire en mode noyau (top d'horloge)

## Du code monothread au code multithread

- De nombreux programmes sont écrits pour les processus monothread
  - Le convertir en multithread est plus délicat qu'on pourrait l'imaginer
  - Ce n'est pas impossible
  - Il est préférable de réfléchir au multithread lors de la conception du programme

## Threads de POSIX

- Norme IEEE
  - Paquetage Pthread
- **pthread\_create**
  - Crée le nouveau thread
- **pthread\_exit**
  - Termine le thread appelant
- **pthread\_join**
  - Attend la fin d'un thread
- **pthread\_yield**
  - Libère l'UC pour laisser un autre thread s'exécuter
- **pthread\_attr\_init**
  - Crée et initialise une structure attribut de thread
- **pthread\_attr\_destroy**
  - Supprime une structure attribut de thread



# Notions de programmation des processus

## Threads en C

- La bibliothèque pthread permet de créer et gérer les threads
- Inclure `#include <pthread.h>` et compiler avec l'option `-lpthread`
- `int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(* start_routine) (void*), void *arg);`
  - Crée un nouveau thread, lance *start\_routine* avec les arguments *arg*
- `void pthread_exit (void *retval);`
  - Détruit le thread courant
- `int pthread_join (pthread_t th, void **thread_return);`
  - Attend que thread se termine

# Notions de programmation des processus

## Threads en C

### ● Exemple

```
1  #include<pthread.h>
2  #include<stdio.h>
3
4  #define pthread_attr_default ((pthread_attr_t *)NULL)
5
6  int val[3];
7  void * thread(void *);
8
9  int main(int c, char *v[]) {
10     int num,i;
11     pthread_t pthread_id[3];
12     for(num=0;num<3;num++) {
13         pthread_create(pthread_id+num,
14             pthread_attr_default, thread,
15             (void *)num);
16     }
17
18     for(i=0;i<30;i++){
19         printf("%d_%d_%d\n",val[0],val[1],val[2]);
20     }
21     exit(0);
22 }
23
24 void *thread(void *num) {
25     for(;;) {
26         val[(int)num]++;
27     }
28 }
```



Introduction

Les processus

Notions de programmation des  
processus

L'ordonnancement

Les threads

**Synchronisation**

Conclusion

## Introduction

- Synchronisation entre processus

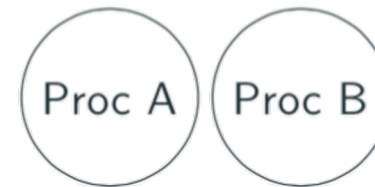
- Les processus ont besoin de communiquer entre eux et de se synchroniser
- Comment un processus passe des informations à un autre processus ?
- Une solution :
  - Utilisation d'un espace de stockage commun
- Cet espace de partage implique des accès concurrents aux informations
  - Cela peut entraîner des incohérences qui peuvent être plus ou moins critiques
- Note : fonctionne pareil pour les threads
  - Les threads ont déjà un espace de stockage commun
- **Les conditions de concurrence**

# Synchronisation

## Exemple

- File d'impression

out : 5  
in = 7



### Proc A :

```
local_in = in;  
print(local_in, « AAA.pdf »);  
in = local_in+1
```

### Proc B :

```
local_in = in;  
print(local_in, « BBB.pdf »);  
in = local_in+1
```

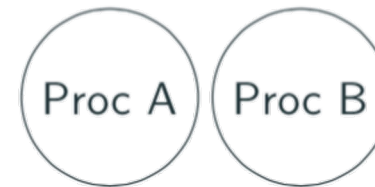
# Synchronisation

## Exemple

- File d'impression



out : 5  
in = 7



### Proc A :

```
local_in = in;  
print(local_in, « AAA.pdf »);  
in = local_in+1
```

### Proc B :

```
local_in = in;  
print(local_in, « BBB.pdf »);  
in = local_in+1
```

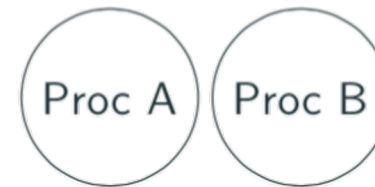
**Proc A est préempté, Proc B prend la main**

# Synchronisation

## Exemple

- File d'impression

out : 5  
in = 7



### Proc A :

```
local_in = in;  
print(local_in, « AAA.pdf »);  
in = local_in+1
```

### Proc B :

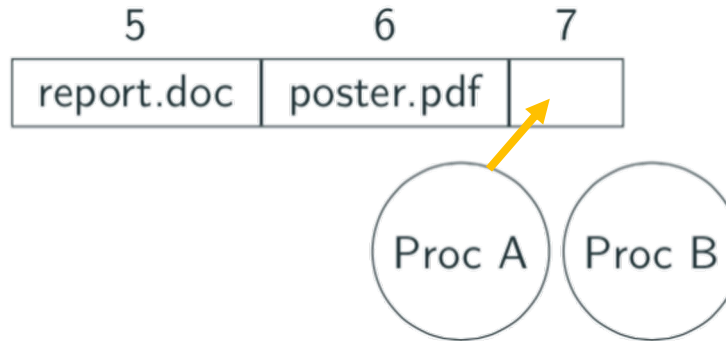
```
local_in = in;  
print(local_in, « BBB.pdf »);  
in = local_in+1
```

# Synchronisation

## Exemple

- File d'impression

out : 5  
in = 7



### Proc A :

```
local_in = in;  
print(local_in, « AAA.pdf »);  
in = local_in+1
```

### Proc B :

```
local_in = in;  
print(local_in, « BBB.pdf »);  
in = local_in+1
```

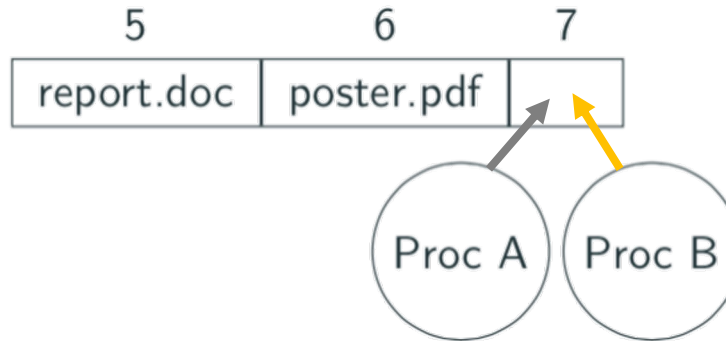


# Synchronisation

## Exemple

- File d'impression

out : 5  
in = 7



### Proc A :

```
local_in = in;  
print (local_in, « AAA.pdf »);  
in = local_in+1
```

### Proc B :

```
local_in = in;  
print(local_in, « BBB.pdf »);  
in = local_in+1
```

**Incohérence : BBB.pdf remplace AAA.pdf dans la file 7**

## L'exclusion mutuelle

- Définitions

- Ressource critique :

- Objet partagé par plusieurs processus et susceptible d'être nécessaire au même instant

- Section critique :

- Morceau de programme qui accède à une ressource critique

- L'exclusion mutuelle

- Interdiction pour plusieurs processus de se trouver simultanément à l'intérieur d'une section critique

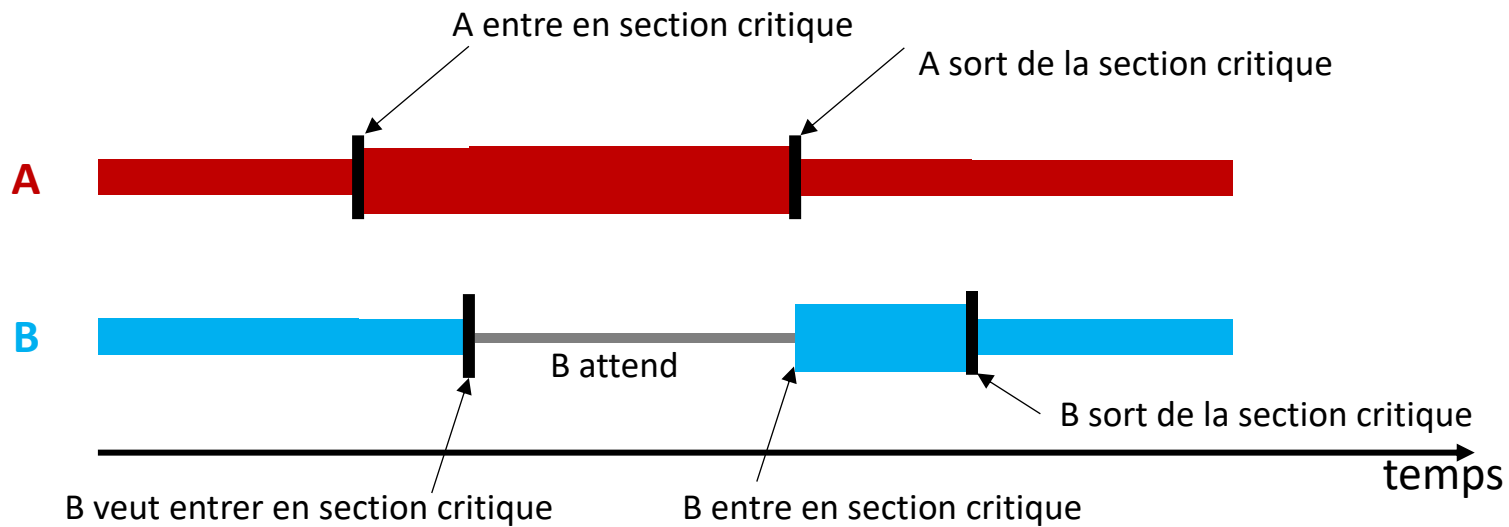
## L'exclusion mutuelle

- Quatre conditions pour réaliser l'exclusion mutuelle :
  - 1. Deux processus ne doivent pas se trouver simultanément dans leurs sections critiques
  - 2. Il ne faut pas faire de supposition quand à la vitesse ou au nombre de processus mis en œuvre
  - 3. Aucun processus s'exécutant à l'extérieur de sa section critique ne doit bloquer d'autres processus
  - 4. Aucun processus ne doit attendre indéfiniment pour pouvoir entrer dans sa section critique

# Synchronisation

## L'exclusion mutuelle

### ● Illustration



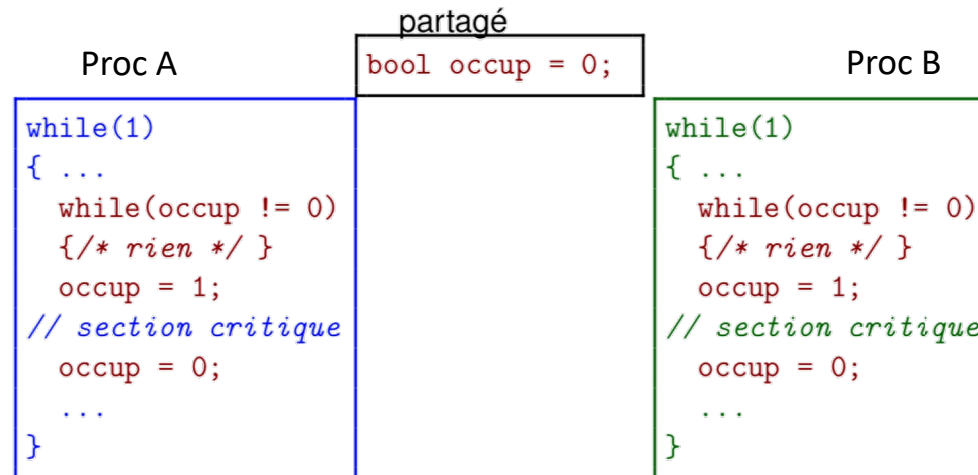
## L'exclusion mutuelle

- Mécanismes d'exclusion mutuelle
  - 1. Attente active
  - 2. Le sommeil et l'activation
  - 3. Les sémaphores
  - 4. Les mutex

# Synchronisation

## L'exclusion mutuelle

- Attente active
  - Test d'une condition dans une boucle
  - Jusqu'à ce que la condition soit vraie
- Alternance strict

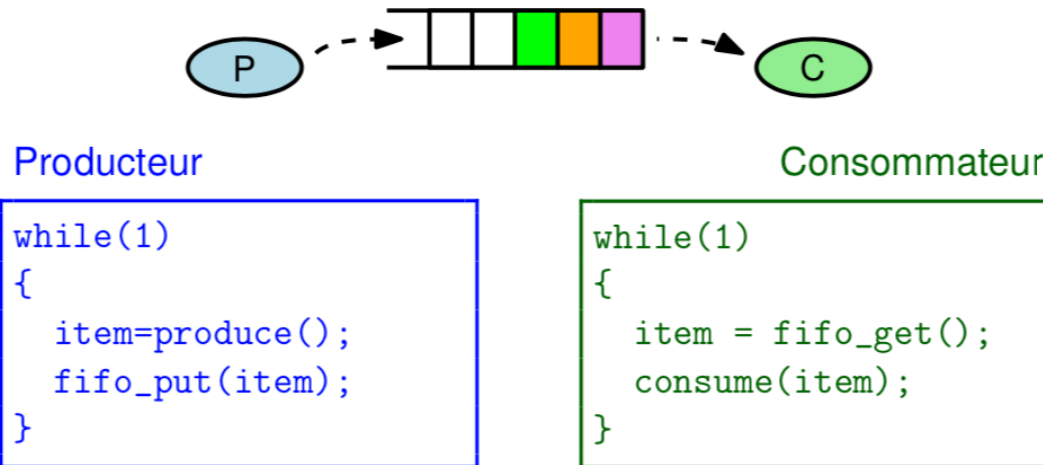


- Grande consommation de temps processeur
- Attente active acceptable dans le cas de processus brefs

# Synchronisation

## Le problème du producteur-consommateur

- Deux processus communiquent via une file FIFO partagée



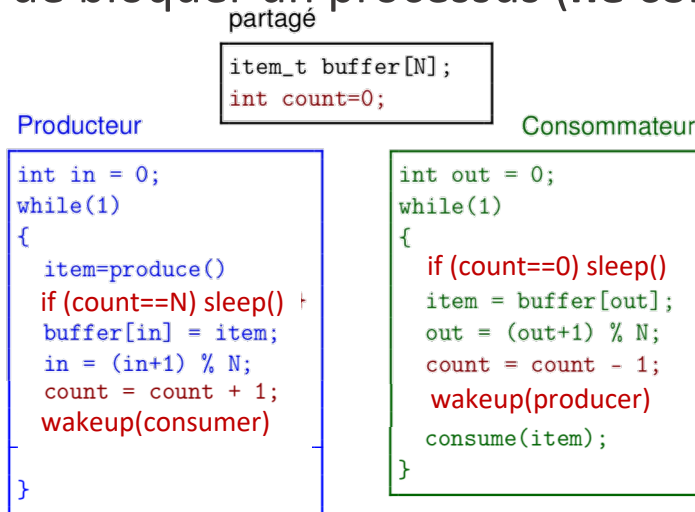
- Taille de la file fixe
- Le producteur doit attendre tant que la file est pleine
- Le consommateur doit attendre tant que la file est vide

# Synchronisation

## L'exclusion mutuelle

- Le sommeil et l'activation

- Utilisation de primitives sleep() et wakeup()
- sleep() permet de bloquer un processus (**ne consomme pas l'UC**)



- Problème si changement de processus au mauvais moment
- Le consommateur voit 0 puis est préempté (sans faire sleep() )
- wakeup() est envoyé par P à C qui ne dort pas (signal perdu)
- Lorsque C reprend la main, il avait lu donc s'endort... à jamais



# Synchronisation

## L'exclusion mutuelle

- Les sémaphores

- Contient une variable entière  $k$ 
  - Initialisée avec  $k \geq 0$
- Contient une **liste d'attente** de threads bloqués
- Offre deux méthodes atomiques  $P()$  et  $V()$ 
  - ou  $down()$  et  $up()$

$P(S)$

```
S.k = S.k - 1;  
if( S.k < 0 )  
{  
    /* suspendre le thread  
       appelant, et le  
       mettre dans la file  
       d'attente de S */  
}
```

$V(S)$

```
S.k = S.k + 1;  
if( S.k <= 0 )  
{  
    /* réveiller l'un des  
       threads de la file  
       d'attente de S */  
}
```

# Synchronisation

## L'exclusion mutuelle

- Les sémaphores

- Full et empty

partagé

```
item_t buffer[N];  
sem_t emptyslots=N;  
sem_t fullslots=0;
```

Producteur

```
in = 0;  
while(1)  
{  
    item=produce()  
    P(emptyslots);  
    buffer[in] = item;  
    in = (in+1) % N;  
    V(fullslots);  
}
```

Consommateur

```
out = 0;  
while(1)  
{  
    P(fullslots);  
    item = buffer[out];  
    out = (out+1) % N;  
    V(emptyslots);  
    consume(item);  
}
```

- Mécanisme de synchronisation très polyvalent

## L'exclusion mutuelle

- Les mutex (verrou exclusif)
  - Version simplifiée des sémaphores
    - Lorsque le décompte n'est pas nécessaire (nb de places dans la file)
  - Deux états possibles
    - `lock()` : si le verrou est libre, le prendre sinon se bloquer jusqu'à ce qu'il soit libre
    - `unlock()` : relâcher le verrou, le rendre libre
- Remarques
  - Dans le cas des sémaphores, si  $k$  est initialisé à 1, alors c'est équivalent à  $P() = \text{lock}()$  et  $V() = \text{unlock}()$
  - Un sémaphore ayant le rôle de mutex est souvent utilisé
    - En plus des compteurs full et empty
    - Permet d'assurer l'exclusion mutuelle dans le cas de plusieurs processeurs

## L'exclusion mutuelle

- Autres mécanismes d'exclusion mutuelle
  - 5. Les moniteurs
  - 6. L'échange de message
  - 7. Les barrières



Introduction

Les processus

L'ordonnancement

Les threads

Notions de programmation des  
processus et threads

Synchronisation

**Conclusion**

## Conclusion

- Threads

- Un thread est un fil d'exécution du processus
- Un processus peut avoir plusieurs threads
- Chaque thread d'un même processus partagent l'espace mémoire du processus

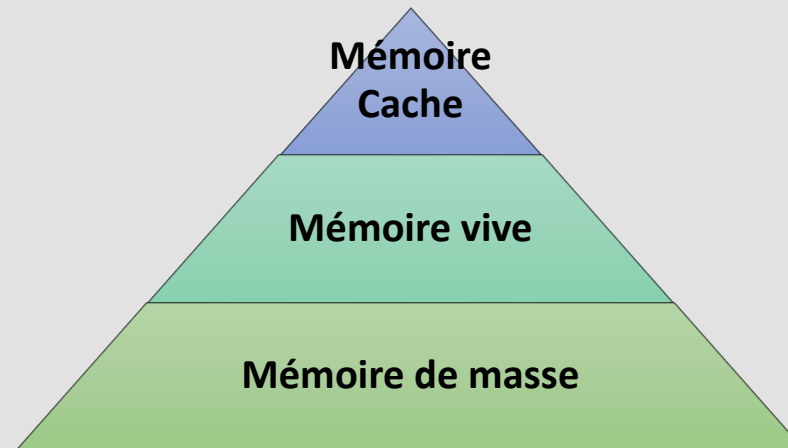
- Synchronisation entre les processus

- Concurrence entre les processus
- Exclusion mutuelle
  - Stratégie pour éviter des sections critiques simultanées
- Sémaphore
  - Mécanisme de synchronisation universel

# La prochaine séance

## La prochaine séance

- **Chapitre 3 : La mémoire**



# Conclusion

## Et maintenant ?

- **On passe aux exercices !**
  - Disponibles sur Moodle
- N'oubliez pas à l'issue de la séance
  - Quizz
  - Feedback



# Systemes d'exploitation

ENSISA 1A

## Chapitre 2

### Processus