# Principes des systèmes d'exploitation

# Synchronisation

UFR IM$^2$AG

M1 MIAGE & M1 MEEF NSI 2023-2024

Renaud Lachaize

# Crédits et remerciements

- **Le contenu de ce support de cours est partiellement inspiré, voire emprunté aux travaux d'autres personnes :**

  - Rodrigue Chakode, Fabienne Boyer, Vincent Danjean, Sacha Krakowiak, Arnaud Legrand, Vania Marangozova-Martin (UFR IM$^2$AG)
  - David Mazières (Stanford University)
  - Randall Bryant, David O'Hallaron, Gregory Kesden, Markus Püschel (Carnegie Mellon University)

  - Ouvrages de référence (voir détails sur la page web) :
    - Silberschatz et al. Operating systems concepts with Java.
    - Tanenbaum. Modern operating systems.
    - Bryant and O'Hallaron. Computer systems: a programmer's perspective.

# Objectifs du cours

- **Comprendre les besoins de synchronisation entre différents flots d'exécution (processus ou threads)**

- **Comprendre les mécanismes qui permettent de réaliser la synchronisation**
  - Implémentation
  - Utilisation correcte et efficace

# Plan

- **Introduction & définitions**
- **Mécanismes de base**
- **Notions complémentaires**

# Introduction

- **Flots d'exécution concurrents**
  - Threads au sein d'un même processus
  - Différents processus
  - **Par la suite, sauf mention contraire, on utilisera le terme « processus » de manière générique**

- **Situation de compétition  ou de coopération entre processus**

  - **Compétition** : plusieurs processus veulent accéder à la même ressource (par exemple, modifier le même fichier ou la même variable)

  - **Coopération** : plusieurs processus interagissent pour mener à bien une tâche – ils doivent communiquer régulièrement pour échanger des informations et déterminer l'avancement global

  - Les deux situations nécessitent de synchroniser les processus

# Exemple introductif n°1 : Compte bancaire

- **Deux opérations sur le même compte exécutées en concurrence**

**processus p1 : créditer(1867A, 1000)**

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 1000
3. ecrire_compte (1867A, nouveau)
```

**processus p2 : créditer(1867A, 3000)**

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 3000
3. ecrire_compte (1867A, nouveau)
```

- **À noter :**
  - les variables **`courant`** et **`nouveau`** sont locales à chaque processus
  - les deux processus se déroulent en parallèle. L'exécution des opérations peut être entrelacée dans un ordre quelconque, à condition de respecter l'ordre local pour chacun des processus
- **Exemple de séquences d'exécution**
  - Exécution n°1 : p2.1 ; p2.2 ; p2.3 ; p1.1 ; p1.2 ; p1.3
  - Exécution n°2 : p1.1 ; p1.2 ; p2.1 ; p2.2 ; p2.3 ; p1.3

# Exemple introductif n°2 : tampon producteur/consommateur (1/2)

- Producteur

```
while (true) {

  // produce an item in
  // nextProduced

  while (count == BUFFER_SIZE)
    ; // do nothing

  buffer[in] = nextProduced;
  in = (in + 1)%BUFFER_SIZE;
  count++;
}
```

- Consommateur

```
while (true) {

  while (count == 0)
    ; // do nothing

  nextConsumed = buffer[out];
  out = (out + 1)%BUFFER_SIZE;
  count--;

  // consume the item in
  // nextConsumed
}
```

# Exemple introductif n°2 : tampon producteur/consommateur (2/2)

- En langage machine

```
count++ se traduit par :
    register1 = count
    register1 = register1 + 1
    count = register1
```

```
count-- se traduit par :
    register2 = count
    register2 = register2 - 1
    count = register2
```

- Considérons la séquence suivante, avec initialement la valeur 5 stockée dans count

  S0: prod. exécute register1 = count {register1 = 5}
  S1: prod. exécute register1 = register1 + 1 {register1 = 6}
  S2: cons. exécute register2 = count {register2 = 5}
  S3: cons. exécute register2 = register2 - 1 {register2 = 4}
  S4: prod. exécute count = register1 {count = 6 }
  S5: cons. exécute count = register2 {count = 4}

- Conclusion ?

# Le problème de l'exclusion mutuelle

Comment éviter les problèmes d'incohérences introduits par des accès concurrents à une ressource partagée ?

Assurer que l'ensemble des opérations (consultation + mise à jour) est exécutée de manière **indivisible** (« atomique »)

Ainsi, pas d'interférences possibles de la part d'autres opérations exécutées en parallèle

Exemple :

A1

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 1000
3. ecrire_compte (1867A, nouveau)
```

A2

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 3000
3. ecrire_compte (1867A, nouveau)
```

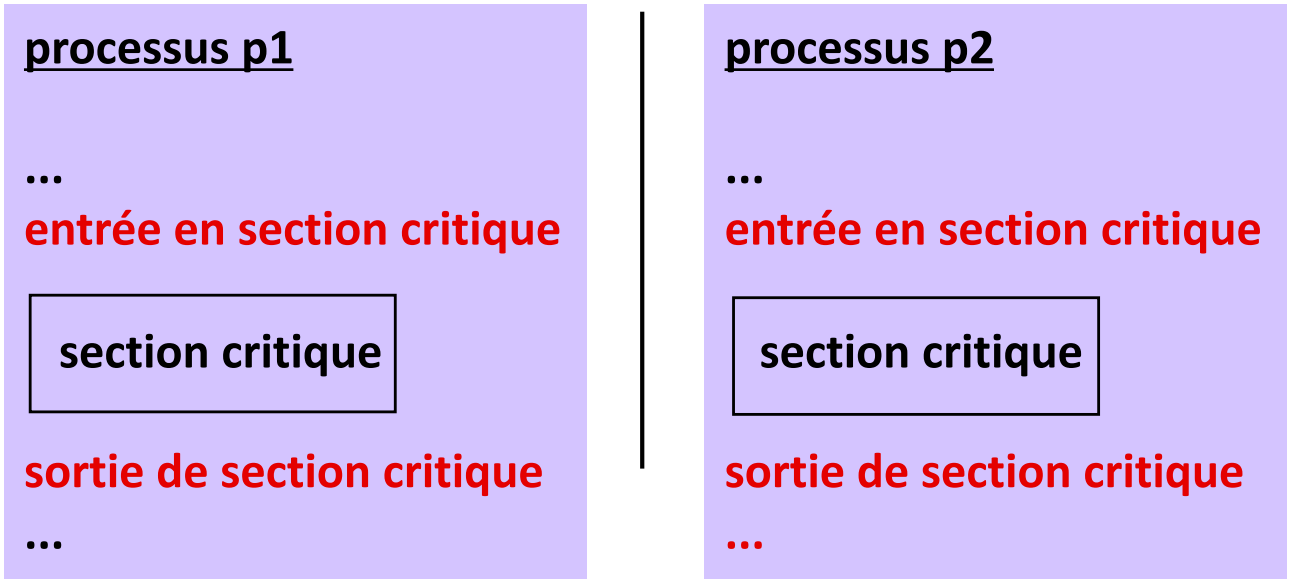
Si A1 et A2 sont atomiques, le résultat de l'exécution parallèle de A1 et A2 ne peut être que celui de A1 ; A2 ou de A2 ; A1, à l'exclusion de tout autre.

On dit aussi que la séquence d'actions 1; 2; 3 (dans A1 et A2) est une **section critique** : elle doit être exécutée en **exclusion mutuelle** (un seul processus au plus peut être dans sa section critique à un instant donné)

# Section critique

- **Schéma général**

**déclaration et initialisation de variables communes**




**Les opérations “entrée en section critique”, ”sortie de section critique” doivent garantir l’exclusion mutuelle**

**Attention : Ne faire aucune hypothèse sur les vitesses d’exécution relatives des différents processus**

# Section critique

- **Trois propriétés requises pour une solution au problème de la section critique**

- **Exclusion mutuelle**
  - Si un processus Pi se trouve dans la section critique, aucun autre processus ne peut y entrer.
- **Progrès**
  - Si aucun processus ne se trouve dans la section critique et un ou plusieurs processus souhaitent y entrer, alors seuls les processus candidats à la section critique peuvent participer au choix du prochain processus admis en section critique, et ce choix est fait au bout d'un temps borné.
- **Attente bornée**
  - Quand un processus demande à entrer en section critique, il existe une limite sur le nombre d'admissions d'autres processus dans cette section critique qui ont lieu entre le moment où la demande est faite et le moment où la demande est acceptée.

# Réalisation d'une section critique

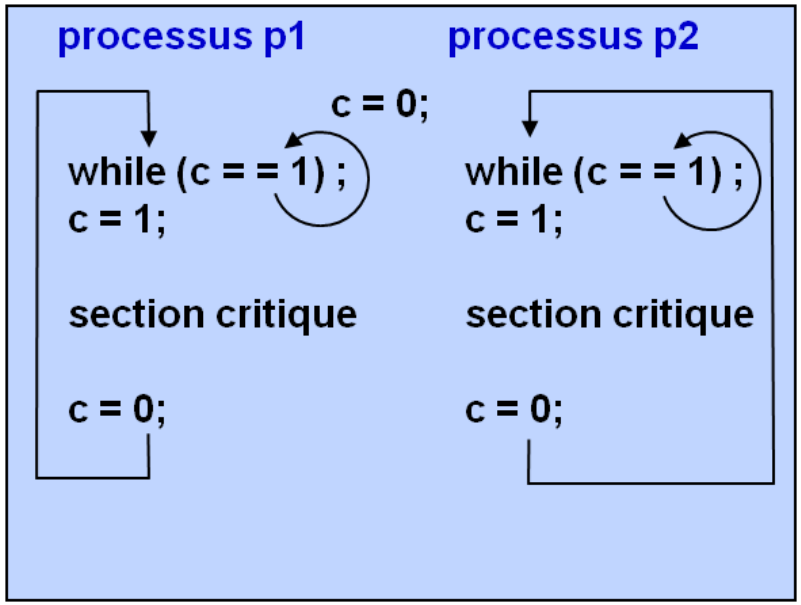
- **Pour commencer, considérons une approche aussi simple que possible, dite par « attente active »**
  - Un processus qui veut entrer en section critique boucle jusqu'à ce qu'il soit admis
  - Approche **inefficace** (en particulier dans un système monoprocesseur) – à éviter

- **Autres hypothèses :**
  - Seulement deux processus
  - Système monoprocesseur

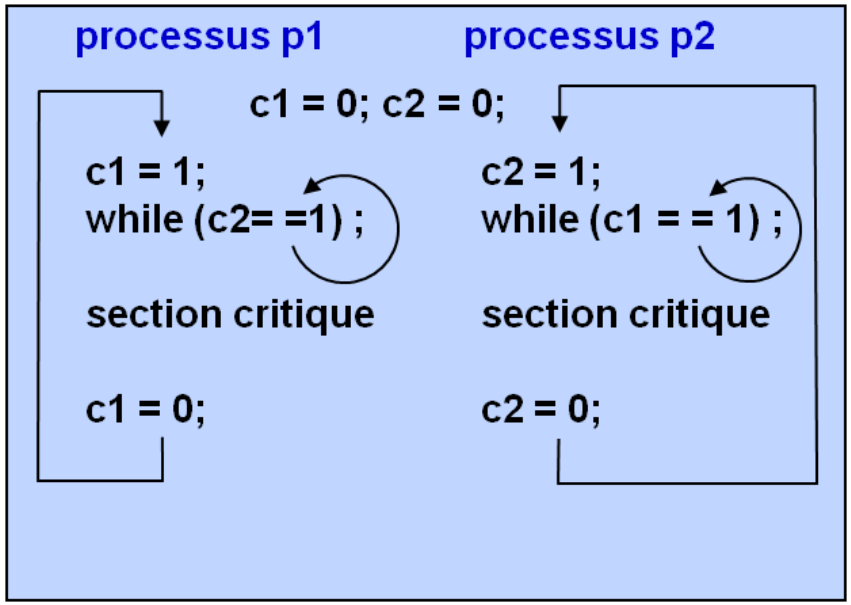
- **Même dans un contexte aussi simple, le problème de la section critique n'est pas trivial ...**

# Réalisation d'une section critique par attente active

## Proposition n°1

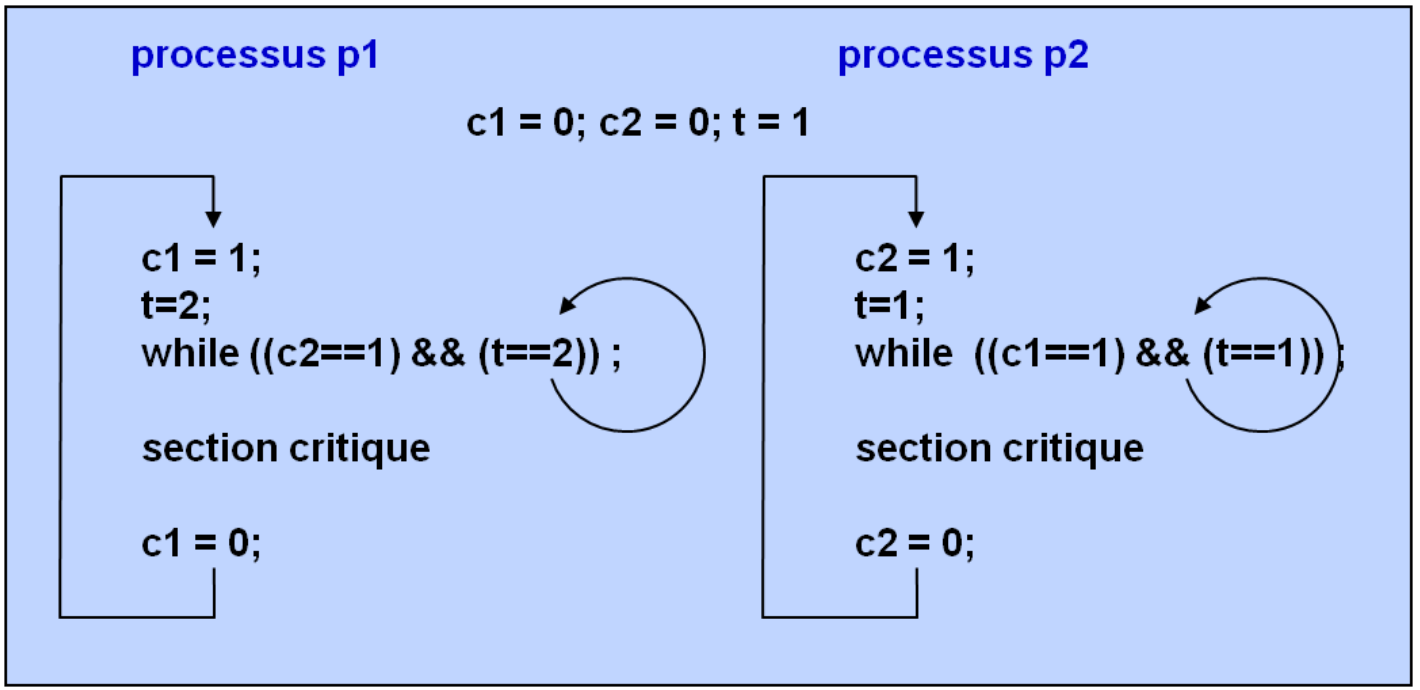



## Proposition n°2



# Réalisation d'une section critique par attente active (suite)

**Une solution correcte pour l'exclusion mutuelle par attente active avec 2 processus (Peterson, 1981)**




**Exercice : prouver que cette solution respecte les 3 propriétés liées au problème de la section critique**

# Réalisation d'une section critique
# Solution plus générale

- **De façon plus générale, pour garantir une gestion correcte d'une section critique, il faut :**
  - Des primitives permettant de prendre un « verrou » exclusif (*lock*) en entrée de la section critique et de le relâcher en sortie (*unlock*)
  - Garantir l'indivisibilité de la séquence qui consulte et met à jour l'état d'un verrou
- **Comment faire ?**

# Primitives de verrouillage

## Cas d'un système monoprocesseur

- **Comment implémenter les primitives lock/unlock de façon indivisible ?**
- **Quelles sont les circonstances qui peuvent remettre en cause l'indivisibilité de l'opération ?**
  - Un appel synchrone à l'ordonnanceur ?
    - Non : pas de point de blocage entre la consultation et la modification de l'état du verrou
  - Un appel asynchrone à l'ordonnanceur ?
    - Oui : par défaut, un interruption matérielle peut survenir n'importe quand
- **On peut implémenter les primitives lock/unlock via des appels systèmes**
  - Idée : le noyau peut masquer les interruptions avant d'exécuter l'opération puis les démasquer ensuite

# Primitives de verrouillage
# Cas d'un système multiprocesseurs

- **La solution précédente est-elle valable dans ce contexte ?**
  - **Non** :
    - Les interruptions ne sont pas la seule cause d'entrelacements problématiques entre plusieurs flots d'exécutions
    - Plusieurs processeurs => parallélisme matériel : plusieurs instructions exécutées en parallèle

- **Comment assurer l'indivisibilité de la consultation et de la mise à jour d'une variable « verrou » malgré tout ?**
  - Impossible à garantir de façon générale/efficace en utilisant uniquement des techniques logicielles
  - Nécessité d'un support spécifique au niveau matériel :
    - Instruction « Test & Set » ou « Swap »
  - Technique également valable sur machine monoprocesseur

# Instruction matérielle « Test & Set »

## Définition

```
boolean
TestAndSet(boolean*target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

## Utilisation pour réaliser l'exclusion mutuelle

```
do {

  while (TestAndSet(&lock))
    ; // do nothing

  // critical section
  ...

  lock = FALSE;

  // remainder section
  ...

} while (TRUE);
```

# Instruction matérielle « Swap »

## Définition

```
void
Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

## Utilisation pour réaliser l'exclusion mutuelle

```
do {
  key = TRUE;
  while (key == TRUE)
    Swap(&lock, &key);

  // critical section
  ...

  lock = FALSE;

  // remainder section
  ...

} while (TRUE);
```

# Exclusion mutuelle basée sur « Test & Set »

## Version qui satisfait le critère d'attente bornée

```
// data structures
// initially set to FALSE

boolean waiting[n];
boolean lock;
```

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // critical section
    ...

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    // remainder section
    ...

} while (TRUE);
```

# Attente active et attente passive (1/2)

- **Attente active**
  - Principe des solutions étudiées jusqu'à présent :
    - Une demande de verrou est non bloquante
    - En cas d'échec, un processus réitère sa demande jusqu'à obtenir le verrou
  - Approche très inefficace sur un système monoprocesseur : à éviter systématiquement
  - Sur un système multiprocesseurs : approche potentiellement efficace dans certaines situations
    - Section critique très brève
    - Processeur mobilisé par l'attente active non réutilisable pour une autre activité

# Attente active et attente passive (2/2)

- **Attente passive**
  - Une demande de verrou est bloquante :
    - Si le verrou est disponible, la demande est immédiatement satisfaite
    - Sinon :
      - Le processus demandeur est bloqué (mis en attente)
      - Et sera débloqué lorsque le verrou est disponible (ou, selon les cas, lorsque le verrou a une chance d'être disponible)
  - Solution plus efficace que l'attente active dans la plupart des cas