Cours Programmation Concurrente

Master MIAGE M1

Jean-François Pradat-Peyre,
Lom Hillah
Université Paris Nanterre - UFR SEGMI

2020-2021

4 : Paradigmes de la concurrence

Paradigmes: une définition

- Briques de base pour toute étude, analyse ou construction de système ou d'application coopérative ("concurrency design patterns")
- Exemples type qui permettent de modéliser des classes de problèmes réels fréquemment rencontrés et présents à tous les niveaux dans les systèmes et dans les applications concurrentes.
- Acceptés par la communauté pour leur capacité à fournir des schémas de base de conception
- La solution d'un paradigme est un archétype qui décrit un comportement acceptable pour l'ensemble des processus concurrents dans le cadre du problème à résoudre, et cela quelle que soit la nature des processus.

Principaux paradigmes de la concurrence

- l'*exclusion mutuelle* qui modélise l'accès cohérent à des ressources partagées,
- la *cohorte* qui modélise la coexistence d'un groupe de taille maximale donnée,
- le *passage de témoin* qui modélise la coopération par découpage des tâches entre les processus
- les *producteurs-consommateurs*, exemple qui modélise la communication par un canal fiable,
- les *lecteurs-rédacteurs* exemple qui modélise la compétition cohérente,
- le *repas des philosophes* qui modélise l'allocation de plusieurs ressources et l'interblocage.

Exclusion Mutuelle

Exclusion Mutuelle : Terminologie

- Ressource critique : ressource ne pouvant être utilisée que par un processus à la fois ; par exemple une imprimante, une variable dont l'accès n'est pas atomique, un fichier de base de données, etc.

- Section critique : séquence d'instructions d'un processus à une ressource critique

- Exclusion mutuelle : condition de fonctionnement garantissant à un processus l'accès exclusif à une ressource critique pendant une suite d'opérations avec cette ressource

Exclusion mutuelle : Hypothèses

- H0 : Les vitesses relatives des processus sont quelconques.
- H2 : Les priorités ou les droits des processus sont quelconques.
- H3 : Tout processus sort de sa section critique au bout d'un temps fini ; en particulier ni panne ni blocage perpétuel ne sont permis en section critique.

Exclusion mutuelle : Comportement attendu

- C1 : Un processus au plus en section critique. Peu importe l'ordre d'accès.
- C2 : Pas d'interblocage actif ou passif. Si aucun processus n'est en section critique et que plusieurs processus attendent d'entrer dans leur section critique, alors l'un d'eux doit nécessairement y entrer au bout d'un temps fini. (contrexemple : déclaration et politesse)
- C3 : Un processus bloqué en dehors de section critique, en particulier un processus en panne, ne doit pas empêcher l'entrée d'un autre processus dans sa section critique. (contrexemple: accès à l'alternat)
- C4 : La solution ne doit faire d'hypothèse ni sur les vitesses, ni sur les priorités relatives des processus. De ce point de vue la solution doit être symétrique.

Exclusion mutuelle : Variantes

- V1 : Pas de coalition voulue ou fortuite entraînant la famine d'un processus. Aucun processus qui demande à entrer en section critique ne doit attendre indéfiniment d'y entrer. C'est la propriété d'équité. (Danger présent avec des priorités fixes)

- V2 : Le processus le plus prioritaire du système entre en premier en section critique

- V3 : Pas de règle particulière d'équité ou de temps de réponse. C'est le cas le plus fréquent

Exclusion Mutuelle : Exemple 1

- NICOLAS et ROSE Miagistes à Paris Nanterre, ont un compte bancaire joint : CC et ont chacun une carte bancaire donnant accès à ce compte
- LEUR BANQUE A INSTALLE DES GAB :
  - ✓ C : consultation du compte ; R : retrait avec mise à jour du solde
- OPERATIONS CONCURRENTES AVEC LA BANQUE
  - ✓ C1 : Nicolas consulte CC depuis le GAB Saint-Martin
  - ✓ R1 : Nicolas retire 800 ssi le compte est alimenté (CC >= 800)
  - ✓ C2 : Rose consulte CC depuis le GAB Montgolfier
  - ✓ R2 : Rose retire 600 ssi le compte est alimenté (CC >= 600)
- CONCURRENCE MAL GEREE (sans exclusion mutuelle)
  - ✓ {CC = 1000 } C1; C2; R1; R2 {CC = -400 donc découvert}
  - ✓ {CC = 1000 } C1; C2; R2; R1 {CC = -400 donc découvert}
- CONCURRENCE BIEN GEREE (avec exclusion mutuelle)
  - ✓ {CC = 1000 } C2; R2; C1; R1 {CC = 400 }
  - ✓ {CC = 1000 } C1; R1; C2; R2 {CC = 200 }
- RESSOURCE CRITIQUE : CC
- SECTIONS CRITIQUES POUR ACCES A CC : C1; R1; ou C2; R2

Exclusion Mutuelle : Exemple 2

```
public class Race {
        static double val = 0.0;
```

```
public class Thread_plus extends Thread{

    public void run(){
            for (int i=0; i< 1000000; i++){
                    val += 1.0;
            }
    }
}
```

```
public class Thread_moins extends Thread{
    public void run(){
            for (int i=0; i< 1000000; i++){
                    val -= 1.0;
            }
    }
}
```

```
public void go() throws InterruptedException{
    Thread_plus Tp = new Thread_plus();
    Rhread_moins Tm = new Thread_moins();

    Tp.start();
    Tm.start();
    Tp.join();
    Tm.join();
    System.out.println(val);

}
public static void main(String[] args) throws InterruptedException{
    Race r = new Race();
    r.go();
}
```

Les deux tâches s'exécutent en parallèle

Instructions non atomiques

Le résultat affiché par ce programme varie entre -1 000 000 et +1 000 000 !

Exclusion Mutuelle : Archétype avec sémaphores

Données communes aux processus

S : Semaphore ; *// possibilité d'utiliser un Mutex*
E0(S, 1) ; *// accès possible initialement; au plus 1 en SC*

Code des processus

```
P(S);
    Section Critique ;
V(S);
```

Exclusion Mutuelle : Archétype moniteurs Java

```
synchronized type_res m (paramètres …){
        Code en section critique ;
}
```

L'accès à la méthode m d'une instance o sera fait en exclusion mutuelle avec tout accès sur la même instance o de code marqué « synchronized »

o est un objet partagé entre les threads

```
synchronized (o){
        Code en section critique ;
}
```

Exclusion Mutuelle : Exemple 2 corrigé

```
public class Race {

    static double val = 0.0;
    Object o = new Object();

public class Thread_plus extends Thread{

    public void run(){
            for (int i=0; i< 1000000; i++){
                synchronized(o){val += 1.0;}
            }
    }
}

public class Thread_moins extends Thread{

    public void run(){
            for (int i=0; i< 1000000; i++){
                synchronized(o){val -= 1.0;}
            }
    }
}

public void go() throws InterruptedException{
        Thread_plus Tp = new Thread_plus();
        Thread_moins Tm = new Tthread_moins();

        Tp.start();
        Tm.start();
        Tp.join();
        Tm.join();
        System.out.println(val);

    }
    public static void main(String[] args) throws InterruptedException{
        Race r = new Race();
        r.go();
    }
}
```

Les deux tâches s'exécutent en parallèle

Instructions en section critique

Le résultat affiché par ce programme est toujours 0

La Cohorte

La Cohorte : comportement attendu

- N processus au plus, N fixé, peuvent coopérer de façon asynchrone pour :
  - se répartir une tâche ou un service à fournir
  - partager une ressource banalisée en N exemplaires

La Cohorte : hypothèses

- H1: Les vitesses relatives des processus sont quelconques.
- H2: Les priorités ou les droits des processus sont quelconques.
- H3: Tout processus quitte la cohorte au bout d'un temps fini.

La Cohorte : archétype avec sémaphores

Données communes aux processus

```
S : Semaphore ;  // IMPOSSIBILITE d'utiliser un Mutex
E0(S, N) ; // N processus en cohorte
```

Code d'un processus

```
P(S);
   Action en cohorte ;
V(S);
```

La Cohorte : archétype avec moniteurs Java

```
public class Cohorte {
        protected int compteur = 0;

        Cohorte(int N){
                this.compteur = N;
        }

        public synchronized void entreCohorte(){
                while (this.compteur == 0)
                        try {
                                wait();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                this.compteur --;
        }

        public synchronized void sortCohorte(){
                this.compteur ++;
                if (this.compteur ==  1)
                        notify();
                }
        }
}
```

Appel pour entrer

Appel pour sortir

Passage de témoin

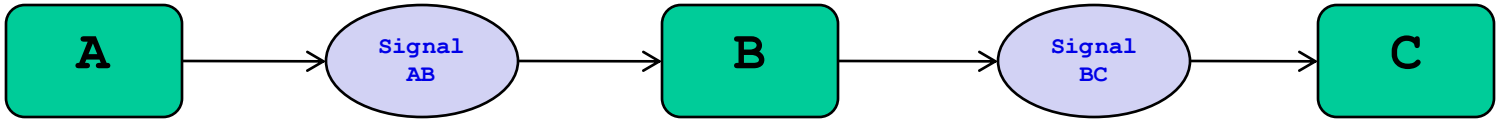
Passage de témoin

- Coopération par division du travail entre les processus
- Envoi d'un signal, témoin de la fin d'une action
- Signaux mémorisés ou non. Point à point ou diffusion

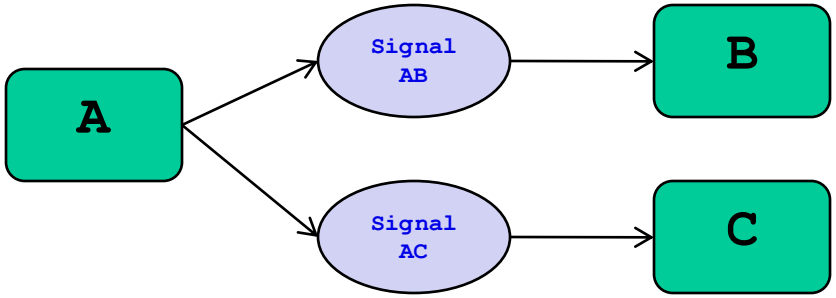
- *Différentes variantes :*
  - *séquences d'actions A --> B --> C*
  - *précédences : A --> B || C*
  - *rendez-vous symétrique*
  - *appel de procédure à distance synchrone ou asynchrone*

Passage de témoin (suite)

Séquence

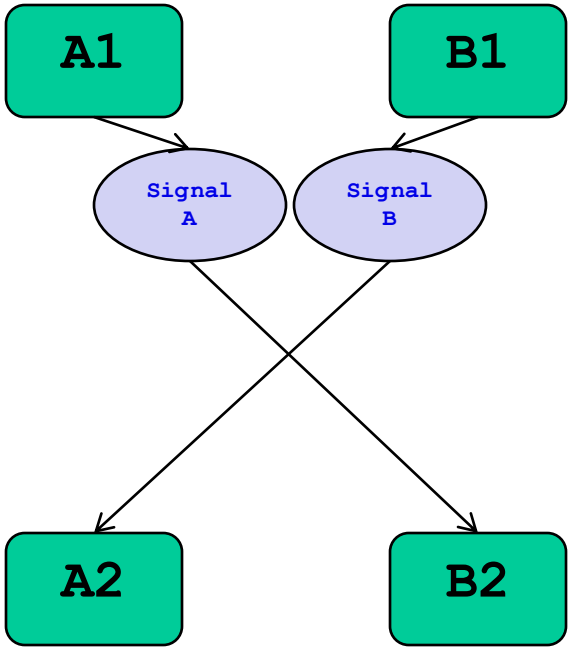


Précédence

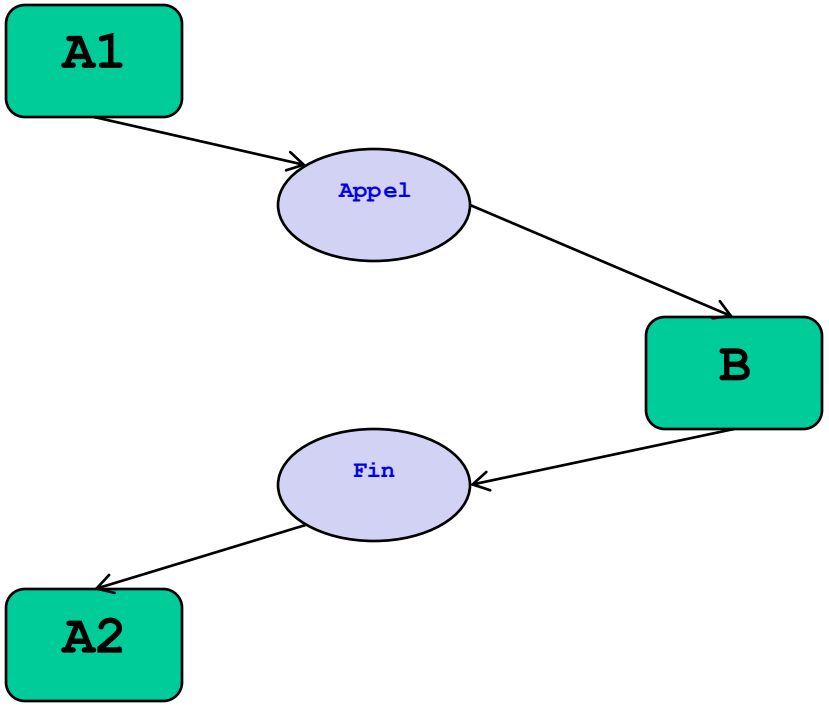


Passage de témoin (suite)

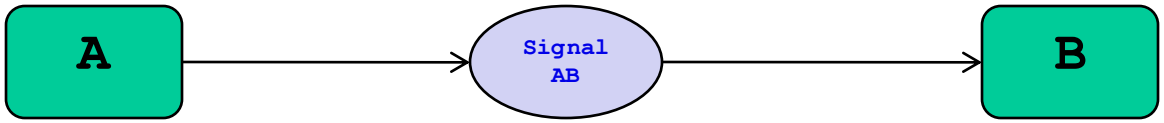
Rendez-vous



RPC synchrone



Passage de témoin : Archétype de base

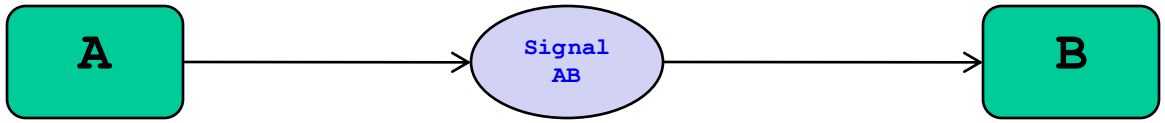



```
S_AB : Semaphore_binaire ;
E0(S_AB, 0);
```

```
V(S_AB);
```

```
P(S_AB);
```

Passage de témoin : Archétype de base




```
public class Signal {
  // classe signal sans comptage
 boolean present = false;

 public synchronized void sendSig(){
    this.present = true;
    notify();
 }
 public synchronized void waitSig(){
    while (! this.present){
    try {
       wait();
    } catch (InterruptedException e) {
    e.printStackTrace();
    }
 }
 this.present = false;
 }
}
```

```
Signal S_AB = new signal();
```

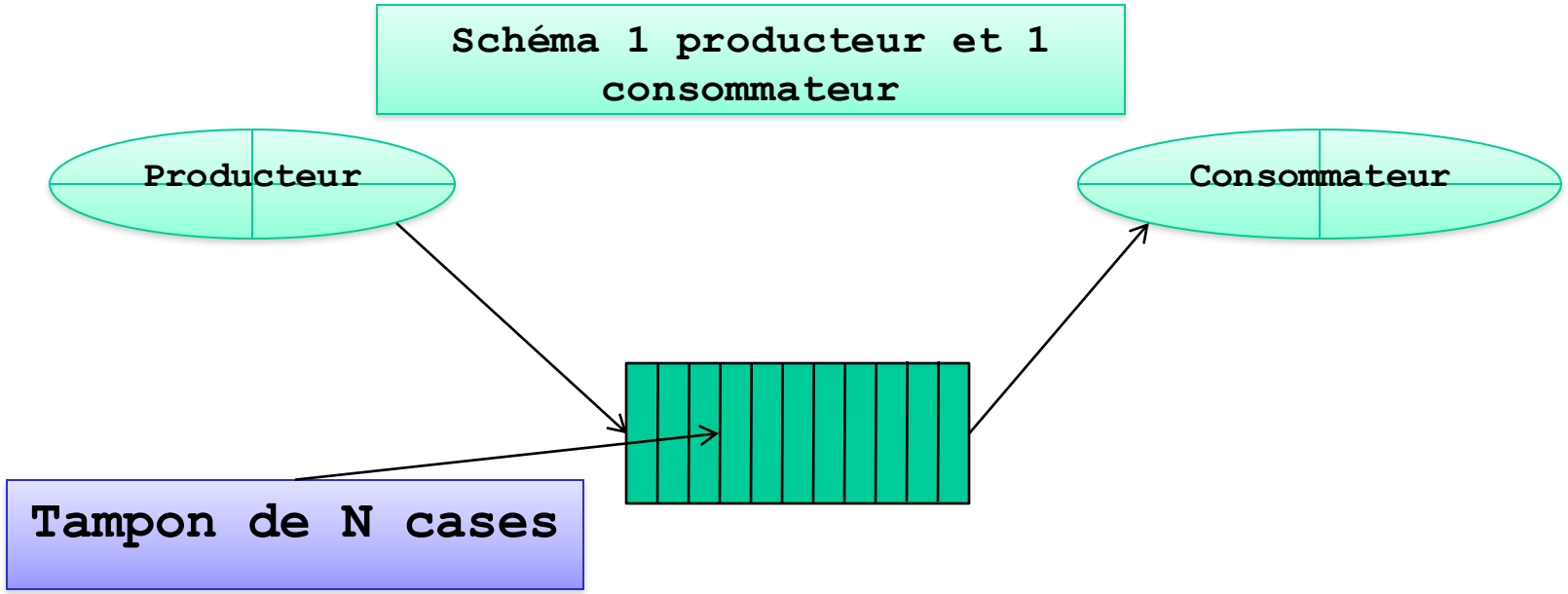
```
S_AB.sendSig();
```

```
S_AB.waitSig();
```

Producteur(s) Consommateur(s)

Producteur consommateur : Principes

- Deux classes de processus : les producteurs et les consommateurs
- Les producteurs produisent des données / messages qui seront consommées par les consommateurs
- Les données produites sont stockées dans un tampon de N cases

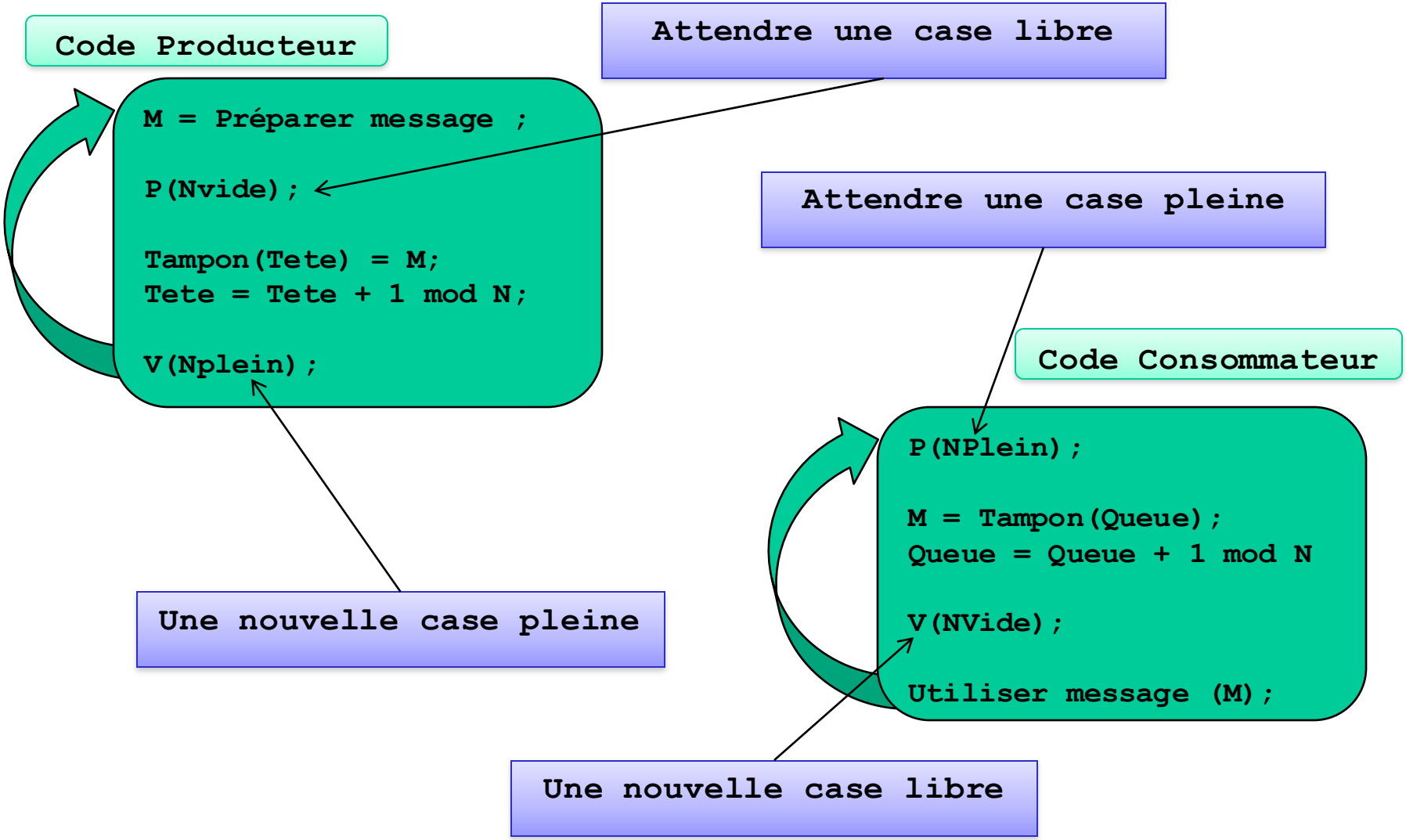


Producteur consommateur : Hypothèses

- H1 : les vitesses relatives des processus sont quelconques
- H2 : les priorités des processus sont quelconques
- H3 : tout processus met un temps fini pour déposer ou retirer un message ; en particulier pas de panne pendant ces actions

→ Débit irrégulier : il faut asservir le rythme moyen du producteur au rythme moyen du consommateur; le tampon joue ce rôle

Producteur consommateur : Archétype avec sémaphores



Producteur consommateur : Archétype avec moniteurs Java

public class ProdCons {

Attendre une case libre

Une nouvelle case pleine

```
private Object buffer[];
private int sizeMax;
private int lire, ecrire;
private int nbElmt;

ProdCons(int N){
   this.buffer = new Object[N];
   this.sizeMax = N;
   this.lire = this.ecrire = this.nbElmt = 0;
}
```

```
void synchronized Put(Object m){
while (this.nbElmt == this.sizeMax)
try {
wait();
} catch (InterruptedException e) {
e.printStackTrace();
}
this.buffer[this.ecrire++] = m;
notify();
this.ecrire %= this.sizeMax;
}
void synchronized Object Get(){
while (this.nbElmt == 0)
try {
wait();
} catch (InterruptedException e) {
e.printStackTrace();
}
Object m = this.buffer[this.lire++];
this.lire %= this.sizeMax;
notify();
return m;
}
```

Une nouvelle case libre

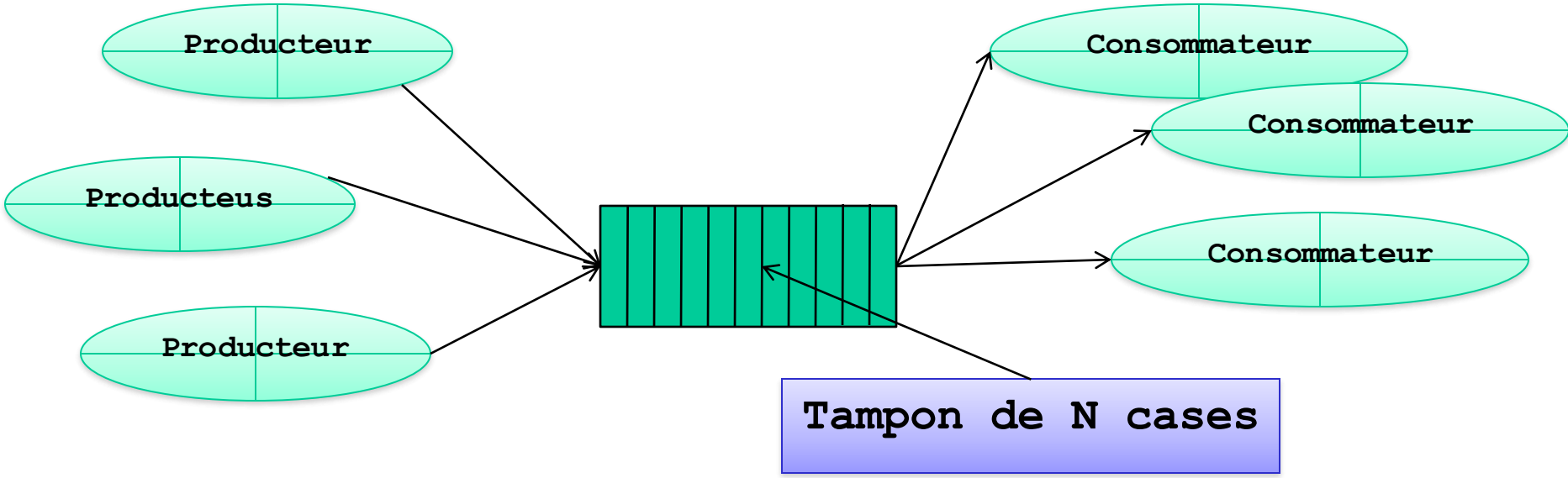
Attendre une case pleine

}

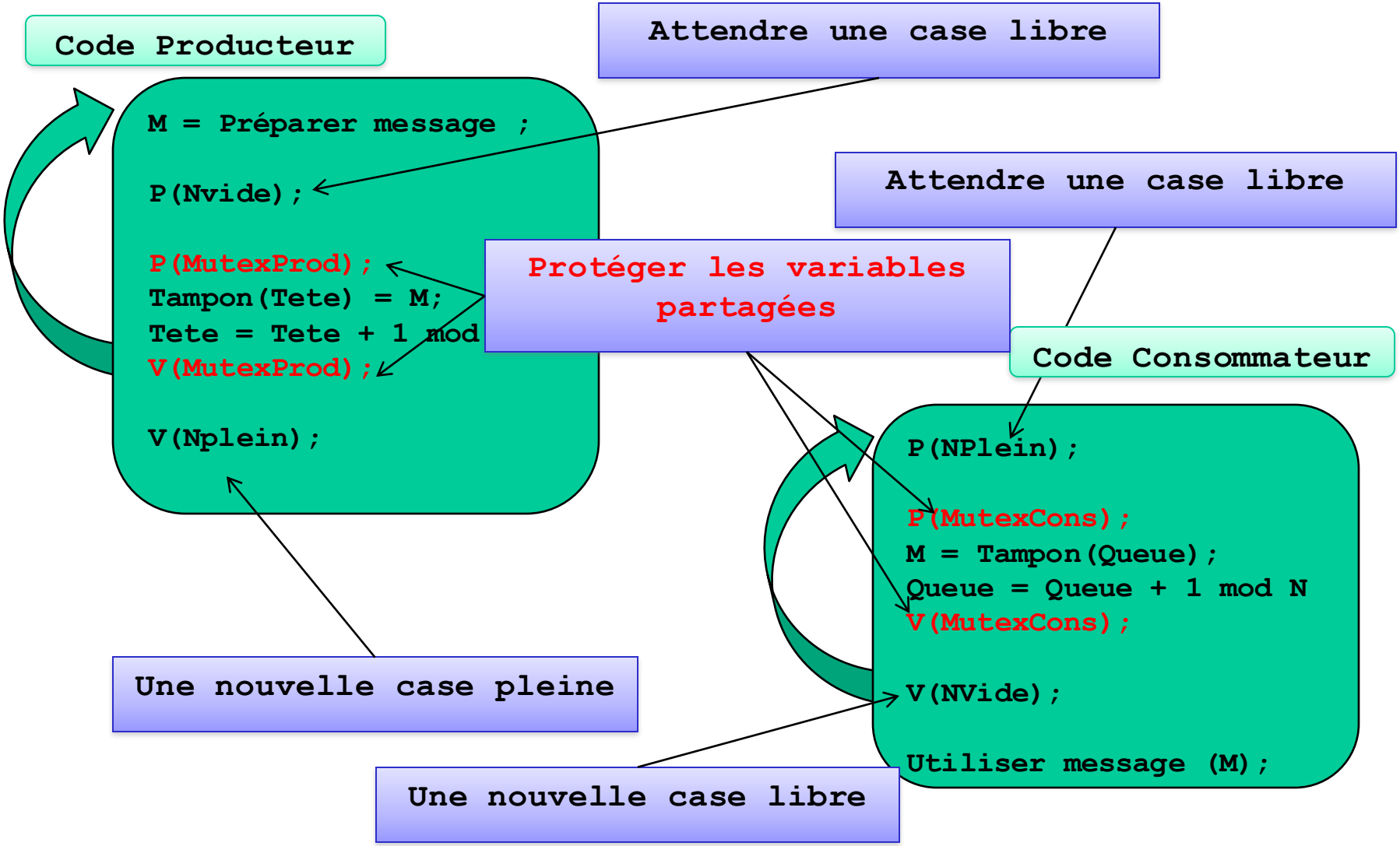
Producteurs consommateurs : Principes

- Même principes et même hypothèses que dans le cas où il n'y a qu'un producteur et un consommateur

- Gestion de la concurrence d'accès entre les producteurs et entre les consommateurs



Producteurs consommateurs : Archétype avec sémaphores



Producteurs consommateurs : Archétype avec moniteurs Java

public class ProdCons {

Attendre une case libre

Une nouvelle case pleine

```
private Object buffer[];
private int sizeMax;
private int lire, ecrire;
private int nbElmt;

ProdCons(int N){
   this.buffer = new Object[N];
```

```
public synchronized Put(Object m){
while (this.nbElmt == this.sizeMax)
try {
wait();
} catch (InterruptedException e) {
e.printStackTrace();
}
this.buffer[this.ecrire++] = m;
notify(); nbElmt++
this.ecrire %= this.sizeMax;
}
public synchronized Object Get(){
while (this.nbElmt == 0)
try {
wait();
} catch (InterruptedException e) {
e.printStackTrace();
}
Object m = this.buffer[this.lire++];
this.lire %= this.sizeMax;
notify(); nbElmt --;
return m;
}
```

Grace aux moniteurs Java les variables partagées sont déjà protégées

Une nouvelle case libre

Attendre une case pleine

}

Lecteurs Rédacteurs

Lecteurs Rédacteurs

- Compétition d'accès à un ensemble de données par un ensemble de processus
  - lecteurs accès seulement en lecture
  - rédacteurs accès en lecture et écriture
- Objectif : Garantir la cohérence des données
- Spécification
  - plusieurs lectures simultanément
  - les écritures sont en exclusion mutuelle
- Hypothèses
  - les vitesses relatives des processus sont quelconques
  - pas de panne en section critique

Lecteurs Rédacteurs: Archétype avec sémaphores

Rédacteurs

```
P(Mutex_Glob) ;

  Ecrire ;

V(Mutex_Glob) ;
```

Lecteurs

```
P(S_Att_Lire);
If (Nb_L == 0)
   P(Mutex_Glob) ;
Nb_L ++;
V(S_Att_Lire);

   Lire;

P(S_Att_Lire);
Nb_L --;
If (Nb_L == 0)
   V(Mutex_Glob) ;
V(S_Att_Lire);
```

Le premier lecteur verrouille pour sa classe

Le dernier lecteur déverrouille

Lecteurs Rédacteurs: Archétype en Java

```
public class LecteursRedacteurs {

private int nbLecteurs = 0;
private boolean redacteurPresent = false;
```

Le premier lecteur verrouille pour sa classe

```
synchronized void entreLecture() {
while (this.redacteurPresent)
try {
wait();
} catch (InterruptedException e) {
e.printStackTrace();
}
nbLecteurs++;
}
synchronized void sortLecture() {
nbLecteurs--;
notify();
}
```

```
synchronized void entreEcriture() {
while ((this.redacteurPresent)|| (this.nbLecteurs > 0))
try {
wait();
} catch (InterruptedException e) {
e.printStackTrace();
}
this.redacteurPresent = true;
}
synchronized void sortEcriture(){
this.redacteurPresent = false;
notify();
}
}
```

barrière

Le dernier lecteur déverrouille