Cinquième chapitre

Synchronisation entre processus

Problème de l'exclusion mutuelle

Exemple : deux banques modifient un compte en même temps

Agence Nancy

```
1. courant = get_account(1867A)
2. nouveau = courant + 10
3. update_account (1867A, nouveau)
```

Agence Karlsruhe

```
1. aktuelles = get_account(1867A)
2. neue = aktuelles - 10
3. update_account(1867A, neue)
```

- variables partagées + exécutions parallèles entremêlées ⇒ différents résultats :
- (0 ; ? ; ?) N1(0 ; 0 ; ?) N2(0 ;10 ; ?) N3(10 ;10 ; ?)
  K1(10 ;10 ;10)K2(10 ;10 ;0) K3(0 ;10 ; 0) → compte inchangé
- (0 ; ? ; ?)N1 (0 ;0 ; ?) K1 (0 ;0 ;0) N2(0 ;10 ;0)
  K2(0 ;10 ;-10) N3(10 ;10 ;-10) K3(-10 ;10 ;-10) → compte −= 10
- (0 ; ? ; ?)K1 (0 ; ? ;0) N1 (0 ;0 ;0) K2(0 ;0 ;-10)
  N2(0 ;10 ;-10) K3(-10 ;10 ;-10)N3(10 ;10 ;-10) → compte += 10

C'est une **condition de compétition** (*race condition*)

- Solution : opérations **atomiques** ; pas d'exécutions entremêlées
- Cette opération est une **section critique** à exécuter en **exclusion mutuelle**

Réalisation d'une section critique

Schéma général

Processus 1

```
...
entrée en section critique
    section critique
sortie de section critique
...
```

Processus 2

```
...
entrée en section critique
    section critique
sortie de section critique
...
```

- Exclusion mutuelle garantie par les opérations `(entrée en section critique)` et `(sortie de section critique)`

Réalisation

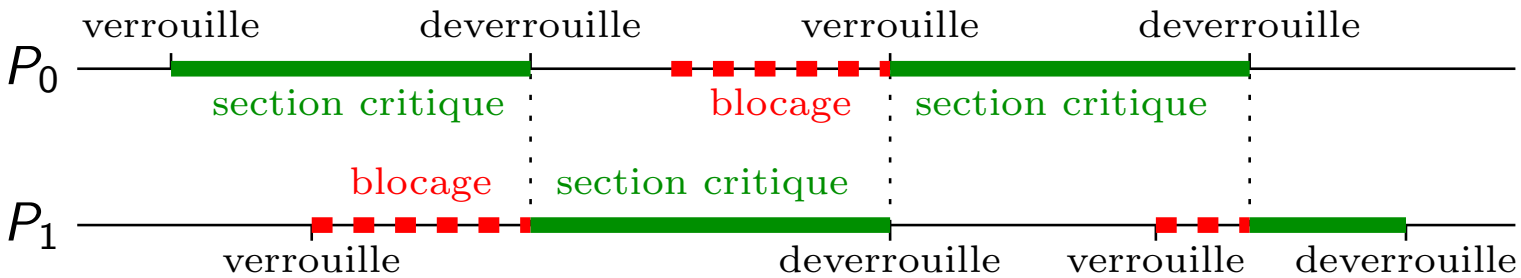
- Attente active : processus à l'entrée section critique boucle un test d'entrée
  - **Inefficace** (sur mono-processeur)
  - Parfois utilisé dans conditions praticulière dans le noyau
- Primitives spéciales : fournies par le système
  - Primitives générales : sémaphores, mutex (on y revient)
  - Mécanismes spécifiques : comme verrouillage de fichiers (idem)
  - Les primitives doivent être atomiques...

Exclusion mutuelle par verrouillage de fichiers

- Verrouiller une ressource garanti un accès exclusif à la ressource
- Unix permet de verrouiller les fichiers
- Appels systèmes de verrouillage/déverrouillage (*noms symboliques*) :
  - `verrouille(fich)` : verrouille `fich` avec accès exclusif
  - `déverrouille(fich)` : déverrouille `fich`
- Propriétés :
  - Opérations atomiques (assuré par l'OS)
  - Verrouillage par au plus un processus
  - Tentative de verrouillage d'un fichier verrouillé ⇒ blocage du processus
  - Deverrouillage ⇒ réveille d'un processus en attente du verrou (et un seul)

```
...
verrouille(fich);
accès à fich (section critique);
deverrouille(fich);
...
```

```
...
verrouille(fich);
accès à fich (section critique);
deverrouille(fich);
...
```



Verrouillage partagé

- Sémantique sur les fichiers : écritures exclusives, lectures concurrentes
- Nouvelle opération : ver-part (verrouillage partagé)

Déjà réalisé	Nouveau verrouille possible ?	Nouveau ver-part possible ?
verrouille(f)	non	non
ver-part(f)	non	oui

- Tentative de verrouillage d'un fichier verrouillé $\Rightarrow$ blocage du processus
- Déverrouillage $\Rightarrow$ réveille d'un processus en attente du verrou (et un seul)

Verrouillage de fichier sous Unix

Opérations disponibles

- Deux types de verrous : exclusifs ou partagés
- Deux modes de verrouillage :
  - Impératif (*mandatory*) : **bloque les accès** incompatibles avec verrous présents (mode décrit précédemment – pas POSIX)
  - Consultatif (*advisory*) : ne **bloque que la pose** de verrous incompatibles
  - Tous les verrous d'un fichier sont de même mode
- Deux primitives :
  - fcntl : générale et complexe
  - lockf : enveloppe plus simple, mais mode exclusif seulement (pas partagé)
- Possibilité de verrouiller fichier entier ou une partie

Interface du verrouillage de fichier sous Unix

```
#include <unistd.h>
int lockf(int fd, int commande, off_t taille);
```

- `fd` : descripteur du fichier à verrouiller
- `commande` : mode de verrouillage
  - F_ULOCK : déverrouiller
  - F_LOCK : verrouillage exclusif
  - F_TLOCK : verrouillage exclusif avec test (ne bloque jamais, retourne une erreur)
  - F_TEST : test seulement
- `taille` : spécifie la partie à verrouiller
  - $= 0$: fichier complet
  - > 0 : `taille` octets après position courante
  - < 0 : `taille` octets avant position courante
- Verrouillage consultatif et exclusif seulement
  Utiliser fcntl pour verrouillage impératif et/ou partagé
- Retour : 0 si succès, -1 sinon (*cf.* `errno`)

Exemple de verrouillage de fichiers Unix

testlock.c

```
#include <unistd.h>
int main (void) {
  int fd;
  fd = open("toto", O_RDWR);  /* doit exister */
  while (1) {
    if (lockf(fd, F_TLOCK, 0) == 0) {
      printf("%d: verrouille le fichier\n",
             getpid());
      sleep(5);
      if (lockf(fd, F_ULOCK, 0) == 0)
        printf("%d: déverrouille le fichier\n",
               getpid());
      return;
    } else {
      printf("%d: déjà verrouillé...\n",
             getpid());
      sleep (2);
    }
  }
}
```

```
$ testlock & testlock &
15545: verrouille le fichier
[4] 15545
15546: déjà verrouillé...
[5] 15546
$ 15546: déjà verrouillé...
15546: déjà verrouillé...
15545: déverrouille le fichier
15546: verrouille le fichier
15546: déverrouille le fichier
$
```

Exercice : que se passe-t-il si on remplace F_TLOCK par F_LOCK ?

15546 bloque jusqu'à ce que 15545 déverrouille le fichier
(et n'affiche rien entre temps)

L'exemple des banques revisité

Rappel du problème

Agence Nancy

```
1. courant = get_account(1867A)
2. nouveau = courant + 10
3. update_account (1867A, nouveau)
```

Agence Karlsruhe

```
1. aktuelles = get_account(1867A)
2. neue = aktuelles - 10
3. update_account(1867A, neue)
```

Implémentation avec le verrouillage de fichier UNIX

Agence Nancy

```
   int fd = open("fichier partagé", O_RDWR);
   lockf(fd, T_LOCK, 0);
1. courant = get_account(1867A)
2. nouveau = courant + 10
3. update_account (1867A, nouveau)
   lockf(fd, T_ULOCK, 0);
```

Agence Karlsruhe

```
   int fd = open("fichier partagé", O_RDWR);
   lockf(fd, T_LOCK, 0);
1. aktuelles = get_account(1867A)
2. neue = aktuelles - 10
3. update_account(1867A, neue)
   lockf(fd, T_ULOCK, 0);
```

Remarques

- Fichier partagé entre deux banques difficile à réaliser
- Autres solutions techniques pour les systèmes distribués ; l'idée reste la même

Notion d'interblocage

Utilisation simultanée de plusieurs verrous ⇒ problème potentiel

Situation

- Deux processus verrouillent deux fichiers

Processus 1

```
...
verrouille (f1) /* 1A */
accès à f1
...
verrouille (f2) /* 1B */
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

Processus 2

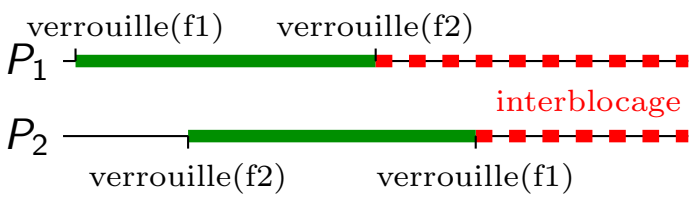
```
...
verrouille (f2) /* 2A */
accès à f2
...
verrouille (f1) /* 2B */
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

Déroulement

Exécution (pseudo-)parallèle

- Première possibilité : 1a ; 1b ; 2a ; 2b
- Seconde possibilité : 2a ; 2b ; 1a ; 1b
- Troisième possibilité : 1a ; 2a ; 1b ; 2b

Exécution de 1a ;2a ;1b ;2b



- P1 et P2 sont bloqués *ad vitam eternam* :
  - P1 attend le `deverrouille(f2)` de P2
  - P2 attend le `deverrouille(f1)` de P1
- C'est un **interblocage** (*deadlock*)

Situation d'interblocage

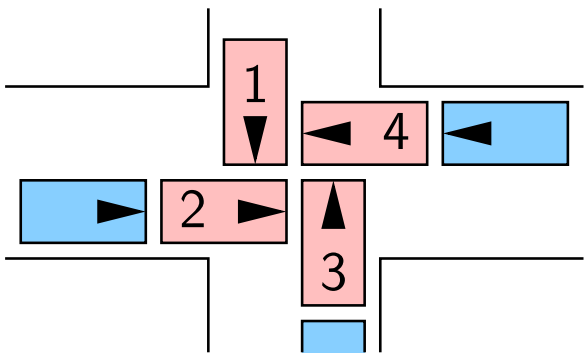
Définition

- Plusieurs processus bloqués dans l'attente d'une action de l'un des autres
- Impossible de sortir d'un interblocage sans intervention extérieure

Conditions d'apparitions

- Plusieurs processus en compétition pour les mêmes ressources
- Cycle dans la chaîne des attentes

Exemple : carrefour un vendredi à 18h



Exercice : quelles sont les ressources ?

Exercice : comment sortir de l'interblocage ?

Situation réelle d'interblocage

Comment prévenir l'interblocage ?

Solution 1 : réservation globale

- Demandes en bloc de toutes les ressources nécessaires
- Inconvénient : réduit les possibilités de parallélisme
- Analogie du carrefour : damier jaune

Solution 2 : requêtes ordonnées

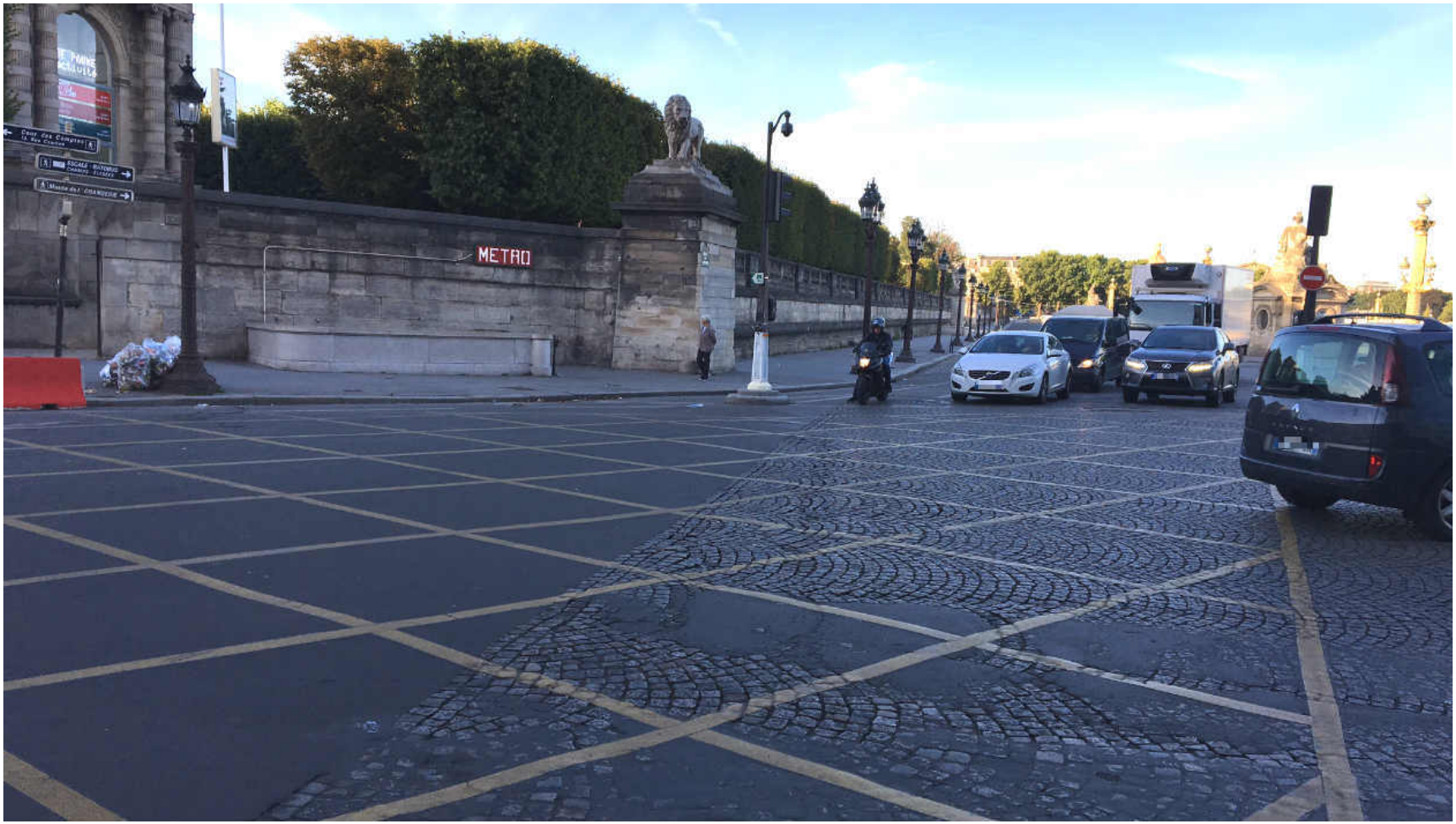
- Tous les processus demandent les ressources dans le même ordre
- Interblocage alors impossible
- Analogie du carrefour : construire un rond-point

```
verrouille (f1)
verrouille (f2)
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

```
verrouille (f1)
verrouille (f2)
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

Solution 3 : modification de l'algorithme

- Modifier code utilisateur pour rendre impossible l'interblocage
- Analogie du carrefour : construire un pont au dessus du carrefour


METRO

Sortir de l'interblocage (quand on a pas su prévenir)

Impossible sans perdre quelque chose

Possibilités de guérison

- Faire revenir un processus en arrière
  - Nécessite d'avoir un point de reprise (*checkpoint*)
  - Travail depuis dernier point de reprise perdu
- Tuer l'un des processus pour casser le cycle

Conclusion

- Prévention et guérison sont tous les deux coûteux
  - Prévention : l'application doit faire attention
  - Guérison : détection + pertes dues au retour en arrière
- La ≪meilleure≫ solution dépend de la situation

Section critique par attente active

Principe (rappel)

- On demande tant qu'on a pas obtenu

Défaut

- Manque d'efficacité car gaspillage de ressources

Avantage

- Implémentable sans primitive de l'OS ni du matériel

$\Rightarrow$ utilisé dans certaines parties de l'OS, par exemple

Attention, ce n'est pas si simple à faire

Réalisation d'une section critique par attente active

- Hypothèses : atomicité d'accès et de modifications de variables
  V ne change pas au milieu de V==1 ni de V=1 (raisonnable sur monoprocesseur)

Solution **FAUSSE** numéro 1

```
while (tour != MOI);  /* attente active */
SC();                 /* section critique */
tour = 1-MOI;         /* soyons fair-play */
```

Demande alternance stricte :
Entrer deux fois de suite dans SC()
impossible (même si seul processeur)
C'est une **famine**

Solution **FAUSSE** numéro 2

```
while(flag[1-MOI]);     /*attendre autre*/
flag[MOI] = VRAI;       /*annonce entrer*/
SC();
flag[MOI] = FAUX;       /*débloque autre*/
```

Pas d'exclusion mutuelle (**compétition**) :
P0 teste flag[1] et trouve faux
P1 teste flag[0] et trouve faux
Les deux entrent dans SC()

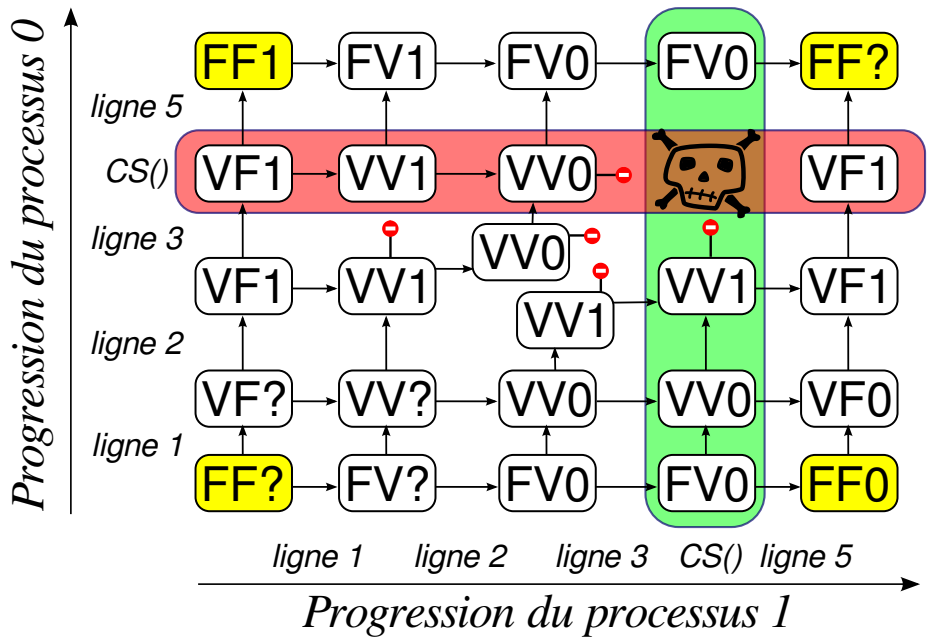
Solution **FAUSSE** numéro 3

```
flag[MOI]=VRAI;   /*annonce entrer avant*/
while (flag[1-MOI]);  /*attendre l'autre*/
SC();
flag[MOI] = FAUX;       /*débloque autre*/
```

Possibilité d'**interblocage** :
P0 : flag[0] $\leftarrow$ VRAI
P1 : flag[1] $\leftarrow$ VRAI
Les deux entrent dans leur boucle

Algorithme correct d'attente active

```
1. flag[MOI] = VRAI;                                  /* Annonce être intéressé */
2. tour = 1-MOI;                                /* mais on laisse la priorité à l'autre */
3. while ((flag[1-MOI]==VRAI)  /* on entre si l'autre ne veut pas entrer */
          && (tour == 1-MOI));      /* ou s'il nous a laissé la priorité */
4. CS();
5. flag[MOI] = FAUX;                                                    /* release */
```



GL Peterson. *A New Solution to Lamport's Concurrent Programming Problem*. 1983. (lire l'article)

`http://en.wikipedia.org/wiki/Peterson's_algorithm`

- Ceci est un diagramme de transition (rarement aussi régulier)

Problèmes de synchronisation (résumé)

- Condition de **compétition** *(race condition)*
  - Définition : le résultat change avec l'ordre des instructions
  - Difficile à corriger car difficile à reproduire (ordre ≪aléatoire≫)
  - Également type de problème de sécurité :
    - Un programme crée un fichier temporaire, le remplit puis utilise le contenu
    - L'attaquant crée le fichier avant le programme pour contrôler le contenu
- **Interblocage** *(deadlock)*
  - Définition : un groupe de processus bloqués en attente mutuelle
  - Évitement parfois difficile (correction de l'algorithme)
  - Détection assez simple, mais pas de guérison sans perte
- **Famine** *(starvation)*
  - Définition : un processus attend indéfiniment une ressource (problème d'équité)
  - Servir équitablement les processus demandeurs

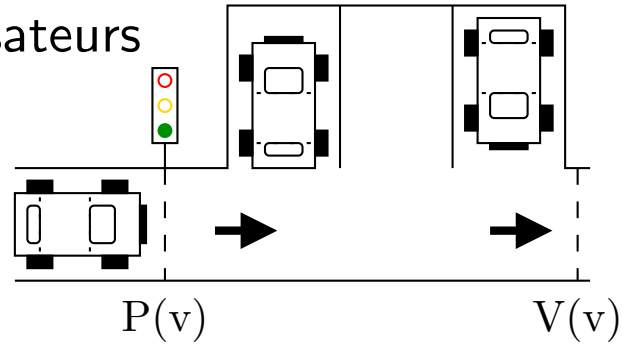
Cinquième chapitre

Synchronisation entre processus

Sémaphore : outil de base pour la synchronisation

Cohorte : généralisation de l'exclusion mutuelle

- Ressource partagée par au plus N utilisateurs
- Exemples :
  - Parking payant
  - Serveur informatique



Sémaphore

- Objet composé :
  - D'une variable : valeur du sémaphore (nombre de places restantes)
  - D'une file d'attente : liste des processus bloqués sur le sémaphore
- Primitives associées :
  - Initialisation (avec une valeur positive ou nulle)
  - Prise (P, Probeer, down, wait) = demande d'autorisation (≪puis-je ?≫)
    Si *valeur* = 0, blocage du processus ; Si non, *valeur* = *valeur* − 1
  - Validation (V, Verhoog, up, signal) = fin d'utilisation (≪vas-y≫)
    Si *valeur* = 0 et processus bloqué, déblocage d'un processus ;
    Si non, *valeur* = *valeur* + 1

E.W. Dijkstra. *Solution of a Problem in Concurrent Programming Control*. 1965. (lire l'article)

Schémas de synchronisation

Situations usuelles se retrouvant lors de coopérations inter-processus

- Exclusion mutuelle : ressource accessible par une seule entitée à la fois
  - Compte bancaire ; Carte son
- Problème de cohorte : ressource partagée par au plus N utilisateurs
  - Un parking souterrain peut accueillir 500 voitures (pas une de plus)
  - Un serveur doom peut accueillir 2000 joueurs
- Rendez-vous : des processus collaborant doivent s'attendre mutuellement
  - Roméo et Juliette ne peuvent se prendre la main que s'ils se rencontrent
  - Le GIGN doit entrer en même temps par le toit, la porte et la fenêtre
  - Processus devant échanger des informations entre les étapes de l'algorithme
- Producteurs/Consommateurs : un processus doit attendre la fin d'un autre
  - Une Formule 1 ne repart que quand tous les mécaniciens ont le bras levé
  - Réception de données sur le réseau *puis* traitement
- Lecteurs/Rédacteurs : notion d'accès exclusif entre *catégories* d'utilisateurs
  - Sur une section de voie unique, tous les trains doivent rouler dans le même sens
  - Un fichier pouvant être lu par plusieurs, si personne ne le modifie
  - Tâches de maintenance (défragmentation) quand pas de tâches interactives

Comment résoudre ces problèmes avec les sémaphores ?

Exclusion mutuelle par sémaphore

Très simple

Initialisation

```
sem=semaphore(1)
```

Agence Nancy

```
P(sem)
courant = get_account(1867A)
nouveau = courant + 1000
update_account (1867A, nouveau)
V(sem)
```

Agence Karlsruhe

```
P(sem)
aktuelles = get_account(1867A)
neue = aktuelles - 1000
update_account(1867A, neue)
V(sem)
```

Cohortes et sémaphores

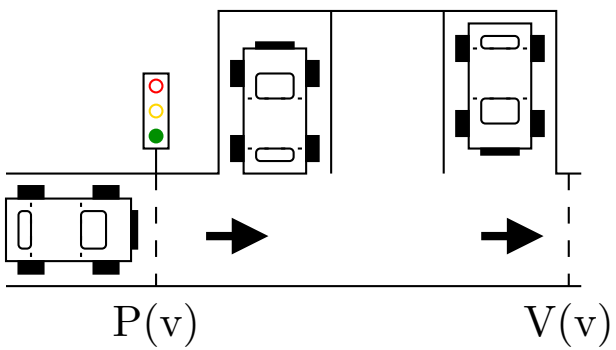
Sémaphore inventée pour cela

Initialisation

```
sem=semaphore(3)    /* nombre de places */
```

Garer sa voiture

```
P(sem)
poser sa voiture au parking
aller faire les courses
Reprendre la voiture
V(sem)
partir
```



Exercice : quelle est la valeur de *sem.v* ici ?

Rendez-vous et sémaphores

- Envoi de signal
  - Un processus indique quelque chose à un autre (disponibilité donnée)

Initialisation

```
top=semaphore(0)
```

Processus 1

```
...
calcul(info)
V(top)
...
```

Processus 2

```
...
P(top)   /*Bloque en attente*/
utilisation(info)
...
```

  - `top` = sémaphore privée (initialisée à 0)
    utilisée pour synchro *avec quelqu'un*, pas sur une ressource
- Rendez-vous entre deux processus
  - Les processus s'attendent mutuellement

Initialisation

```
roméo=semaphore(0)
juliette=semaphore(0)
```

Processus romeo

```
P(romeo)        /*se bloque*/
V(juliette)     /*libère J*/
```

Processus juliette

```
V(romeo)          /*libère R*/
P(juliette)        /*bloque*/
```

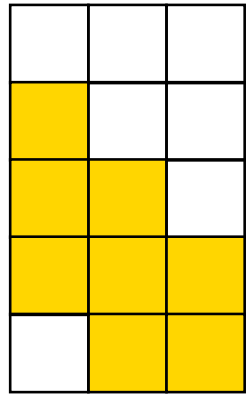
- Rendez-vous entre trois processus et plus
  - On parle de **barrière de synchronisation**
  - La solution précédente est généralisable, mais un peu lourde
  - Souvent une primitive du système

Producteurs et consomateurs

Contrôle de flux entre producteur(s) et consommateur(s) par tampon

Principe

- Situation initiale : tampon vide (pas de lecture possible)
- Après une production :
- Une autre production :
- Encore une production : (plus de production possible)
- Une consommation : (production de nouveau possible)



Réalisation

Initialisation

```
placeDispo=semaphore(N)
infoPrete=semaphore(0)
```

Producteur

```
répéter
  calcul(info)
  P(placeDispo)
  déposer(info)
  V(infoPrete)
```

Consommateur

```
répéter
  P(infoPrete)
  extraire(info)
  V(placeDispo)
  utiliser(info)
```

- Le tampon doit être circulaire pour traiter données dans l'ordre de production
- Attention aux compétitions entre producteurs (idem entre consommateurs)

Lecteurs et rédacteurs

Contrôle d'accès exclusif entre *catégories* d'utilisateurs

- Accès simultané de plusieurs lecteurs
- Accès exclusif d'un seul rédacteur, également exclusif de tout lecteur
- Schéma assez classique (fichier sur disque, zone mémoire, *etc.*)

Première solution

Initialisation

```
lecteur=semaphore(1)
rédacteur=semaphore(1)
NbLect=0
```

Lecteur

```
P(lecteur)
  NbLect = NbLect + 1
  si NbLect == 1 alors
    P(rédacteur)
V(lecteur)
lectures()
P(lecteur)
  NbLect = NbLect - 1
  si NbLect == 0 alors
    V(rédacteur)
V(lecteur)
```

Rédacteur

```
P(rédacteur)
  lecturesEtEcritures()
V(rédacteur)
```

- Problème : famine potentielle des rédacteurs

Lecteurs et rédacteurs sans famine

Initialisation

```
lecteur=semaphore(1)
rédacteur=semaphore(1)
fifo=semaphore(1)
NbLect=0
```

Lecteur

```
P(fifo)
  P(lecteur)
    NbLect = NbLect + 1
    si NbLect == 1 alors
      P(rédacteur)
  V(lecteur)
V(fifo)
lectures()
P(lecteur)
  NbLect = NbLect - 1
  si NbLect == 0 alors
    V(rédacteur)
V(lecteur)
```

Rédacteur

```
P(fifo)
  P(rédacteur)
V(fifo)
  lecturesEtEcritures()
V(rédacteur)
```

Exercice : pourquoi cette nouvelle sémaphore empêche la famine des rédacteurs ?

Exercice : montrer que les lecteurs peuvent encore partager l'accès

Cinquième chapitre

Synchronisation entre processus

Moniteur

Problème des sémaphores

- **Tous les processus doivent les utiliser correctement**
- Mauvais comportement d'un seul ⇒ problème pour l'ensemble
  - Oubli d'un P(mutex) : CS pas respectée
  - Oubli d'un V(mutex) : deadlock (Deni de Service – DoS)
- Causes possibles :
  - Erreur de programmation
  - Programme malveillant

Solution : le moniteur (`synchronized` en Java)

- Sorte d'objet contenant :
  - Des variables partagées (privées)
  - Un sémaphore protecteur
  - Des méthodes d'accès
- Le compilateur ajoute les appels au sémaphore pour protéger les méthodes
- Erreur impossible car usage seulement à travers l'API protégée

C.A.R. Hoare. *Monitors: An Operating System Structuring Concept.* 1974. (lire l'article)

L'exemple des banques avec un moniteur Java

```
public class compteBanquaire {
  private int balance;
  compteBanquaire() {
     balance = 0;
   }

  void synchronized modifie(int montant) {
    balance = balance + montant;
  }
}
```

La méthode est invoquable par un thread au plus (en exclusion mutuelle)

La complexité est laissée au compilateur

Synchroniser conjointement des méthodes Java

```
public class compteBanquaire {
 private int balance;
 compteBanquaire() { balance = 0; }

 void synchronized ajoute(int montant)
                          throws Exception {
  if (montant<0) {
    throw new Exception("Montant négatif");
  } else {
    balance = balance + montant;
  }
 }


 void synchronized retire(int montant)
                          throws Exception {
  if (montant<0) {
    throw new Exception("Montant négatif");
  } else if (balance - montant < 0) {
    throw new Exception("Solde insuffisant");
  } else {
    balance = balance - montant;
 }
}
```

Invocations sérialisées au sein de l'objet

- Entre threads
- Entre les méthodes du même objet
- Pas entre les instances
  (invocations parallèles sur $\neq$ objets)

Moniteurs et conditions

Moniteurs ne permettent pas d'attendre un événement

- Comparable au P() sur sémaphore à 0 (cf. place libérée sur le parking)
- Nouveau type de variable : `condition x;` (sorte de file d'attente)
- Primitives associées :
  - `x.wait()` : Entre dans la file d'attente associée
  - `x.notify()` : Libère un processus en attente (ou noop si personne n'attend)
- En Java, chaque objet a une condition implicite associée

Exemple : lecteur/écrivain

```
class Channel {
  private int contenu;
  private boolean plein;
  Channel() { plein = false; }
  synchronized void enqueue(int val) {
    while (plein) {
      try {
        wait();
      } catch (InterruptedException e){}
    }
    contenu = val; plein = true;
    notifyAll();
  }
  synchronized int dequeue() {
    int recu;

    while (!plein) {
      try {
        wait();
      } catch (InterruptedException e){}
    }
    recu = contenu;
    full = false;
    notifyAll();
    return recu;
  }
}
```

Synchronisations POSIX

Les sémaphores

- Font partie de System V, mais également de POSIX

Les verrous : mutex (mutual exclusion)

- Comme un sémaphore de capacité 1 (un processus prend le verrou)
- Sémantique plus simple / pauvre
- Standard = interface ; multiples implémentations (Linux : futex ; BSD : spinlock)
- Mutex réentrants : impossible de faire un deadlock avec soi-même

Les variables de condition

- Pour envoyer un événement aux gens qui l'attendent (ça les débloque)
- `signal()` débloque un seul processus ; `broadcast()` débloque tout le monde.
- Par rapport aux sémaphores : message perdu si personne n'est encore bloqué
- Usage assez complexe : il faut protéger chaque variable par un mutex

On y revient dans le prochain chapitre

compare-and-swap

Autre problème des sémaphores, moniteurs and co

- C'est bloquant : seul un algorithme peut agir en même temps
- Si on cherche non-bloquant, il faut des opérations atomiques

Compare-And-Swap (instructions binaires CAS et CAS2)

- Opération atomique : tentative de modification d'une variable partagée
- Si la valeur partagée est ce que je pense, je change la variable
  Si non, je suis informé de la nouvelle valeur partagée

```
int CAS(int*adresse, int *ancienne_val, int nouvelle_val) {
  if (*adresse == *ancienne_val) {
    *adresse = nouvelle_val;
    return TRUE;
  } else {
    *ancienne_val = *adresse;
    return FALSE;
  }   }
```

- Brique de base pour implémenter les sémaphores dans l'OS
- Recherche (actuelle) : structures partagées « Lock-free » et « Wait-free »

John D. Valois. *Lock-Free Linked Lists Using Compare-and-Swap.* 1995. (lire l'article)

RCU (Read-Copy-Update) dans Linux

Résumé du cinquième chapitre

Problèmes à éviter lors des synchronisations (et définitions)

- Interblocage : groupe de processus en attente mutuelle
- Compétition : le résultat dépend de l'ordre d'exécution
- Famine : un processus n'obtient pas la ressource car les autres l'en empêchent

Les sémaphores

- Principe : distributeur de jetons
- Opérations :
  - initialisation : mettre des jetons dans la boite
  - P : prendre un jeton (ou bloquer s'il y en a plus)
  - V : rendre un jeton pris

Schémas de synchronisation classiques

- Exclusion mutuelle : Ressource utilisée par au plus un utilisateur
- Cohorte : Ressource partagée entre au plus N utilisateurs
- Rendez-vous : un processus attend un autre, ou attente mutuelle
- Producteurs/consommateurs : utilisation **après** création
- Lecteurs/rédacteurs : concurrence entre **catégories** d'utilisateurs

Résumé du cinquième chapitre (suite)

Moniteurs

- Principe : Objet auto-protégé par sémaphore
- Avantage : Pas d'erreur de manipulation possible
- Usage en Java : synchronized

Variable de condition

- Principe : Comme un sémaphore privé pour la communication entre threads
- Avantage : Permet le passage de relai entre threads
- Usage en Java : `wait()/notifyAll()`

CAS (Compare-And-Swap)

- Principe : Tentative de modif. si personne n'a modifié depuis dernière lecture
- Avantages : Implementer les sémaphores ; algo lock-free

Conclusion générale

Il n'y a pas de magie noire dans l'ordinateur

- Même un OS est finalement assez simple, avec des idées très générales
- Concepts : processus, mémoire, système de fichiers, pseudo-parallélisme
- Idées : interposition, préemption, tout-est-fichier, bibliothèque de fonctions
- Synchro : compétition, interblocage, famine. Sémaphore, mutex, moniteur ...

Indispensable pour comprendre l'ordinateur

- 1% d'entre vous vont coder dans l'OS, 5% vont coder si bas niveau
- Mais ces connaissances restent indispensables pour comprendre la machine
- Programmer efficace en Java demande de comprendre le tas et les threads
- Les problèmes de synchro restent assez similaires dans les couches hautes

Ce que nous n'avons pas vu

- Le chapitre sur les threads
- L'implémentation de l'OS (cf. RSA) et des machines virtuelles
- High Perf Computing : Message-passing, cache-aware, out-of-core, parallélisme
- Recherche en OS : Virtualisation, synchro wait-free, spec formelle