Partie 2 :
Concurrence et Synchronisation

Chapitre 2 :
Synchronisation de processus

Table des Matières

1 Parallèlisme et concurrence

1.1 Parallèlisme (vrai ou simulé)

Plusieurs processus **séquentiels**
en parallèle

- Compétition d ’exécution

=> gestion de l'exécution (**planification**)

- Compétition sur les ressources

=> gestion des ressources :

  - processus **concurrents** : compétition d 'accès aux ressources (cpu, mémoire, fichiers…)
  - processus **coopératifs (threads)** : partage de ressources

1 Parallèlisme et concurrence

1.2 Ressources critiques

- Types de ressources :
  - consommables/réutilisables
    - consommables : détruite après utilisation
    - réutilisables : demeure après utilisation
  - locale/commune
    - locale : éphémère et interne à un processus
    - commune : demeure après utilisation
  - partageables ou non
    - partageable : plusieurs processus à la fois
    - non partageable : un seul processus à la fois

- Problème de **compétition** :
  ressource commune et non partageable
  => **ressource critique**

1 Parallèlisme et concurrence

1.3 Exemple de compétition

- Ressources critiques (RC)
- Accès concurrents (race conditions)

```
var RC: shared integer;
```

	Processus P1	Processus P2	
	var A : integer;	var B : integer;	
	begin	begin	
P11	A:=RC;	B:=RC;	P21
P12	RC:=A+1;	RC:=B+2;	P22
	end.	end.	

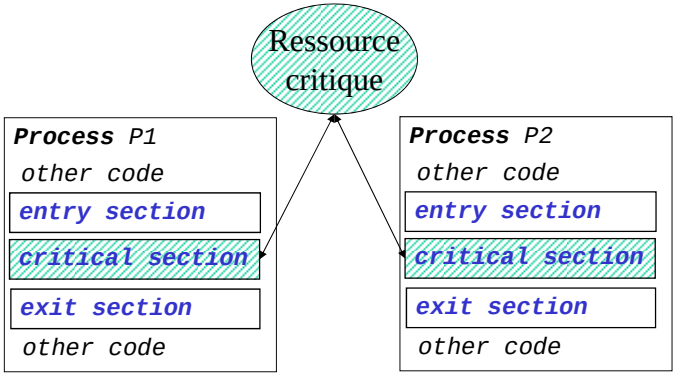
- P11 P21 P12 P22 ↯ RC+2 (erreur)
- P21 P11 P22 P12 ↯ RC+1 (erreur)
- ...
- P11 P12 P21 P22 ↯ RC+3 (bon)

2 Exclusion Mutuelle

2.1 Section critique

- Zone du code où une ressource critique est utilisée

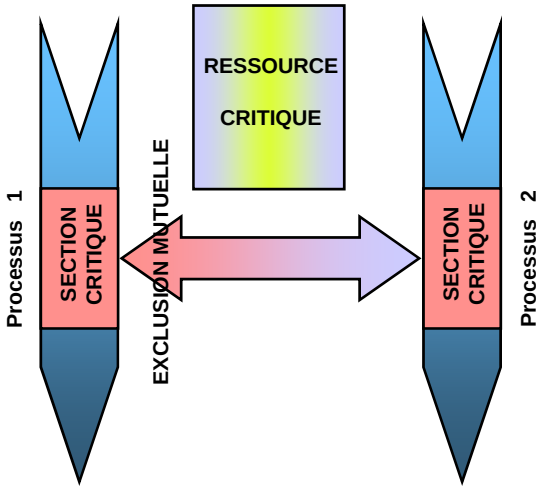
  => zone à protéger



2 Exclusion Mutuelle

2.2 Exclusion Mutuelle

- Protection des sections critiques
- Synchronisation par **Exclusion Mutuelle** :
  - un seul processus à la fois dans la section critique liée à une ressource critique
  - les autres attendent



2 Exclusion Mutuelle

2.3 Critères de validité

- E. W. Dijkstra
  - Un seul processus dans sa section critique
    ➧ *exclusion mutuelle*
  - Aucune hypothèses sur les vitesses relatives d'exécution ➧ *blocages*
  - Indépendance si arrêt hors de la section critique ➧ *deadlock, progression*
  - Admission en section critique en un temps fini ➧ *famine, attente bornée*

- Autres critères
  - Symétrie : rôle identique de chaque processus
  - justice : attente bornée, équité
  - simplicité
  - robustesse
  - distribuable

2 Exclusion Mutuelle

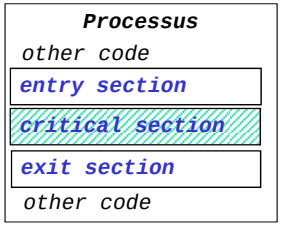
2.4 Forme générale des solutions

DÉCLARATIONS COMMUNES :

- De toutes les variables et structures communes invoquées par tous les processus

Pour CHAQUE PROCESSUS :

- Une séquence d ’INITIALISATION
  - Initialisation des entités invoquées dans la suite
- Une séquence d ’ENTRÉE en section critique (PROLOGUE)
  - Assure la réalisation des condition garantissant l ’accès en section critique dans les conditions souhaitées
- Une séquence de SORTIE de la section critique (ÉPILOGUE)
  - Permet d ’informer les autres processus que la section critique devient libre



2 Exclusion Mutuelle

2.5 Implémentation des solutions

- Matériel
  - instructions spécifiques du processeur
- OS
  - appels systèmes (sémaphores, conditions)
- Langage de programmation
  - fonctionnalités multiprocessus du langage
    - rendezvous en Ada
    - moniteurs en Java
- Application
  - prise en charge par l ’utilisateur

- Toutes ces solutions permettent la gestion des sections critiques par E.M.

2 Exclusion Mutuelle

2.6 Types d ’attente

Attente de l’autorisation d’accès à la section critique

- Attente active
  - solution logicielle
  - boucle d ’attente
  - consommation de cpu
- Attente passive
  - solution matérielle/système
  - changement de l’état du processus
    - actif -> bloqué
    - bloqué -> activable
  - pas de consommation cpu

2 Exclusion Mutuelle

2.7 Types d ’implémentation des solutions

- gestion locale ou distribuée
  - chaque processus gère ses conflits avec les autres processus concurrents
  - plus complexe car code de synchronisation distribué dans chacun des processus en compétition
  - ex: sémaphores
- gestion centralisée
  - un serveur de synchronisation centralise la gestion des conflits
  - plus simple car code centralisé
  - ex: moniteurs

3 Solutions avec attente active

- 3.1 Verrou logiciel
- 3.2 Verrou matériel
- 3.3 Indicateurs d'occupation
- 3.4 L'alternance (tour)
- 3.5 Algorithme de Dekker
- 3.6 Algorithme de Peterson
- 3.7 Algorithme de Dijkstra

3 Solutions avec attente active

- 3.1 Verrou ou marqueur logiciel

- PRINCIPE :
  - Créer une variable booléenne commune contrôlant l ’entrée en section critique
    - Vrai : interdiction entrée en section critique
    - Faux : autorisation d ’entrée
- INITIALISATION :
  - Marqueur à faux
- PROLOGUE :
  - Tester valeur du marqueur
  - Si vrai, recommencer test
  - Si faux, mettre marqueur à vrai
  - Exécuter section critique
- ÉPILOGUE :
  - Marqueur à faux

3 Solutions avec attente active

- 3.1 Verrou ou marqueur logiciel
  Variable de verrouillage

```
Boolean : lock := false;
```

`While lock do ;` `lock := true;`	`While lock do ;` `lock := true;`
section critique P1	section critique P2
`lock := false;`	`lock := false;`

  - 1 variable booléenne partagée
  - multi-processus
  - Inconvénients
    - pas d ’exclusion mutuelle !
    - Le verrou est une ressource critique !
    - attente active

3 Solutions avec attente active

- 3.2 Verrou ou marqueur matériel
  Variable partagée + Primitive de verrouillage

```
Boolean : lock := false;
```

`While tas(lock) do ;`	`While tas(lock) do ;`
section critique P1	section critique P2
`lock := false;`	`lock := false;`

  - 1 variable partagée
  - Primitive matérielle (instruction *TAS, TST, TSL*)
  - Primitive atomique (indivisible)
  - Avantages
    - Exclusion Mutuelle multi-processus
    - simple, facile à vérifier
    - sections critiques multiples
  - Inconvénients
    - variable partagée
    - attente active
    - *famine possible, interblocages possibles*

3 Solutions avec attente active

- 3.3 Indicateurs d'occupation (drapeau)

- PRINCIPE :
  - Créer une variable booléenne commune pour chacun des processus, indiquant s 'il est ou non en section critique
    - Vrai : en section critique
    - Faux : hors section critique
- INITIALISATION :
  - Indicateur à faux
- PROLOGUE :
  - Mettre à vrai son indicateur (1)
  - Tester autre indicateur
  - Si vrai, recommencer test
  - Si faux, exécuter section critique
- ÉPILOGUE :
  - Marquer à faux son indicateur

3 Solutions avec attente active

- 3.3 Indicateurs d'occupation (drapeau)

Variables partagées indiquant la présence en section critique

```
Boolean : SC1, SC2 := false;
```

`SC1 := true;`	`SC2 := true;`
`While SC2 do ;`	`While SC1 do ;`
section critique P1	section critique P2
`SC1 := false;`	`SC2 := false;`

  - 2 variables partagées, limité à 2 processus !
  - Avantages
    - Exclusion Mutuelle
  - Inconvénients
    - variables partagées
    - attente active
    - *interblocage possible*

3 Solutions avec attente active

- 3.4 Tour (alternance)

- PRINCIPE :
  - Créer une variable entière donnant le numéro du processus autorisé à entrer en section critique (tour)
  - Chaque processus termine en autorisant un autre à entrer.

- INITIALISATION :
  - Entier à 1 ou 2
- PROLOGUE :
  - Comparaison du numéro processus à celui autorisé
  - Si différent, recommencer
  - Si égalité, entrer en section critique
- ÉPILOGUE :
  - Mettre numéro autorisé à celui de l ’autre processus

3 Solutions avec attente active

- 3.4 Tour (alternance)

  Variable partagée indiquant le numéro du processus à entrer en section critique

```
Integer : Tour := 1;
```

```
While Tour ≠ 1 do ;
section critique P1
Tour := 2;
```

```
While Tour ≠ 2 do ;
section critique P2
Tour := 1;
```

  - 1 variable partagée, limité à 2 processus !
  - Avantages
    - Exclusion Mutuelle
  - Inconvénients
    - variable partagée
    - attente active
    - *ordre d ’alternance imposé*
    - *Attente infinie si l ’un des processus ne repasse pas par sa section critique*

3 Solutions avec attente active

- 3.5 Algorithme de Dekker

- PRINCIPE :
  - Combinaison des algorithmes Tour+Indicateurs
- INITIALISATION :
  - Indicateurs à faux, TOUR à 1 ou 2
- PROLOGUE :
  - Mise à vrai de l ’indicateur soi-même
  - Examen de l ’indicateur de l ’autre
  - Si vrai consultation du tour autorisé
  - Si tour à l ’autre, remise indicateur propre à faux et attente
  - Si c ’est le bon, attente libération indicateur de l ’autre, et entrée section critique
- ÉPILOGUE :
  - Mettre numéro tour à celui de l ’autre
  - Mettre indicateur propre à faux

3 Solutions avec attente active

- 3.5 Algorithme de Dekker

```
Integer : Tour := 1;
Boolean : SC1, SC2 := false;
```

```
SC1 := true;
while SC2 do
   if tour=2 then
      SC1 := false;
      while tour=2 do;
      SC1 := true;
```

```
SC2 := true;
while SC1 do
   if tour=1 then
      SC2 := false;
      while tour=1 do;
      SC2 := true;
```

section critique P1

```
Tour := 2;
SC1 := false;
```

section critique P2

```
Tour := 1;
SC2 := false;
```

  - 3 variables partagées, limité à 2 processus !
  - Avantages
    - Exclusion Mutuelle
  - Inconvénients
    - variable partagée
    - attente active

3 Solutions avec attente active

- 3.6 Algorithme de Peterson (1981)

- PRINCIPE :
  - Amélioration de l 'algorithmes de Dekker. Même profils de variables
- INITIALISATION :
  - Indicateurs à faux, TOUR à 1 ou 2
- PROLOGUE :
  - Mise à vrai de l 'indicateur soi-même
  - Donne tour à l 'autre
  - Examine indicateur pour l 'autre
  - Si vrai et tour à l 'autre, attente
  - Sinon, entrée section critique
- ÉPILOGUE :
  - Mettre indicateur propre à faux

3 Solutions avec attente active

- 3.6 Algorithme de Peterson (1981)

```
Integer : Tour := 1;
Boolean : SC1, SC2 := false;
```

`SC1 := true;`
`tour := 2;`
`while SC2 and tour=2 do;`
	`SC2 := true;`
	`tour := 1;`
	`while SC1 and tour=1 do;`

section critique P1	section critique P2
`SC1 := false;`	`SC2 := false;`

  - 3 variables partagées, limité à 2 processus !
  - Avantages
    - Exclusion Mutuelle
  - Inconvénients
    - variable partagée
    - attente active

3 Solutions avec attente active

- 3.7 Algorithme de Dijkstra
- PRINCIPE :
  - Algorithme applicable à N processus
  - Utilisation du numéro de tour
  - Gestion de deux indicateurs, l 'un pour la requête, l 'autre pour la mise en attente
- FONCTIONNEMENT :
  - Sans conflit d 'accès, place sa requête, son tour, entre en attente, puis en SC lorsqu'il reste seul en attente
  - Si conflit, place sa requête, examine le n° de tour pour savoir quel processus actif, attend la fin du processus, se place en attente. Entre en SC lorsque aucun autre processus en attente

3 Solutions avec attente active

- 3.7 Algorithme de Dijkstra

```
Array (0..N) of Boolean : REQ  (0…N := faux);
Array (0..N) of Boolean : ATT  (0…N := faux);
Integer : TOUR := 0
```

```
REQ[i] := true;
Boolean continuer := true;
while continuer
   while TOUR ≠ i do
      ATT[i] := false;
      if not REQ[TOUR] then
         TOUR := i;
   ATT[i] := true;
   continuer := false;
   for Integer : j :=1 to N do
      if j ≠ i and ATT[j] then
         continuer := true;
         break;
```

```
section critique Pi
```

```
TOUR := 0
REQ[i] := false;
ATT[i] := false;
```

- 2N+3 variables partagées, N processus
- complexe !

3 Solutions avec attente active

- 3.8 Algorithme de Lamport

- PRINCIPE :
  - Algorithme applicable à N processus
  - Utilisation d ’un numéro d ’ordre et d ’un indicateur
  - Gestion de deux indicateurs, l ’un pour le numéro d ’ordre, l ’autre pour les indicateurs de mise en attente

- FONCTIONNEMENT :
  - Sans conflit d ’accès, élection du processus demandeur après balayage de la file d ’attente
  - Si conflit, obtient un numéro d ’ordre, et le place en file d ’attente. Ne se donnera l ’autorisation de pénétrer en SC qu ’après examen qu ’il n ’existe pas de numéro inférieur au sien (sauf 0)

- ***CRITIQUE :***
  - ***Algorithme d ’ordonnancement de type FIFO***
  - ***Croissance illimitée des numéros si toujours au moins un processus en attente***
  - ***Attente active***

3 Solutions avec attente active

- 3.8 Algorithme de Lamport (1974)

```
Array (1..N) of Boolean : REQ  (0…N := false);
Array (1..N) of Integer : ATT  (0…N := 0);
```

```
REQ[i] := true;
ATT[i] := 1+ max(ATT[1..N]);
REQ[i] := false;
for Integer : j:=1 to N do
   while REQ[j] do;
   while 0 < ATT[j] < ATT[i] do;
```

```
section critique Pi
```

```
ATT[i] := 0;
```

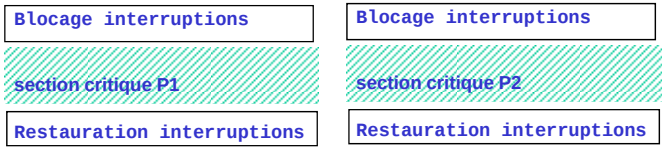
  - 2N variables partagées, N processus

4 Solutions avec attente passive

- 4.1 Masquage des interruptions
- 4.2 Primitives sleep/wakeup
- 4.3 Sémaphores
- 4.4 Moniteurs

4 Solutions avec attente passive

- 4.1 Masquage des interruptions



- Avantages
  - Exclusion Mutuelle
  - attente passive
- Inconvénients
  - très dangereux (restauration non effectuée, Risque de blocage définitif)
  - impossible en multiprocessing
  - généralement réservé au mode noyau

4 Solutions avec attente passive

- 4.2 Primitives sleep/wakeup

Modification directe de l'état du processus
Appels système

```
Sleep(P2);
section critique P1
wakeup(P2);
```

```
Sleep(P1);
section critique P2
wakeup(P1);
```

- Avantages
  - Exclusion Mutuelle
  - attente active
- Inconvénients
  - connaître tous les processus en compétition

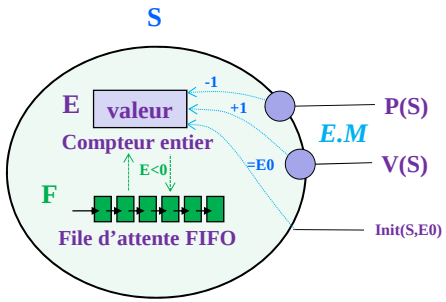
- NB : signaux SIGINT, SIGCONT en unix

```
kill –SIGINT pid
kill –SIGINT pid

kill(
```


4 Solutions avec attente passive

4.3 Les **Sémaphores** *(DIJSKTRA 1965)*



4 Solutions avec attente passive

- 4.3 Sémaphores *(DIJSKTRA 1965)*

OBJECTIFS :

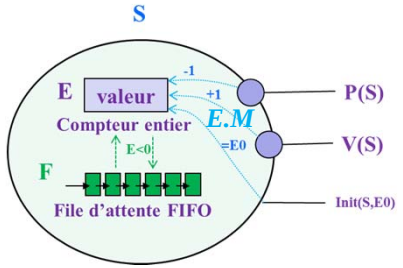
- Contrôle d'accès à des ressources critiques
- Suppression de l'attente active
- Ordonnancement des requêtes d'accès

PRINCIPE :

- Blocage du processus dont la requête ne peut être immédiatement satisfaite et mise en file d 'attente
- Activation d'un processus dès libération de la ressource
- Gestion de la file d'attente

4 Solutions avec attente passive

- 4.3 Sémaphores - Définition



Données S := (E, F) :

- **E** : 1 variable entière (valeur sémaphore, compteur)
- **F**: 1 file d'attente FIFO de processus

Primitives P(S),V(S) :

S.E uniquement modifiable par des primitives indivisibles (atomiques),
Attente passive dans S.F

- **P (S)** - demande d'accès à ressource protégée
  - décrémente le compteur S.E : S.E := S.E - 1
  - si S.E < 0, mise en sommeil en file d'attente S.F du processus
- **V (S)** - libération de la ressource protégée
  - incrémente le compteur S.E : S.E := S.E + 1
  - si S.E ≤ 0, réveil du 1er processus en file d'attente S.F (le plus ancien)
- INIT(S, valeur) - initialisation - (valeur ≥ 0)
  - S.E := valeur, S.F = ∅ (file d'attente vide)

NB:

- P et V du néerlandais *Proberen* et *Verhogen*, pour *tester* et *incrémenter*
- synonymes P(S) ⇔ S.wait, et V(S) ⇔ S.signal

4 Solutions avec attente passive

- 4.3 Sémaphores - Signification

Valeur S.E initiale

- Nombre de processus appeler P(S) (ou S.wait) successivement sans blocage (mise en sommeil dans F)

= Quantité disponible d'une ressource

Valeur S.E courante

- Si négative : |S.E| = nombre de processus en file d 'attente
- Si positive ou nulle : S.E = nombre de processus pouvant encore appeler P(S) successivement sans blocage

Remarques sur les primitives

- Autant de V(S) que de P(S) au final
- V(S) jamais bloquant (alors que P(S) peut l'être)

4 Solutions avec attente passive

- 4.3 Sémaphores – Classification

Sémaphores Entiers :

- La variable est de type Entier (entier relatif)
- Sémaphores de Contingentement ou de Comptage :
  - Limite la consommation d 'une ressource
    - Valeur initiale = N -> Quantité de ressources disponibles
- Sémaphores Point de rencontre :
  - Pour la synchronisation sur événements multiples
    - Valeur initiale = - n (|n| = nombre de processus à synchroniser)

Sémaphores Binaires :

- La variable est de type Binaire (1,0)
- Sémaphores d'Exclusion Mutuelle :
  - Protège l 'accès à une section critique
    - Valeur initiale = 1 -> disponible (available)
    - Valeur = 0 -> occupée (owned)
- Sémaphores de Signalisation :
  - Pour la synchronisation inter-processus sur événements (attente, rendez-vous)
    - Valeur initiale = 0 -> établi, armé (set)
    - valeur = 1 -> libre (clear)

Variantes : Sémaphores avec estampilles, avec messages.

4 Solutions avec attente passive

- 4.3 Sémaphore de comptage - primitives

Variables communes du sémaphore S
entier : E := E_0 (compteur entier)
tableau (1 .. N) : F := 0 (file d'attente de processus)

primitive P(S)
-- exécutée par Pr
entrée section critique de S
E := E - 1 ;
si E < 0 alors
empiler Pr dans F
mise en attente de Pr
et sortie de section critique de S
sinon
sortie section critique de S
finsi

primitive V(S)
--
entrée section critique de S
E := E + 1 ;
si E <= 0 alors
Pr := dépiler processus de F
sortie section critique de S
réveiller processus Pr
sinon
sortie section critique de S
finsi

4 Solutions avec attente passive

- 4.3 Sémaphores - Expression abstraite

sémaphore : S
S.E := E_0;

Processus : P (i)
…
P (S)
… section protégée
V (S)
…

i processus ….

- Signaux du protocole :
#EXC(P) = nombre d'exécution de la primitive P(s)
#EXC(V) = nombre d'exécution de la primitive V(s)
#AUT(SP) = nombre d'entrées en section protégée
#ATT(SP) = nombre de processus en file d'attente F

- Donnée du protocole :
E = Valeur actuelle de la variable sémaphore E
E_0 = Valeur initiale de cette variable

4 Solutions avec attente passive

- 4.3 Sémaphores - Expression abstraite

#EXC(P) = nombre d'exécution de la primitive P(s)
#EXC(V) = nombre d'exécution de la primitive V(s)
#AUT(SP) = nombre d'entrées en section protégée
#ATT(SP) = nombre de processus en file d'attente F
E = Valeur actuelle de la variable sémaphore E
E_0 = Valeur initiale de cette variable

- Relations de base :

$E = E_0 - \#EXC(P) + \#EXC(V)$ (1)
$\#AUT(SP) <= \#EXC(P)$ (2)

- Conditions d 'exécution de la section protégée sans mise en file d 'attente :

$\#ATT(SP) = 0$
d 'où $\#AUT(SP) = \#EXC(SP)$ et $\#AUT(SP) < \#EXC(V) + E_0$

- Conditions de sortie d 'un processus de la file d 'attente :

$\#ATT(SP) > 0$
d 'où $\#EXC(P) > \#AUT(SP)$ et $\#AUT(SP) = \#EXC(V) + E_0$

- D 'où invariant du protocole :

$\#AUT(SP) = \min (\#EXC(P) , \#EXC(V) + E_0)$

4 Solutions avec attente passive

- 4.3 Sémaphores - Expression abstraite - Preuve

#EXC (P) = nombre d 'exécution de la primitive P(s)
#EXC (V) = nombre d 'exécution de la primitive V(s)
#AUT (SP) = nombre d 'entrées en section protégée
#ATT (SP) = nombre de processus en file d'attente F
E = Valeur actuelle de la variable sémaphore E
E_0 = Valeur initiale de cette variable

- Rappel :

$E = E_0 - \#EXC(P) + \#EXC(V)$ (1)
$\#AUT(SP) <= \#EXC(P)$ (2)

invariant : $\#AUT(SP) = \min (\#EXC(P) , \#EXC(V) + E_0)$ (3)

- Pour E < 0,

la relation (1) donne : $E_0 + \#.EXC (V) < \#EXC(P)$
qui porté dans (3) entraîne : $\#AUT(SP) = \#EXC (V) + E_0$
où, en remplaçant par sa valeur tirée de (1) :
$- E = \#EXC(P) - \#AUT(SP)$

Donc | E| = longueur de la file d 'attente

- Pour E > 0 ,

la relation (1) donne : $\#EXC(P) < \#EXC(V) + E_0$
soit $\#EXC(P) - \#EXC(V) < E_0$

Donc E_0 = nombre maximum de processus traversant la SP sans blocage

4 Solutions avec attente passive

- 4.3 Sémaphores - Utilisation

- Exclusion Mutuelle
- Partage de ressources
- Relation de précédence (ordonnancement)
- Rendez-vous, point de synchronisation

4 Solutions avec attente passive

- 4.3 Sémaphores - Exclusion Mutuelle

Variables communes/partagées

Semaphore : mutex
mutex.E := 1 (ou mutex(1) ou init(Mutex, 1))

Processus : P1
tant que faux faire
...
P (mutex)
section critique
V (mutex)
...
fin tant que

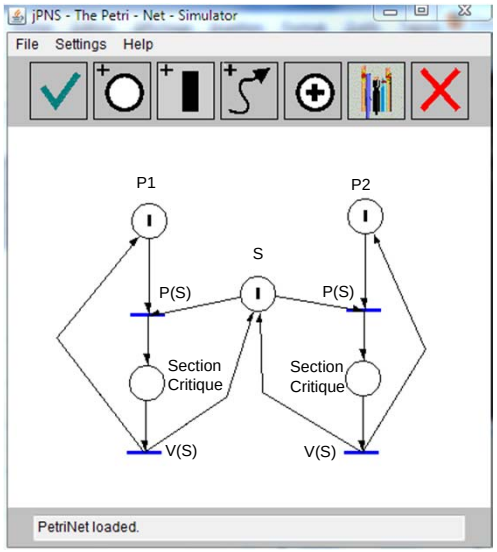
n processus ...

Processus : Pn
tant que faux faire
...
P (mutex)
section critique
V (mutex)
...
fin tant que

RAPPEL PROPRIÉTÉS :
- Un seul processus à la fois en section critique
- Pas d 'interblocage hors incident en SC
- Pas d 'incidence si blocage hors SC d 'un quelconque processus

4 Solutions avec attente passive

- 4.3 Sémaphores - Exclusion Mutuelle



4 Solutions avec attente passive

- 4.3 Sémaphores - Exemples

- Alternance:

A compléter

```
Processus : P1
  .....
  Repeter
    ....
    Ping
    ....
  Jusqua Faux
  .....
```

```
Processus : P2
  .....
  Repeter
    ....
    Pong
    ....
  Jusqua Faux
  .....
```

- Etreinte fatale:

A compléter

```
Processus : P1
  .....
  Repeter
    ....
    Prendre Crayon
    Prendre Feuille
    Ecrire
    ....
  Jusqua Faux
  .....
```

```
Processus : P2
  .....
  Repeter
    ....
    Prendre Feuille
    Prendre Crayon
    Ecrire ....
  Jusqua Faux
  .....
```

4 Solutions avec attente passive

4.4 Sémaphores - Problème assemblage Avion

A compléter

Semaphore:

```
Processus : Carlingue
   repeter
      Une carlingue est achevée
```

```
Processus : Carlingue1Ailes2
   repeter
      Un assemblage 1 carlingue avec 2 ailes est acheve
```

```
Processus : Aile
   repeter
      Une aile est achevée
```

```
Processus : Moteur
   repeter
      Une aile est achevée
```

```
Processus : Carlingue1Ailes2Roues3
   Repeter
      Un assemblage 1 carlingue et 2 ailes avec 3 roue est acheve
```

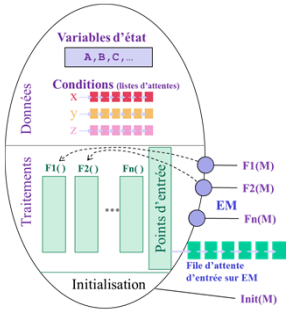
```
Processus : Roue
   repeter
      Une aile est achevée
```

```
Processus  : Aeroplane
   Repeter
      Un aéroplaneest acheve
```

```
Processus : Principal
   ParDebut
      Carlingue1Ailes2
      Carlingue1Ailes2Roues3
      Aeroplane
      Carlingue
      Aile
      Moteur
      Roue
   ParFin
```

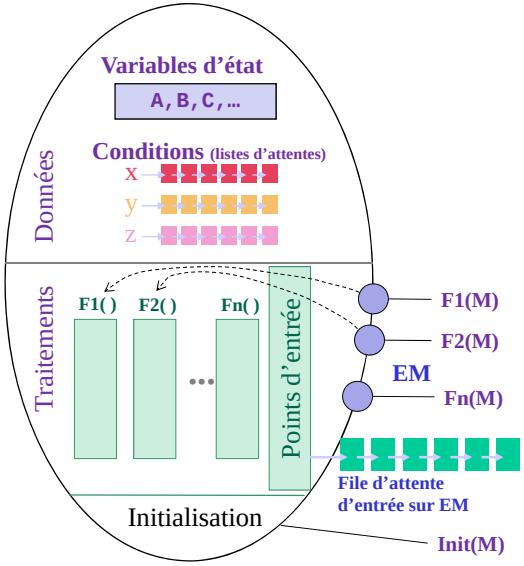

4 Solutions avec attente passive

4.4 Les **Moniteurs** (*HOARE & HANSEN 1974*)



4 Solutions avec attente passive

- 4.4 Les moniteurs - ***HOARE & HANSEN (1974)***



4 Solutions avec attente passive

4.4 Moniteur : Méthode de programmation

1. Points d'entrée (fonctions critiques)
   Nom des procédures critiques
2. Variables d'état (ressources critiques)
   Définir les variables d'état à protéger et les variables supplémentaires à ajouter pour résoudre la synchronisation
3. Initialisation
   Fixer les valeurs initiales des variables
4. Conditions (files d'attente)
   Définir les listes d'attentes de processus
5. Assertions
   Définir les conditions logiques de déclenchement des opérations .wait() et .signal() sur chaque condition en fonction des variables d'état
6. Coder

4 Solutions avec attente passive

4.4 Moniteur : Exemple

Implémenter un sémaphore de comptage avec un moniteur

A compléter

4 Solutions avec attente passive

4.4 Moniteur - Problème assemblage Avion

A reprendre et à compléter

```
Moniteur :

Processus : Carlingue
   repeter
      Une carlingue est achevée

Processus : Carlingue1Ailes2
   repeter
      Un assemblage 1 carlingue avec 2 ailes est acheve

Processus : Aile
   repeter
      Une aile est achevée

Processus : Carlingue1Ailes2Roues3
   Repeter
      Un assemblage 1 carlingue et 2 ailes avec 3 roue est acheve

Processus : Moteur
   repeter
      Une aile est achevée

Processus : Roue
   repeter
      Une aile est achevée

Processus  : Avion
   Repeter
      Un avion est acheve

Processus : Principal
   ParDebut
      Carlingue1Ailes2
      Carlingue1Ailes2Roues3
      Avion
      Carlingue
      Aile
      Moteur
      Roue
   ParFin
```

5 Producteurs - Consommateurs

5 Producteurs - Consommateurs

Problèmes des Producteurs - Consommateurs

- Coopération entre plusieurs processus
- Tampon (ressource critique)
- Deux opérations
  - Retirer (processus consommateurs)
    - Modification
    - Exclusion mutuelle
  - Déposer (processus producteurs)
    - Modification
    - Exclusion mutuelle

+ Exclusion mutuelle globale

- Règles
  - Tampon vide
    => Les consommateurs attendent
  - Tampon plein
    ⇒ Les producteurs attendent

- Exemples: tubes unix, programmation événementielle, IHM

5 Producteurs - Consommateurs

Tampon (Entrepôt)

- Ressource Critique
- Borné ou non
- Nombre de places occupées
- Variable commune, tableau, bloc de mémoire partagée, fichier, Tampon circulaire, Tampon multiple, Boite aux lettres (BAL)…

- Section critique, Exclusion Mutuelle
- Contraintes :
  - Objet consommable (une fois)
  - Ordre de consommation FIFO

5 Producteurs - Consommateurs

Tampon/Entrepôt:

```
Type Tampon  …
InitTampon (Tampon: t , Entier: tailleMax)
NbProduits (Tampon: t) → Entier
TailleMax (Tampon: t) → Entier
EstVide (Tampon: t) → Booleen
EstPlein (Tampon: t) → Booleen
Deposer (Tampon: t, Produit: p)
Retirer (Tampon: t) → Produit
```

```
Var
    Tampon t;
    …
```

```
Processus : Producteur
  .....
  Repeter
    ....
    Deposer (t, p)
    ....
  Jusqua Faux
  .....
```

```
Processus : Consommateur
  .....
  Repeter
    ....
    p := Retirer (t)
    ....
  Jusqua Faux
  .....
```

```
Processus : Principal
  InitTampon (t, TailleMax);
  ParDebut
    Producteur;
    Consommateur;
  ParFin
```

5 Producteurs – Consommateurs

Solution avec Sémaphores de Comptage

Tampon non borné :

- Blocage consommateurs (tampon vide)
- 2 sémaphores :
  - Mutex : exclusion mutuelle sur le tampon
  - Places-occupées : ressources disponibles dans l'entrepôt, blocage si 0

Tampon borné :

- TailleMax ressources
- Blocage consommateurs (tampon vide)
- Blocage producteurs (tampon plein)
- 3 sémaphores :
  - Mutex : exclusion mutuelle sur le tampon
  - Places-occupées : nb ressources déposées dans l'entrepôt, blocage consommateurs si 0 (vide)
  - Places-libres : nb espaces disponibles dans l'entrepôt, blocage producteur si 0 (plein)

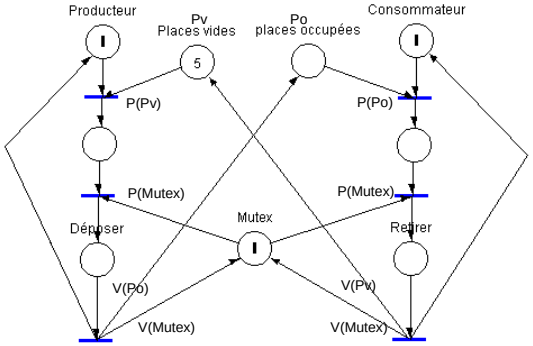
5 Producteurs – Consommateurs

Solution avec Sémaphores de Comptage
1: Exclusion Mutuelle
2: gestion Tampon vide
3: gestion Tampon plein

A compléter

5 Producteurs – Consommateurs
Solution Sémaphores de Comptage

Réseau de Pétri



Simulation avec jpns …

5 Producteurs – Consommateurs
Solution Moniteur

```
Type Tampon  …
InitTampon (Tampon: t , Entier: tailleMax)
NbProduits (Tampon: t) → Entier
TailleMax (Tampon: t) → Entier
EstVide (Tampon: t) → Booleen
EstPlein (Tampon: t) → Booleen
Deposer (Tampon: t, Produit: p)
Retirer (Tampon: t) → Produit
```

Moniteur moniteurPC pour protéger le tampon

- Variable d'état:
  - le tampon (ressource critique)
- Points d'entrée:
  - moniteurRetirer ()
    - pour processus consommateurs
    - Appel Deposer(tampon)
    - Exclusion mutuelle
  - moniteurDéposer ()
    - pour processus producteurs
    - Appel Deposer(tampon)
    - Exclusion mutuelle
- Conditions
  - Tampon Plein : attenteProd
  - Tampon Vide : attenteCons

+ Exclusion mutuelle globale

5 Producteurs – Consommateurs
Solution Moniteur

Solution avec Moniteur (E.M.)

1: gestion Tampon vide

2: gestion Tampon plein

A compléter…

```
Type Tampon  …
InitTampon (Tampon: t , Entier: tailleMax)
NbProduits (Tampon: t) → Entier
TailleMax (Tampon: t) → Entier
EstVide (Tampon: t) → Booleen
EstPlein (Tampon: t) → Booleen
Deposer (Tampon: t, Produit: p)
Retirer (Tampon: t) → Produit
```

```
Moniteur : MoniteurPC

Var
   Tampon t;

   MoniteurDeposer (Produit: p)
   MoniteurRetirer () → Produit;

   Initialisation:
      InitTampon (t, TailleMax)
```

```
MoniteurDeposer (Produit: p)
   Deposer ( t,  p);

MoniteurRetirer ()
   Retirer( t,  p);
   → p
```

```
Processus : Producteur
   .....
   Repeter
      MoniteurDeposer (p)
   Jusqua Faux
   .....
```

```
Processus : Consommateur
   .....
   Repeter
      p := MoniteurRetirer ();
   Jusqua Faux
   .....
```

```
Processus : Principal
   ParDebut
      Producteur
      Consommateur
   ParFin
```


4 Solutions avec attente passive

4.4 Moniteur - Problème assemblage Avion

A reprendre et à compléter
avec une synchronisation producteurs consommateurs

```
Tampons de production: ....
```

```
Processus : Carlingue
   repeter
      Une carlingue est achevée
```

```
Processus : Carlingue1Ailes2
   repeter
      Un assemblage 1 carlingue avec 2 ailes est acheve
```

```
Processus : Aile
   repeter
      Une aile est achevée
```

```
Processus : Moteur
   repeter
      Une aile est achevée
```

```
Processus : Carlingue1Ailes2Roues3
   Repeter
      Un assemblage 1 carlingue et 2 ailes avec 3 roue est acheve
```

```
Processus : Roue
   repeter
      Une aile est achevée
```

```
Processus  : Avion
   Repeter
      Un avion est acheve
```

```
Processus : Principal
   ParDebut
      Carlingue1Ailes2
      Carlingue1Ailes2Roues3
      Avion
      Carlingue
      Aile
      Moteur
      Roue
   ParFin
```

6 Threads en Python

6 Threads en Python

Les Threads

Bibliothèque Python: threading

```
Import threading
```

Création thread:

```
myThread = threading.Thread(target=fonction, args=*nuplet, ...)
```

Démarrage exécution:

```
myThread.start()
```

Attente terminaison:

```
myThread.join()
```

Variable partagée:

```
global myVar
```

Exemple:

```
import threading

compteur_global = 0

# Fonction principale des threads
def thread_main(nom) :
        global compteur_global
        compteur_global += 1

# Fonction principale
t1 = threading.Thread( target=thread_main, args=("T1", ) )
t2 = threading.Thread( target=thread_main, args=("T2", ) )

t1.start()
t2.start()

t1.join()
t2.join()
```

6 Threads en Python
Sémaphores

Les Sémaphores de comptage

Bibliothèque Python: threading

```
Import threading
```

Création Sémaphore:

```
semaphore = threading.Semaphore ( valeurInitiale )
```

Opération P() ou wait():

```
semaphore.acquire()
```

Opération V() ou signal():

```
semaphore.release()
```


6 Threads Python
Sémaphores
Exemple d'exclusion mutuelle

```
import threading

compteur_global = 0


# Fonction principale des threads
def thread_main(nom) :
      global compteur_global
      compteur_global += 1


# Fonction principale
t1 = threading.Thread( target=thread_main, args=("T1", ) )
t2 = threading.Thread( target=thread_main, args=("T2", ) )

t1.start()
t2.start()

t1.join()
t2.join()
```

6 Threads en Python
Sémaphores
Exemple d'exclusion mutuelle

```
import threading

compteur_global = 0

mutex = threading.Semaphore(1)


# Fonction principale des threads
def thread_main(nom) :
        global compteur_global, mutex
        mutex.acquire()
        compteur_global += 1
        mutex.release()


# Fonction principale
t1 = threading.Thread( target=thread_main, args=("T1", ) )
t2 = threading.Thread( target=thread_main, args=("T2", ) )

t1.start()
t2.start()

t1.join()
t2.join()
```


6 Threads en Python
Sémaphores

Autre exemple de compétition à une ressource critique à résoudre par Exclusion Mutuelle

```
import threading

# boucle de répétition de chaque thread
NbCoups = 10

# ressource critique: le compteur
compteur =0

# Fonction principale des threads
def test (nom):
    global compteur
    print( 'Debut de ' + nom)
    for i in range(NbCoups):
        lc = compteur
        compteur = lc +1
# liste des noms de threads à créer
NomsThreads = ['T1','T2']
threads = []


# Création des Thread
for nom in NomsThreads :
    threads.append(threading.Thread(target=test, args=(nom,)))
# ou
# threads = [threading.Thread(target=test_sem, args=(nom,)) for
nom in NomsThreads]


print( 'Valeur initiale du compteur :', compteur)

# Démarrage des threads
for t in threads:
    t.start()

# Attente de terminaison des threads
for t in threads:
    t.join()

print( 'Valeur finale du compteur :', compteur)
```

6 Threads en Python
Moniteurs

Bibliothèque Python: threading

```
Import threading
```

Mutex Variables (Locks, Binary Semaphores):

```
myLock = threading.Lock()

myLock.acquire()

myLock.release()
```

NB: équivalent à Semaphore(1)

Condition Variables:

```
condition = threading.Condition(lock=myLock)

condition.wait()

condition.notify()
```

Autres bibliothèques:

```
time.sleep( value )

value = random.randint(minVal, maxVal)

multiprocessing.current_process()
multiprocessing.current_process().pid
os.getpid()

threading.current_thread()
threading.get_ident()
```


6 Threads en Python
Moniteurs
Exemple d'exclusion mutuelle

```
import threading

compteur_global = 0

mutex = threading.Lock()

# Fonction principale des threads
def thread_main(nom) :
    global compteur_global, mutex
    mutex.acquire()
    compteur_global += 1
    mutex.release()

# Fonction principale
t1 = threading.Thread( target=thread_main, args=("T1", ) )
t2 = threading.Thread( target=thread_main, args=("T2", ) )

t1.start()
t2.start()

t1.join()
t2.join()
```

6 Threads en Python
Sémaphores de comptage par moniteur

Implémenter un sémaphore de comptage

avec les moniteurs en Python (Lock et Condition)

A compléter

6 Threads en Python
Producteurs - Consommateurs

Gestion du Tampon

```
#
#                Problème des Producteurs Consommateurs
#                Definition du tampon et des fonctions d'accès
#

# Notions Globales

# Taille du tampon
TAILLEMAX = 100


#  gestion du tampon

Class Tampon() :
   def __init__ (self, tailleMax) :
      self.element = []
      self.nb_elements = 0
      self.taille_max= tailleMax

def tampon_deposer ( tampon, element ) :
def tampon_retirer ( tampon ) :
def tampon_est_vide ( tampon ) :
def tampon_est_plein (  tampon ) :
def tampon_nbElements (  tampon ) :
def tampon_tailleMax(  tampon ) :
```

6 Threads en Python
Producteurs – Consommateurs

Programme principal

```
import threading

# Ressource Critique
tampon = Tampon(10)

# Fonction principale des threads "producteurs"
def producteur(tampon)  :
   …
   tampon_deposer(tampon, '1')
   …

# Fonction principales des threads "consommateurs"
def consommateur(tampon)  :
   …
   elt=tampon_retirer(tampon)
   …

# Fonction principale de démarrage et de création des threads
nomsThreads  =  ["c1","p1", "p2", "p3", "p4", "c2", "c3", "c4 " … ]
Threads = []

for t in nomsThreads  :
    if t[0]=='p' :
        thread_main = producteur
    else :
        thread_main = consommateur
    threads .append(  threading.Thread(target=thread_main, args=(tampon,)))

for t in threads :
    t.start()

for t in threads :
    t.join()
```


6 Thread en Python
Producteurs – Consommateurs
Sémaphores

Solution Problème des Producteurs-Consommateurs par Sémaphores de comptage

Avec Exclusion mutuelle
puis gestion tampon vide
puis gestion tampon plein

A vous…

6 Thread en Python
Producteurs – Consommateurs
Moniteurs

```
import threading

# Ressource Critique
tampon = Tampon_lifo(2)
moniteur  =  Moniteur_PC (tampon)

# Fonction principale des threads "producteurs"
def producteur(moniteur)  :
    …
    moniteur_deposer(moniteur, '1')
    …

# Fonction principale des threads "consommateurs"
def consommateur(moniteur)  :
    …
    elt=moniteur_retirer(moniteur);
    …

# Fonction principalesde démarrage et de création des threads
nomsThreads  =  ["c1","p1", "p2", "p3", "p4", "c2", "c3", "c4 "  ]
Threads = []

for t in nomsThreads  :
    if t[0]=='p' :
        thread_main = producteur
    else :
        thread_main = consommateur
        threads .append( threading.Thread(target=thread_main, args=(moniteur,)))

for t in threads :
    t.start()

for t in threads :
    t.join()
```


6 Threads en Python
Producteurs – Consommateurs
Moniteurs

Solution Producteurs Consommateurs : Moniteur
Avec Exclusion Mutuelle

```
import threading

# Définition du moniteur

# Variables d'état (ressources critiques à protéger par exclusion mutuelle)

Class MoniteurPC :
    def: __init__(self,tampon):
        self.tampon= tampon

# Points d'entrée

def moniteur_deposer(moniteur, element)  :
…
        # code en E.M. sur mutex
        tampon_deposer(tampon, element)
        # fin d'E.M.
…

def moniteur_retirer( moniteur ) :
…
        # code en E.M. sur mutex
        element = tampon_etirer (tampon)
        # fin d'E.M.
…
    return element

/* fin de definition du moniteur */
```

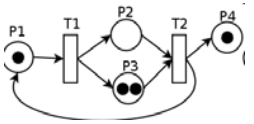
+ Ajouter les variables condition ….

6 Thread en Python
Producteurs – Consommateurs
Moniteurs

Solution Producteurs-Consommateurs par Moniteur
- Avec Exclusion mutuelle
- puis gestion tampon vide
- puis gestion tampon plein

A vous…

Annexe 1: les Réseaux de Petri



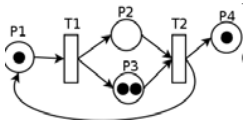
Annexe 1: les Réseaux de Petri

Enjeux

- Système à architecture parallèle :
  - Statique
  - Dynamique
  - Non déterminisme *
  - Explosion combinatoire des états possibles
  - Conception liée aux contraintes, blocages, erreurs

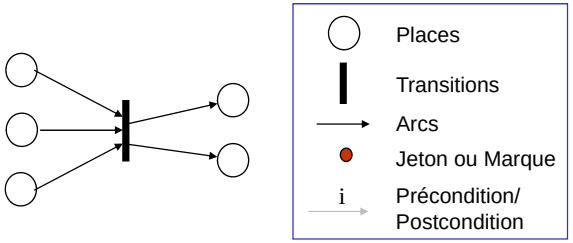
Problème : Modéliser le comportement complexe de ces systèmes

=> Réseaux de Pétri (Petri nets)



Annexe 1: les Réseaux de Petri

Langage/représentation graphique :



Dynamique :

Si toutes les *places entrantes* d'une transition t sont *marquées (précondidtions)*, la transition est **activable**.

Lorsque la transistion t est **activée**, les marques sont retirées des *places entrantes* et ajoutés à toutes les *places sortantes (postconditions)*

Annexe 1: les Réseaux de Petri

Plus formellement :

un graphe orienté bipartite (TxP) doublement valué (M,In,Out)

(P, T, M:P->IN+, In:TxP->IN+, Out:TxP->IN+)

P: ensemble de m places p
T: ensemble de n transitions t

M: fonction P->IN+, M(p)=nb marques de p (*dynamique*)
0 = pas de marque sur p
In: fonction TxP->IN+, In(t,p)=précondition arc p->t
Out : fonction TxP->IN+, out(t,p)=postcondition arc t->p
0 = pas d'arc entre p et t

Dynamique :

Si $\forall p \in Pin(t), M(p) \geq PreCond(p,t)$ alors : (t activable)

$\forall p \in Pin(t), \quad M'(p) := M(p) - In(p,t)$

et $\forall p \in Pout(t), M'(p) := M(p) + Out(p,t)$ (t activée)

Avec Pin(t) = { p∈P | In(t,p) > 0 }, ensemble des places entrantes en t
Pout(t) = { p∈P | Out(t,p) > 0 }, ensemble des places sortantes de t
M(p) = marque de p *avant* activation de t (étape i)
M'(p) = marque de p *après* activation de t (dynamique) (étape i+1)

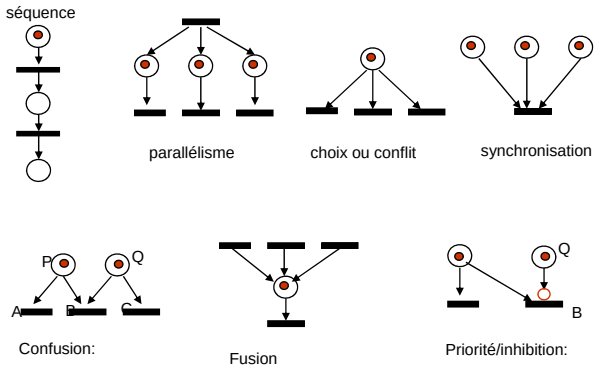
Trace : l'évolution des marques sur les m places à chaque étape i : 1...k

TM=(M1,M2,M3,…, Mi,…, Mk)

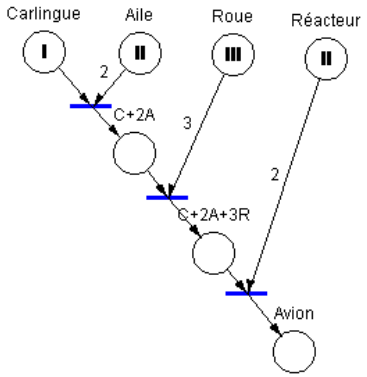
où Mi : marques des m places à l'étape i (vecteur de m entiers)
Mi(pj) : marque de pj (la j-éme place) à l'étape i,
donc TM : matrice(k,m), avec TM(i,j)=Mi(pj)
On doit aussi garder Ti l'ensemble des transitions activées à l'étape i
-- on note M1<T2>M2 le passage de M1 à M2 (automate)
Donc TM=(M1,T2,M2,T2,M3,T3, …, Ti,Mi, …, Tk, Mk)

Annexe 1: les Réseaux de Petri

Quelques enchaînements possibles



Annexe 1: les Réseaux de Petri
Exemple chaine assemblage avion

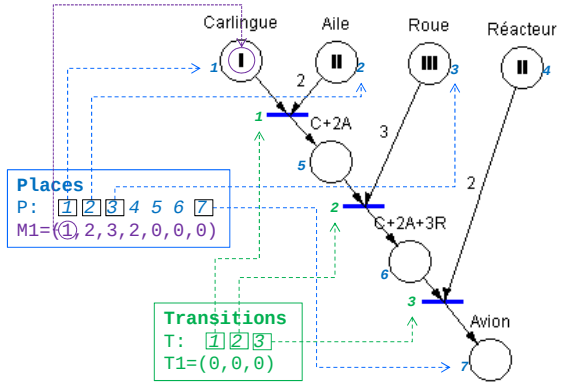


Assemblage d'un avion :

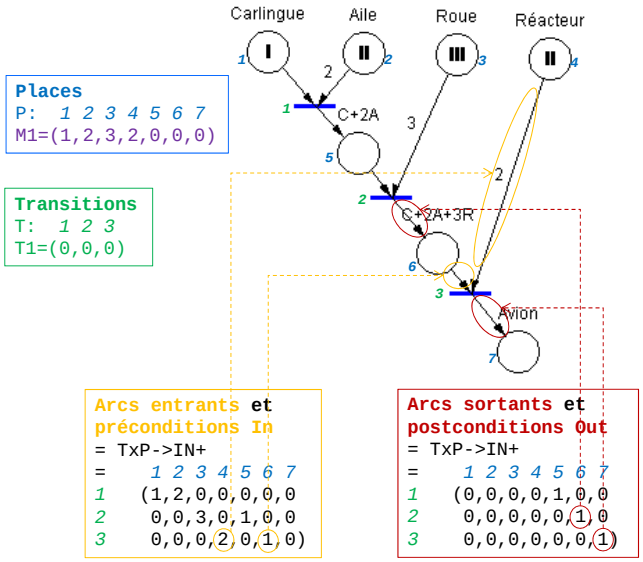
- 1 carlingue
- 2 aile
- 3 roues
- 2 réacteurs

Dans cet ordre uniquement

Annexe 1: les Réseaux de Petri
Exemple chaine assemblage avion



Annexe 1: les Réseaux de Petri
Exemple chaine assemblage avion



Annexe 1: les Réseaux de Petri
Exemple chaine assemblage avion

Dynamique : trace graphique et formelle