1/ Requête : La demande d'aquisition d'une ressource partagée R
{ Si R est disponible alors aquisition
sinon Attente du thread

2/ utilisation de la Ressource partagée

3/ Libération : Chaque ressource aquise doit être libéré

## Deadlock (Interblocage)

- Dans des **conditions normales** de fonctionnement, un processus/thread ne peut utiliser une ressource qu'en suivant la séquence suivante :

Requête – Utilisation - Libération

Graphe d'allocation des ressources et Notations :

$P_i \longrightarrow R_j$ } Le processus/thread demande la ressource partagée

$T_i \longrightarrow R_j$ }

Arc de requête

$R_j \longrightarrow P_i$ } La ressource est allouée aux processus/threads

$R_j \longrightarrow T_i$ }

lien d'allocation

## Deadlock (Interblocage)

- Dans des **conditions normales** de fonctionnement, un processus/thread ne peut utiliser une ressource qu'en suivant la séquence suivante :

On modélise les processus/threads par des cercles : (P) (T)

" " Ressource partagée par [ ], son instance par des points


Exemple :

$P_i = \{P_1, P_2, P_3\}$

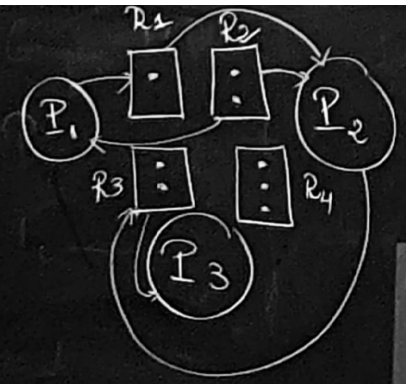
$R_j = \{R_1, R_2, R_3, R_4\}$

$A = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_3 \rightarrow P_1, R_3 \rightarrow P_3\}$

**Deadlock (Interblocage)**

- Dans des **conditions normales** de fonctionnement, un processus/thread ne peut utiliser une ressource qu'en suivant la séquence suivante :

Requête - Utilisation - Libération




→ : Arc de requête = demande de ressource

→ : Arc d'allocation

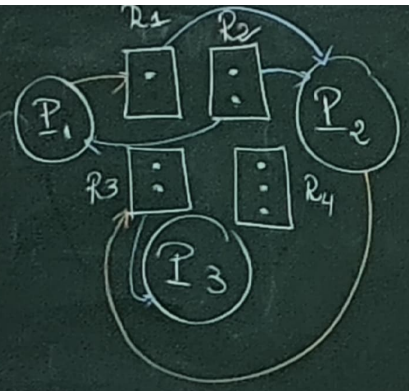
| type de ressource | Instance |
|---|---|
| R1 | 1 |
| R2 | 2 |
| R3 | 2 |
| R4 | 3 |

## Deadlock (Interblocage)

Aquisition des ressources partagées

- Dans des **conditions normales** de fonctionnement, un processus/thread ne peut utiliser une ressource qu'en suivant la séquence suivante :

Requête – Utilisation – Libération




→ : Arc de requête
=
demande
de ressource

→ : Arc d'allocation
=
Aquisition des
ressources partagées

| type de ressource | Instance |
|---|---|
| R1 | 1 |
| R2 | 2 |
| R3 | 2 |
| R4 | 3 |

Exemple 2 : Robo Pathfinder

On considère 3 processus :

$P_1$ : pour le pilotage manuel qui reçoit les ordres wifi et opère avec les moteurs

$P_2$ : envoie le flux des vidéos via la liaison wifi

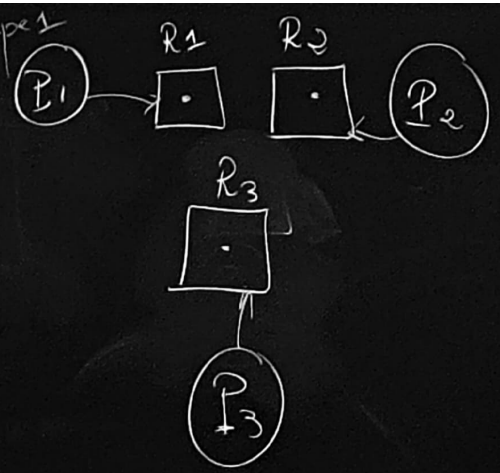
$P_3$ : autotest matériel

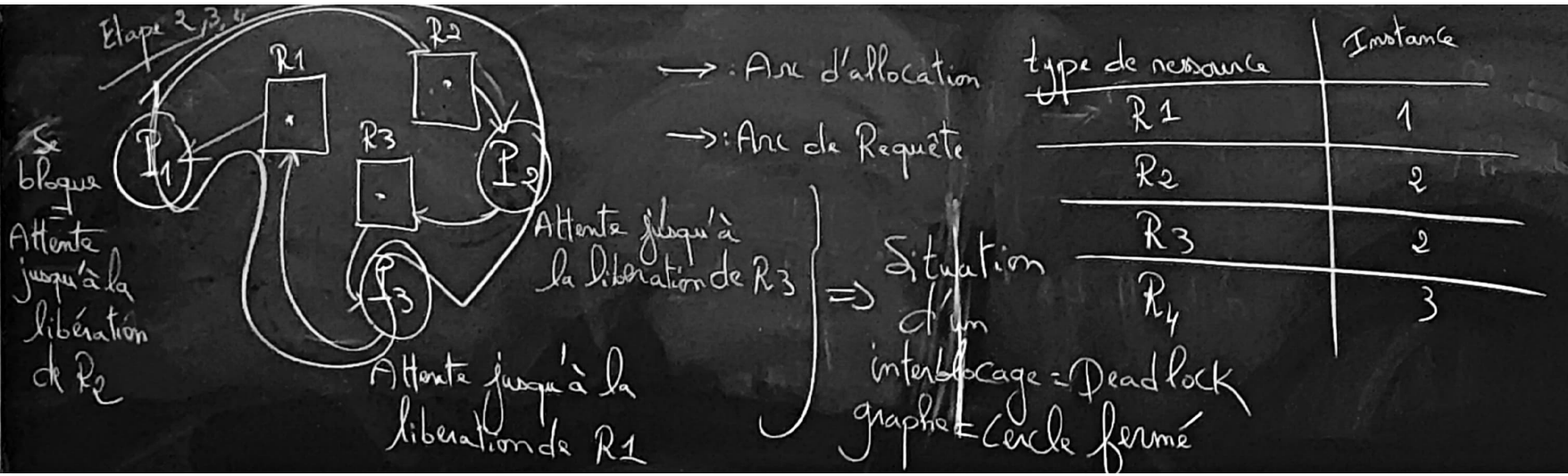
les 3 processus effectuent ces tâche simultanément

| $P_1$ : Pilotage manuel | $P_2$ : envoi du flux vidéo | $P_3$ : autotest matériel |
|---|---|---|
| Demander R1 (moteurs) | Demander R2 (wifi) | Demander R3 (caméra) |
| Acquiert R1 | Acquiert R2 | Acquiert R3 |
| Demander R2 | Demander R3 | Demander R1 |
| Libérer R1 | Libérer R2 | Libérer R3 |
| Libérer R2 | Libérer R3 | Libérer R1 |

Etape 1

Etape 2






| type de ressource | Instance |
|---|---|
| R1 | 1 |
| R2 | 2 |
| R3 | 2 |
| R4 | 3 |

**Les sémaphores** : Pour gérer les problèmes liés à la concurrence et imposer une synchronisation entre les processus/threads ⇒ Indiquer la disponibilité des ressources

- Sémaphore binaire : 0 ou 1 : mutex
- Sémaphore variables : les valeurs varient jusqu'à N valeurs

S → 0 (non disponible) P(S) = S - 1
→ 1 (disponible)

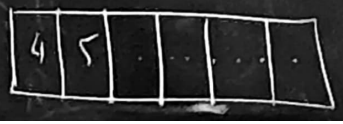
| S > 0 |
| --- |
| S <= 0 |

On utilise 2 fonctions principales : P(S) : Proberen ⇒ tester la disponibilité de la ressource (Puis-je utiliser cette ressource ? décrementer S)

V(S) : Verhogen ⇒ Libère la ressource (Incrémenter S) V(S) = S + 1

Exemple : Producteurs / Consommateurs

- Threads Producteurs : threads qui produisent des données (remplir tableaux, ...)
- Threads Consommateurs : threads qui consomment les données.
- Ressource partagée : Buffer




1/ Situation de blocage pour les threads producteurs : buffer plein
⇒ les threads ne peuvent plus produire des données

2/ Situation de blocage pour threads consommateurs : buffer vide
⇒ Les threads ne peuvent consommer des données

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define N 10
pthread_mutex_t mutex; // imposer un mutex pour notre buffer
sem_t full;
sem_t empty;
```

```
pthread_mutex_init(&mutex);
sem_init(&empty, 0, N); // buffer vide
sem_init(&full, 0, 0); // buffer vide
```

```
void *producer(void *arg){
  int item;
  while(true){
    item = produce(item);
    sem_wait(&empty); // équivalent à P(S) = teste s'il y a des cases vides
    pthread_mutex_lock(&mutex);
      item = put(item);
    pthread_mutex_unlock(&mutex);
    sem_post(&full); // V(S) indique qu'il y a des cases remplies
```

Section critique

Verrouillage du buffer

```
void *consumer(void *arg){
  int item;
  while(true){
    sem_wait(&full); // P(S) => Est ce qu'il y a une/des case(s) remplie(s)
    pthread_mutex_lock(&mutex);
      item = remove(item);
    pthread_mutex_unlock(&mutex);
    sem_post(&empty); // Indiquer qu'il y a des cases vides
```