Chapitre VI : Mutex et Sémaphore

Cours Système Exploitation II

1 LInfo

2024/2025

Dr Sana BENZARTI ISIMM

Batman et Robin création de threads

- Ecrire un code en c dans lequel thread Batman affiche j'utilise ma Batmobile et thread Robin affiche j'utilise ma moto .

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>


void *batman(void *arg) {
   printf("J'utilise ma Batmobile.\n");
   pthread_exit(NULL);
}


void *robin(void *arg) {
   printf("J'utilise ma moto.\n");
   pthread_exit(NULL);
}

int main() {  // thread principal
   pthread_t batman_thread, robin_thread;
   int rc;

   rc = pthread_create(&batman_thread, NULL, batman,
NULL);  // création du thread Batman
 if (rc) {
      printf("Erreur lors de la création du thread Batman.
Code de retour : %d\n", rc);
      exit(-1);
   }

   rc = pthread_create(&robin_thread, NULL, robin,
NULL);
// création du thread Robin
   if (rc) {
      printf("Erreur lors de la création du thread Robin.
Code de retour : %d\n", rc);
      exit(-1);
   }

   pthread_join(batman_thread, NULL);
   pthread_join(robin_thread, NULL);

   printf("Les threads Batman et Robin ont terminé leur
exécution.\n");

   pthread_exit(NULL);
}
```

Batman et Robin création de threads

- Ecrire un code en c dans lequel vous déclarez deux variables globales : batmobile et moto initialises a 0 .
- Thread Batman incrémente batmobile et décrémente moto
- Thread Robin affiche décrémente batmobile et incrémente moto.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int batmobile = 0;  // variable partagée
int moto = 0;  // variable partagée

void *batman(void *arg) {
   printf("Batman utilise la Batmobile.\n");
    batmobile++;
   printf("Batman utilise la moto.\n");
    moto --;
 pthread_exit(NULL);
}

void *robin(void *arg) {
    printf("Robin utilise la moto.\n");
    moto++;
    printf("Robin attend la Batmobile.\n");
    batmobile --;
    printf("Nombre d'utilisations de la moto : %d\n", moto);
    pthread_exit(NULL);
}

int main() {  // thread principal
    pthread_t batman_thread, robin_thread;
    int rc;

    rc = pthread_create(&batman_thread, NULL, batman, NULL);  // création du thread Batman
    if (rc) {
        printf("Erreur lors de la création du thread Batman. Code de retour : %d\n", rc);
        exit(-1);
    }

    rc = pthread_create(&robin_thread, NULL, robin, NULL);  // création du thread Robin
    if (rc) {
        printf("Erreur lors de la création du thread Robin. Code de retour : %d\n", rc);
        exit(-1);
    }

    pthread_join(batman_thread, NULL);
    pthread_join(robin_thread, NULL);

    printf("Les threads Batman et Robin ont terminé leur exécution.\n");

    pthread_exit(NULL);
}
```

Problèmes liés à la concurrence

Condition de course : condition race

- Plusieurs threads accèdent **simultanément** à une même **variable ou ressource** sans synchronisation.
- **Exemple :** Deux threads (batman et robin) modifient la même variable batmobile ce qui provoque un résultat incohérent.

- Se produit lorsque le **résultat dépend de l'ordre d'exécution** non déterministe des threads.
- **Exemple :** Un thread lit une variable pendant qu'un autre la modifie
- → comportement imprévisible.

Deadlock (Interblocage)

- Deux ou plusieurs threads **attendent indéfiniment** des ressources **verrouillées par les autres :** aucun ne peut continuer.

- **Exemple :**
- Thread A a verrouillé ressource X, attend ressource Y.
- Thread B a verrouillé ressource Y, attend ressource X → blocage total.

Exemple concret : Mission Pathfinder

En 1997, la mission Mars Pathfinder rencontre un problème alors que le robot est déjà sur Mars. Après un certain temps, des données sont systématiquement perdues. Les ingénieurs découvrent alors un bug lié à la synchronisation de plusieurs tâches. Les éléments incriminés étaient les suivants :

- une mémoire partagée, qui était protégée par un mutex
- une gestion de bus sur la mémoire partagée, qui avait une priorité haute
- une écriture en mémoire partagée (récupération de données), qui avait la priorité la plus basse
- une troisième routine de communication, avec une priorité moyenne, qui ne touchait pas à la mémoire partagée

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

- La requête : le thread fait une demande pour utiliser la ressource. Si cette demande ne peut pas être satisfaite immédiatement, parce que la ressource n'est pas disponible, le thread demandeur se met en état attente jusqu'à ce que la ressource devienne libre.

Deadlock (Interblocage)

- Utilisation : Le thread peut exploiter le ressource.

- Libération : Le thread libère la ressource qui devient disponible pour les autres threads éventuellement en attente.

Deadlock (Interblocage) : Les conditions

Une situation d'interblocage peut survenir si les quatre conditions suivantes se produisent simultanément (Habermann) :

- 1. Accès exclusif : Les ressources ne peuvent être exploitées que par un seul processus/thread à la fois.

- 2. Attente et occupation : Les processus/threads qui demandent de nouvelles ressources gardent celles qu'ils ont déjà acquises et attendent la satisfaction de leur demande

Deadlock (Interblocage) : Les conditions

- 3. Pas de réquisition : Les ressources déjà allouées ne peuvent pas être réquisitionnées.

- 4. Attente circulaire : Les processus en attente des ressources déjà allouées forment une chaîne circulaire d'attente.

GRAPHE D'ALLOCATION DES RESSOURCES

- On peut décrire l'état d'allocation des ressources d'un système en utilisant un graphe.
- Ce graphe est composé de N nœuds et de A arcs. L'ensemble des nœuds est partitionné en deux types :

T={T1, T2, …, Tm} : l'ensemble de tous les threads

P={P1, P2, …, Pm} : l'ensemble de tous les processus

R={R1, R2, …, Rn} l'ensemble de tous les types de ressources du système

GRAPHE D'ALLOCATION DES RESSOURCES

Un arc allant du processus Pi ou du thread Ti vers un type de ressource Rj est noté Pi→Rj ou Ti→Rj ;

- il signifie que le processus Pi/le thread Ti a demandé une instance du type de ressource Rj.

Un arc du type de ressourcé Rj vers un processus Pi ou un thread Ti est noté Rj→Pi

il signifie qu'une instance du type de ressource Rj a été alloué au processus Pi/thread Ti.

GRAPHE D'ALLOCATION DES RESSOURCES

- Un arc Pi→Rj ou Ti→Rj est appelé arc de requête.
- Un arc Rj→Pi ou Rj→Ti est appelé arc d'affectation.

- Graphiquement, on représente chaque processus Pi / thread Ti par un cercle et chaque type de ressource Rj comme une rectangle.

- Puisque chaque type de ressource Rj peut posséder plus d'une instance, on représente chaque instance comme un point dans le rectangle.

GRAPHE D'ALLOCATION DES RESSOURCES

Ensemble des processus P={P1, P2, P3}

Ensemble des ressources R={R1, R2, R3, R4}

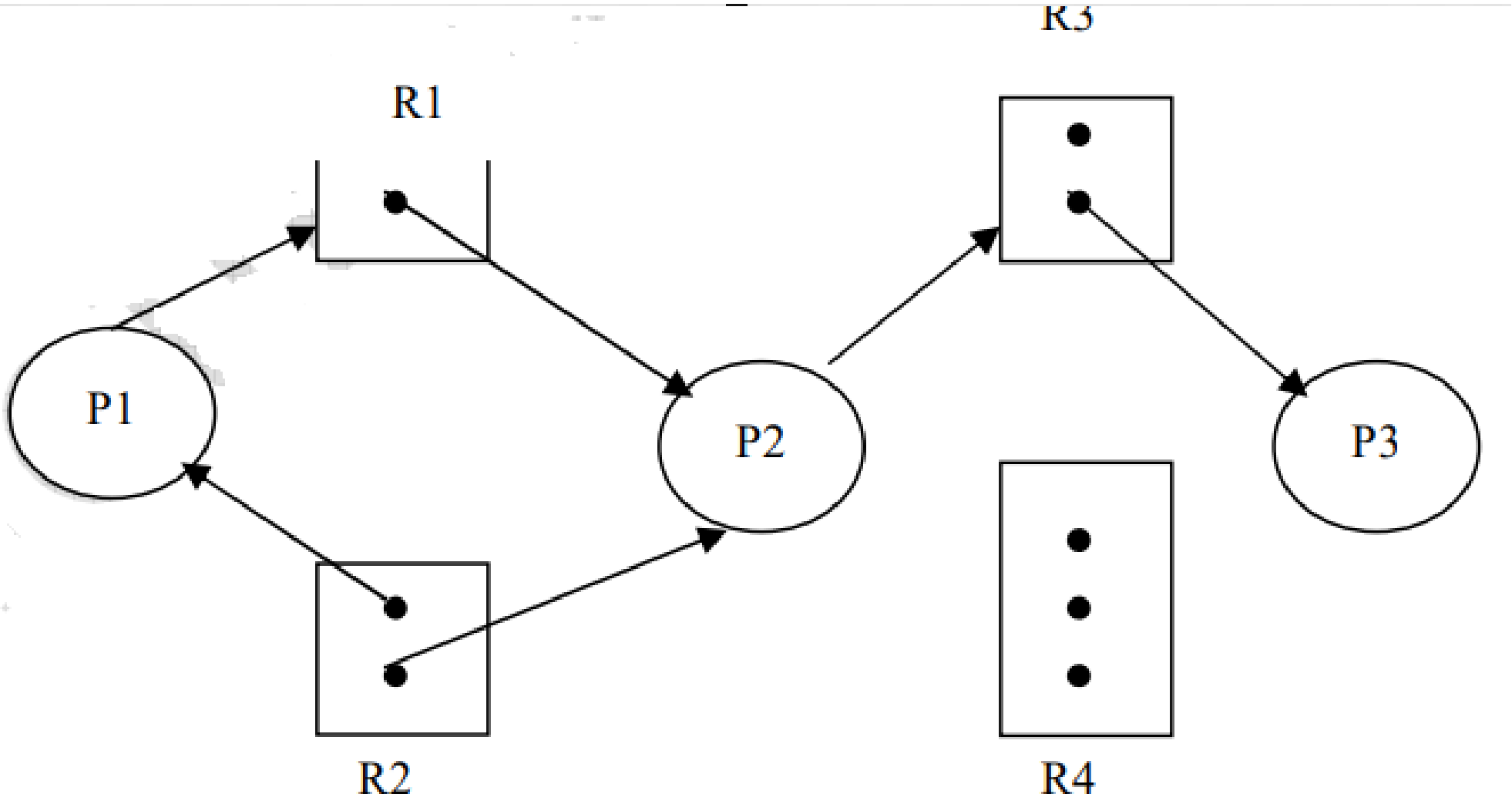
Ensemble des arcs A={P1→R1, P2→R3, R1→P2, R2→P2, R2→P1, R3→P3}

GRAPHE D'ALLOCATION DES RESSOURCES

- Le nombre d'instances par ressources est donné par ce tableau :
- Instance = une version ou une unité spécifique de cette ressource

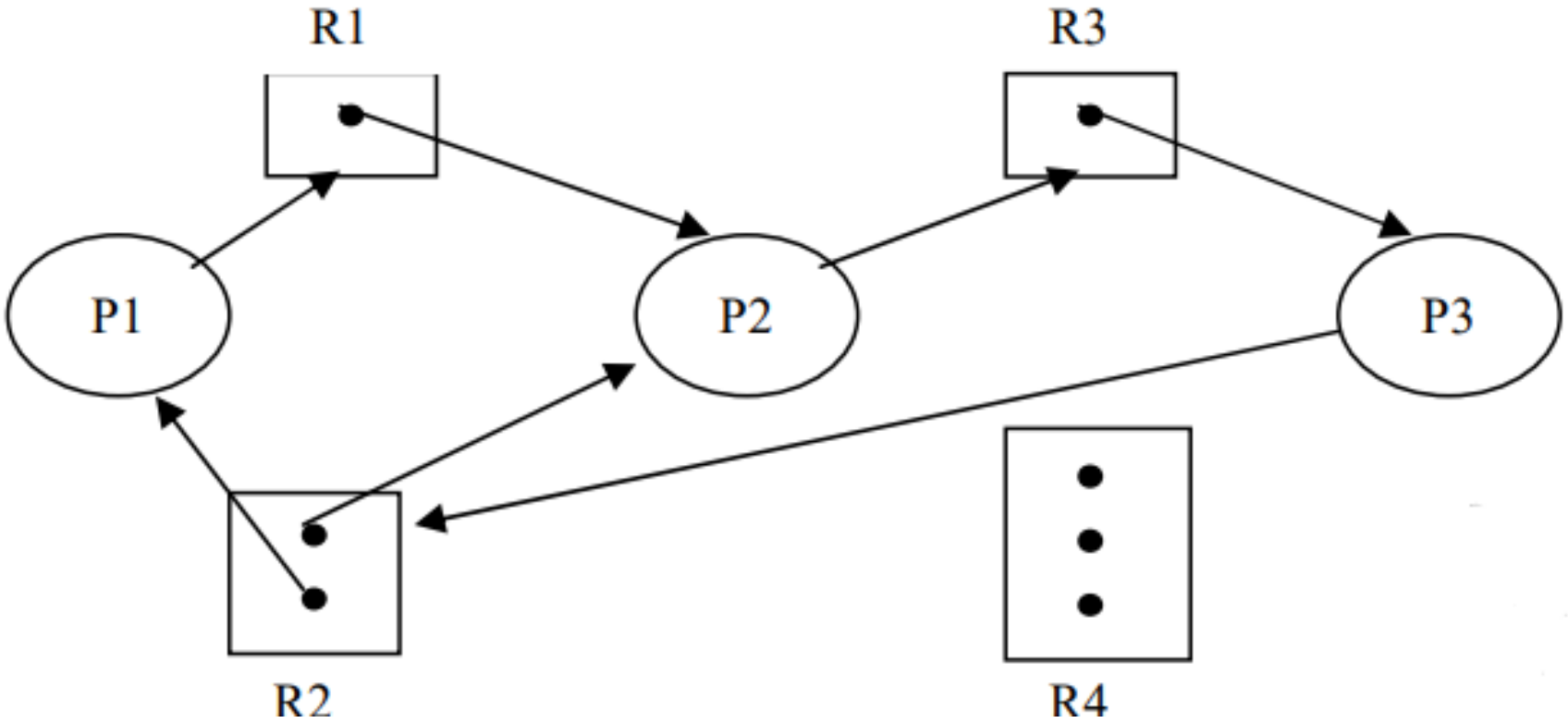
Type de ressources	Nombre d'instances
R1	1
R2	2
R3	2
R4	3

GRAPHE D'ALLOCATION DES RESSOURCES



GRAPHE D'ALLOCATION DES RESSOURCES

Situation d'interblocage



Simulation de l'interblocage

- Nous allons nous même simuler un interblocage dans une situation qui serait assez similaire à celle de la liaison martienne. On va considérer un robot qui a trois ressources :

- Des moteurs qui lui permettent de se déplacer
- Une liaison wifi qui lui permet de communiquer
- Une caméra qui filme son environnement

Simulation de l'interblocage

- Ce robot a trois processus que l'on notera P1, P2 et P3 :

- P1 est le pilotage manuel qui reçoit les ordres par le wifi et opère les moteurs
- P2 envoie le flux vidéo via la liaison wifi
- P3 est le thread qui fait un autotest matériel, hors liaison wifi

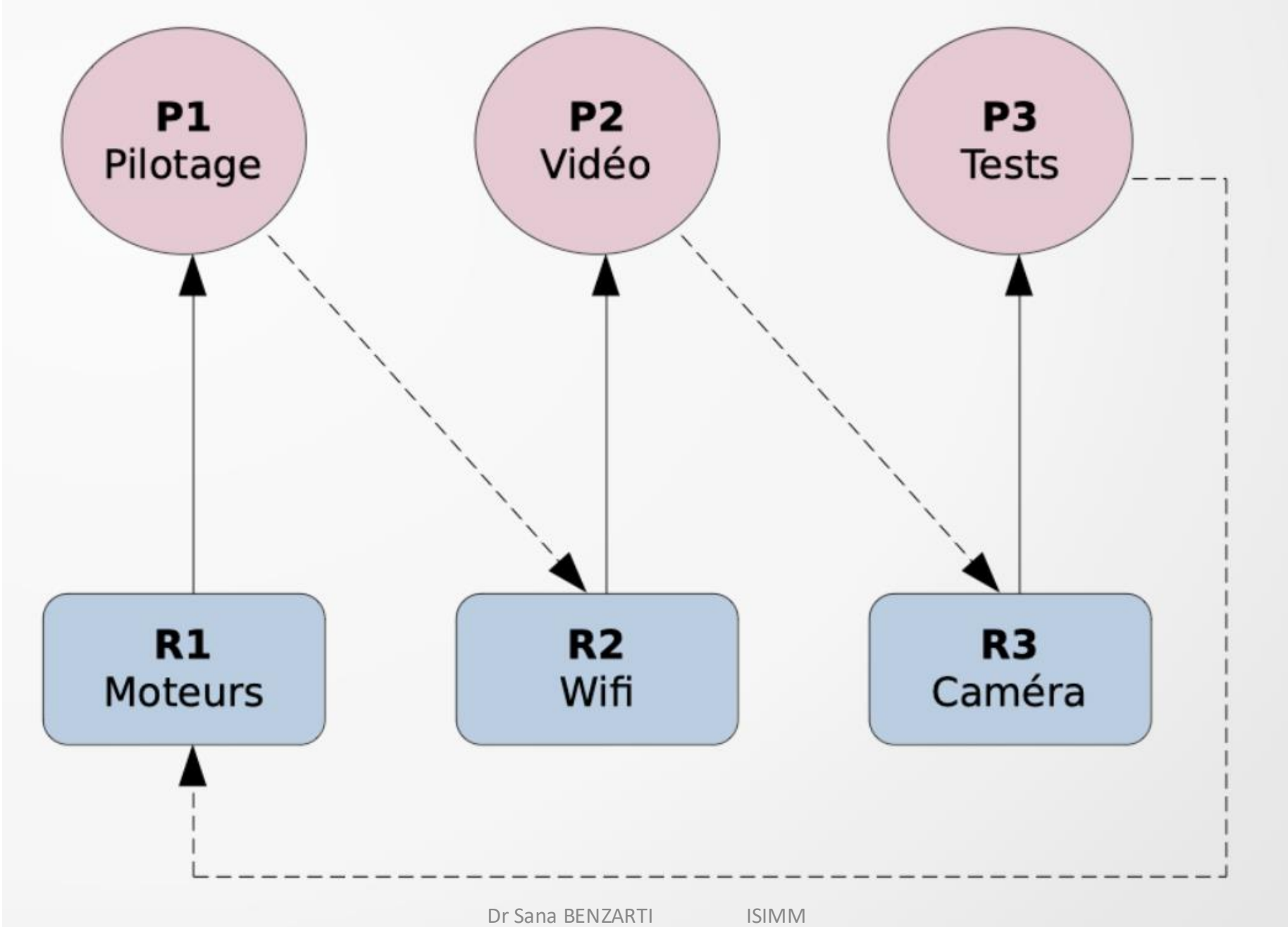
- Le robot effectue les 3 taches en parallèle. Cela peut se résumer dans le tableau suivant

P1 : pilotage manuel	P2 : envoi de flux vidéo	P3 : auto-test matériel
Demande R1 (moteurs)	Demande R2 (wifi)	Demande R3 (camera)
Demande R2 (wifi)	Demande R3 (camera)	demande R1 (moteurs)
Libère R1 (moteurs)	Libère R2 (wifi)	Libère R3 (Caméra)
Libère R2(wifi)	Libère R3 (caméra)	Libère R1 (moteurs)

- Cette séquence d'instruction peut se dérouler parfaitement bien, mais on peut arriver à une situation d'interblocage par un cycle.

- P1 demande la ressource R1, disponible, et la bloque
- P2 demande la ressource R2, disponible, et la bloque
- P3 demande la ressource R3, disponible et la bloque

- étape suivante, P1 demande R2 mais doit attendre que P2 la libère.
- P2 demande R3, qui est bloquée par P3
- Et P3 demande R1 qui est bloqué par P1


P1
Pilotage
P2
Vidéo
P3
Tests
R1
Moteurs
R2
Wifi
R3
Caméra

Problème de la famine : Starvation

- Un thread **n'a jamais accès** à une ressource car d'autres threads l'accaparent constamment.

- **Exemple :** Un thread de basse priorité attend un verrou que des threads prioritaires prennent en boucle.

- On dit qu'un processus/thread est dans une situation de famine s'il attend indéfiniment une ressource (qui est éventuellement occupée par d'autres processus). Notons que l'interblocage implique nécessairement une famine, mais le contraire n'est pas toujours vrai.

Monitor (ou sémaphore)

- Utilisé pour synchroniser l'accès à une ressource partagée.
- Cette ressource peut-être un segment d'un code donné.
- Un thread accède à cette ressource par l'intermédiaire de ce moniteur.
- Ce moniteur est attribué à un seul thread à la fois.
- Pendant que le thread exécute la ressource partagée aucun autre thread ne peut y accéder.

Monitor (ou sémaphore)

- Le thread libère le monitor dès qu'il a terminé l'exécution du code synchronisé.

- Nous assurons ainsi que chaque accès aux données partagées est bien « mutuellement exclusif ».

- Nous allons utiliser pour cela les « mutex » une abréviation pour « Mutual Exclusion » ou « exlusion mutuelle ».

Mutex

- Un mutex est un verrou possédant deux états : déverrouillé (pour signifier qu'il est disponible) et verrouillé (pour signifier qu'il n'est pas disponible).

- Mutex est une variable d'exclusion mutuelle.

- Un mutex est du type « pthread_mutex_t ».


```
pthread_mutex_t mutex;
```

Un Mutex contient (de manière conceptuelle) :

Élément contenu ou géré par le mutex	Rôle
État du verrou (locked/unlocked)	Savoir si le mutex est pris par un thread ou non
Identifiant du thread possesseur	Pour savoir qui détient le verrou
Liste d'attente des threads	Threads qui attendent de prendre le verrou
Attributs internes	(optionnels) comme récursivité, priorité, etc.

Mutex

- Avant de pouvoir utiliser un mutex, il faudra l'initialiser.
- Nous pouvons initialiser un mutex à l'aide d'une macro.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Nous pouvons initialiser aussi un mutex à l'aide de la fonction «pthread_mutex_init ».

correspond au mutex à initialiser

```
int pthread_mutex_init (pthread_mutex_t* mutex,
                        const pthread_mutexattr_t* mutexattr);
```

les attributs à utiliser lors de l'initialisation. Nous pouvons utiliser la valeur « NULL ».

Mutex

- Pour détruire un mutex, nous utilisons la fonction «pthread_mutex_destroy».

```
int pthread_mutex_destroy (pthread_mutex_t* mutex);
```

Les threads batman et robin avec un mutex

Utiliser Mutex

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int batmobile = 0;  // variable partagée
int moto = 0;  // variable partagée
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  // mutex pour synchroniser l'accès aux variables partagées

void *batman(void *arg) {
   pthread_mutex_lock(&mutex);  // verrouillage du mutex
   printf("Batman utilise la Batmobile.\n");
   batmobile++;
  printf("Batman utilise la moto.\n");
   moto--;
  pthread_mutex_unlock(&mutex);  // déverrouillage du mutex
   pthread_exit(NULL);
}

void *robin(void *arg) {
   while (batmobile == 0) { }  // attente de la Batmobile
   pthread_mutex_lock(&mutex);  // verrouillage du mutex
   printf("Robin utilise la moto.\n");
   moto++;
   printf("Robin utilise la batmobile.\n");
   batmobile--;
   pthread_mutex_unlock(&mutex);  // déverrouillage du mutex
   pthread_exit(NULL);
}
```

```
int main() {  // thread principal
   pthread_t batman_thread, robin_thread;
   int rc;
   rc = pthread_create(&batman_thread, NULL, batman, NULL);  // création du thread Batman
   if (rc) {
      printf("Erreur lors de la création du thread Batman. Code de retour : %d\n", rc);
      exit(-1);
   }

   rc = pthread_create(&robin_thread, NULL, robin, NULL);  // création du thread Robin
   if (rc) {
      printf("Erreur lors de la création du thread Robin. Code de retour : %d\n", rc);
      exit(-1);
   }

   pthread_join(batman_thread, NULL);
   pthread_join(robin_thread, NULL);

   printf("Les threads Batman et Robin ont terminé leur exécution.\n");

   pthread_exit(NULL);
}
```

Les sémaphores

- Un **sémaphore** est une variable spéciale utilisée pour :

- **Contrôler l'accès** à une ou plusieurs ressources partagées.
- **Compter** combien de ressources sont disponibles.

- Il existe deux types :
- **Sémaphore binaire (valeur 0 ou 1)** : agit comme un **mutex**.
- **Sémaphore compteur** : permet de gérer **N ressources disponibles**.

Les sémaphores

- Un sémaphore est une variable accessible seulement à l'aide des opérations P (Proberen : tester) et V (Verhogen : incrémenter) suivantes:

- P(S) Puis-je ? → S= S-1

- Si S est nulle ou négative alors le processus/thread réalisant P(S) est bloqué dans un file d'attente spécifique de chaque sémaphore.

- P est utilisée pour prendre (acquérir) le sémaphore avant d'accéder à la ressource partagée.

Les sémaphores

- V(S) Vas-y → S=S+1
- Si S > 0 alors un des processus/thread de la file d'attente est débloqué.
- V est utilisée pour libérer (relâcher) le sémaphore après avoir terminé l'accès à la ressource partagée.

Sémaphore sous Linux

prend comme argument un pointeur vers le sémaphore à initialiser.
Value est la valeur initiale, positive ou nulle, du sémaphore.

```
#include <semaphore.h>

int sem_init(sem_t *sem, unsigned int value);

 int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);

 int sem_post(sem_t *sem);
```

libérer un sémaphore qui a été initialisé

•Équivalent de P(S) : tester la valeur d'un sémaphore. Si la valeur du sémaphore est positive, elle est décrémentée d'une unité et la fonction réussit

Équivalent de V(S)

Opérations fondamentales

- Les sémaphores possèdent deux opérations atomiques

Opération	Description
P() ou wait()	Décrémente la valeur du sémaphore. Si elle devient < 0, le processus attend.
V() ou post()	Incrémente la valeur du sémaphore. Si des processus attendent, l'un d'eux est réveillé.

En C (POSIX) :

- sem_wait(&sem); ← équivalent à P()
- sem_post(&sem); ← équivalent à V()

Attente active (Busy Waiting)

- L'**attente active** consiste à faire tourner un thread **en boucle**, **en testant une condition**, jusqu'à ce qu'elle devienne vraie.

```
while (!condition) {
    // ne fait rien, juste tourne en boucle
}
```

Attente passive (Busy Waiting)

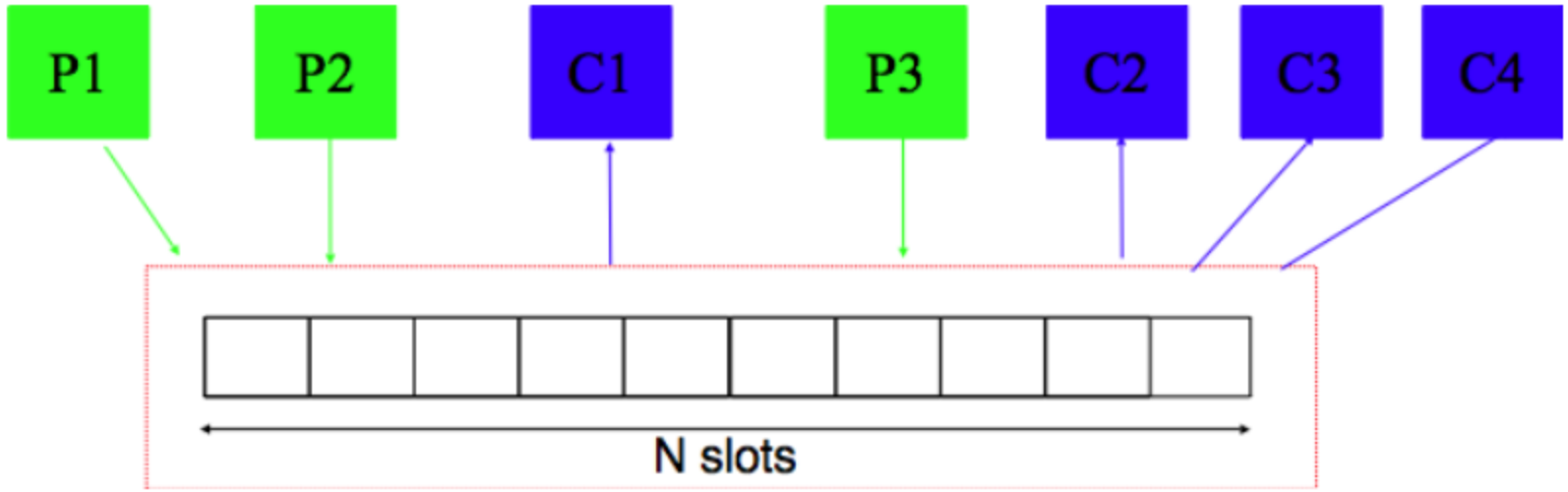
- L'**attente passive** consiste à **bloquer le thread** en le mettant en **sommeil** ou en **attente bloquante**, jusqu'à ce qu'il soit réveillé par un signal ou un événement.

```
wait();            // avec sémaphore ou condition variable
sleep(1);          // pour attendre 1 seconde
pthread_cond_wait();  // attente d'un signal sur une condition
```

Type d'attente	Utilisation CPU	Efficacité	Exemples
Attente active	Oui (en boucle)	Moins bonne	while (!condition) {}
Attente passive	Non (bloquée)	Meilleure	wait(), sleep(), cond_wait()

Exemple classique : Producteurs / Consommateurs

- les *producteurs* : Ce sont des threads qui produisent des données et placent le résultat de leurs calculs dans une zone mémoire accessible aux consommateurs.
- les *consommateurs* : Ce sont des threads qui utilisent les valeurs calculées par les producteurs.
- Ces deux types de threads communiquent en utilisant un buffer qui a une capacité limitée à *N* places comme illustré dans la figure ci-dessous.



Problème des producteurs-consommateurs

Exemple classique : Producteurs / Consommateurs

- Le buffer étant partagé entre les producteurs et les consommateurs, il doit nécessairement être protégé par un mutex.

- Les producteurs doivent pouvoir ajouter de l'information dans le buffer partagé tant qu'il y a au moins une place de libre dans le buffer.

- Un producteur ne doit être bloqué que si tout le buffer est rempli. Inversement, les consommateurs doivent être bloqués uniquement si le buffer est entièrement vide. Dès qu'une donnée est ajoutée dans le buffer, un consommateur doit être réveillé pour traiter cette donnée.

Exemple classique : Producteurs / Consommateurs

- On utilise :

Élément	Rôle
mutex (binaire)	Protéger l'accès au tampon (section critique)
empty (compteur)	Compter les places vides dans le tampon
full (compteur)	Compter les éléments présents dans le tampon

Exemple classique : Producteurs / Consommateurs

```
// Initialisation
#define N 10 // places dans le buffer
pthread_mutex_t mutex;
sem_t empty;
sem_t full;

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

Exemple classique : Producteurs / Consommateurs

- **empty (cases vides)** :
- Ce sémaphore représente le **nombre de places vides dans le buffer**.
- Au départ, le tampon est **vide**, donc il y a **N places disponibles**.
- ➤ On l'initialise à N.

- **full (cases pleines)** :
- Ce sémaphore représente le **nombre de données disponibles à consommer**.
- Au départ, le buffer est **vide**, donc il n'y a **aucune donnée**.
- ➤ On l'initialise à 0.

Exemple classique : Producteurs / Consommateurs

Tout d'abord, le producteur est mis en attente sur le sémaphore empty. Il ne pourra passer que s'il y a au moins une place du buffer qui est vide. Lorsque la ligne **sem_wait(&empty);** réussit, le producteur s'approprie le mutex et modifie le buffer de façon à insérer l'élément produit (dans ce cas un entier). Il libère ensuite le mutex pour sortir de sa section critique.

```
// Producteur
void producer(void)
{
  int item;
  while(true)
  {
    item=produce(item);
    sem_wait(&empty); // attente d'une place libre
    pthread_mutex_lock(&mutex);
     // section critique
     insert_item();
    pthread_mutex_unlock(&mutex);
    sem_post(&full); // il y a une place remplie en plus
  }
}
```

Par exemple si empty =3 cases vides et full = 0 cases pleines alors la simulation sera comme suit :

Étape	**empty**	**full**	**Producteur fait sem_wait(&empty) ?**
Début (vide)	3	0	passe, devient empty = 2
1er item inséré	2	1	passe, devient empty = 1
2e item inséré	1	2	passe, devient empty = 0
3e item inséré	0	3	bloqué !
Consommateur retire un item	1	2	producteur peut se réveiller

Exemple classique : Producteurs / Consommateurs

Le consommateur quant à lui essaie d'abord de prendre le sémaphore full. Si celui-ci est positif, cela indique la présence d'au moins un élément dans le buffer partagé. Ensuite, il entre dans la section critique protégée par le mutex et récupère la donnée se trouvant dans le buffer. Puis, il incrémente la valeur du sémaphore empty de façon à indiquer à un producteur qu'une nouvelle place est disponible dans le buffer.

```
// Consommateur
void consumer(void)
{
 int item;
 while(true)
 {
   sem_wait(&full); // attente d'une place remplie
   pthread_mutex_lock(&mutex);
    // section critique
    item=remove(item);
   pthread_mutex_unlock(&mutex);
   sem_post(&empty); // il y a une place libre en plus
 }
}
```