Les sémaphores

Définition : un sémaphore est un mécanisme pour la synchronisation ou la protection des sections critiques des processus.

Le **sémaphore** contrôle l'accès à une **ressource partagée** par plusieurs processus, en associant cette ressource avec un **feu** qui permet de s'assurer que pas plus d'**un** processus à un moment peut l'utiliser.

Les sémaphores à compteur

Un **sémaphore** est une structure de données de type objet, et est constitué :

1. une variable entière positive/négative (la valeur du sémaphore)
2. une file d'attente
3. 2 méthodes :
   - **P(sem)** Proberen → décrémentation de la valeur du sémaphore et blocage éventuel du processus appelant
   - **V(sem)** Verhegen → incrémentation de la valeur du sémaphore et éveil d'un processus (action atomique)

- Instantiation : **Init** (**sem**,valeur_initiale)

Les sémaphores à compteur

P(sem) :

-sem.cpt <-- sem.cpt -1
-Si sem.cpt < 0
-alors bloquer le processus appelant
-insérer dans sem.fa le descripteur du processus
-fsi

V(sem) :

-sem.cpt <-- sem.cpt + 1
-Si sem.cpt > 0
-alors extraire un descripteur de sem.fa
-éveiller le processus correspondant
-fsi

Action atomique

Les sémaphores binaires

Un **sémaphore binaire** (ou ***mutex***) contrôle l'accès à la section critique d'un processus ou d'un thread, et sa valeur ne peut être que **0** ou **1**. Le sémaphore binaire est un cas particulier d'un sémaphore à compteur.

On suppose que le sémaphore est initialisé à 1 (val_init = 1).

La stratégie générale d'utilisation d'un sémaphore binaire pour contrôler l'accès à une section critique est la suivante :

Init (exmu, val_init)

P(exmu)
SC(R)
V(exmu)
1er passage : exmu.cpt > 0 (val_init = 1)
2ème passage : exmu.cpt = 0

Exemple :

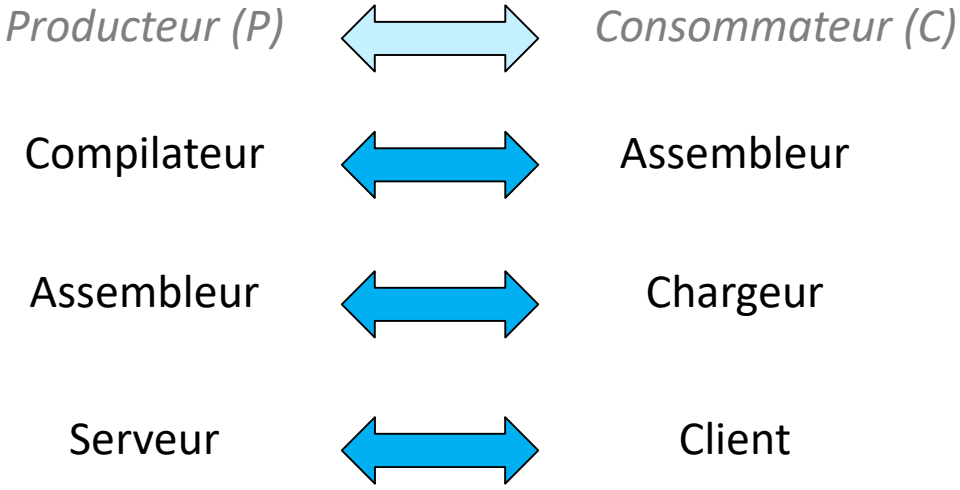
Le problème du producteur-consommateur

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

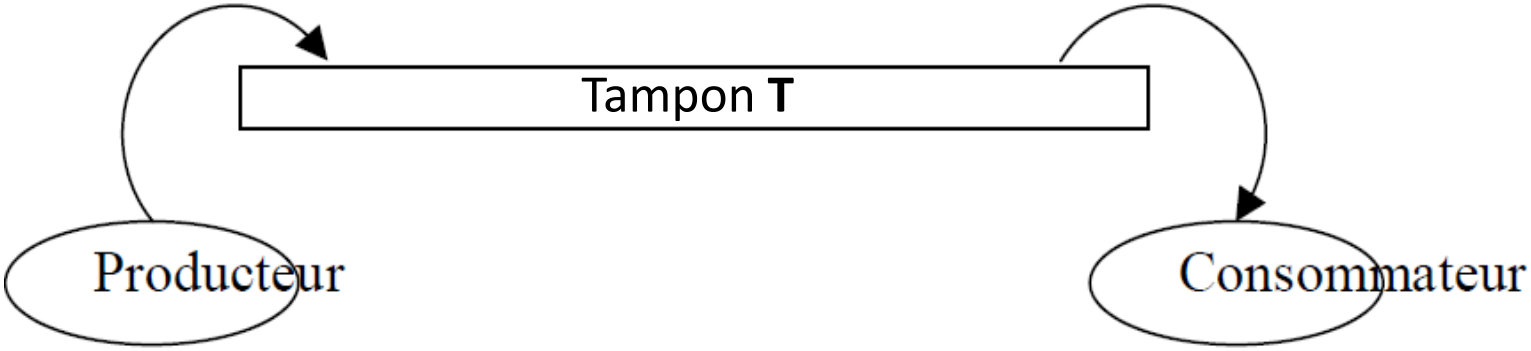
Le problème du producteur-consommateur

Un processus **producteur (P)** produit des informations qui sont consommées par un processus **consommateur (C)**.

Exemples :



Le problème du producteur-consommateur



- Tampon (**T**) borné de taille fixe ***n***
- **T** peut être rempli par **P** et vidé par **C**
- **P** peut produire un élément pendant que **C** consomme un autre
- **P** et **C** doivent être ***synchronisés :***
  - **C** ne tente pas de consommer un élément qui n'a pas encore été produit !
  - **P** ne peut pas insérer un élément dans le tampon s'il est plein !

Le problème du producteur-consommateur

- ***compteur*** est le nombre d'éléments dans le tampon **T**
- **P** doit attendre (s'endormir) si **T** est plein
- **C** doit attendre (s'endormir) si **T** est vide
- Quand **P** voit que **T** était vide et vient d'ajouter un élément dans **T**, il va éveiller **C**
- Quand **C** voit que **T** était plein et vient de retirer un élément, il va éveiller **P**

Producteur :

```
Tque en_vie

produire (R)
insérer(R,tampon)
ftque
```

Consommateur :

```
Tque en_vie

R=extraire(tampon)
      consommer(R)
ftque
```

Le problème du producteur-consommateur

- Comment éviter l'attente active ?
- 2 fonctions :
  - *sleep()*
  - *wakeup()*
- Idée
  - Quand un processus appelle sleep(), il s'endort
  - Si un autre processus appelle wakeup(), il réveille le processus endormi

Le problème du producteur-consommateur

```
#define N 100          /* nbre d'emplacement ds tampon */

int compteur = 0 ;     /* nbre d'objets ds tampon */

void producteur () {
while (VRAI) {
        produire_objet(…) ;
        if (compteur == N) sleep () ;
        mettre_objet(…) ;
        compteur = compteur + 1 ;
        if (compteur == 1)
                wakeup(consommateur) ;
        }
}
```

Le problème du producteur-consommateur

```
void consommateur () {
while (TRUE) {
        if (compteur == 0) sleep() ;
        retirer_objet(…)
        compteur = compteur – 1 ;
        if (compteur == N-1)
                wakeup (producteur) ;
        consommer_objet(…) ;
        }
}
```

Le problème du producteur-consommateur

- Problème
  - Conflit sur la variable ***compteur*** *(le nombre d'éléments dans le tampon)*
- Illustration
  - Un consommateur teste ***compteur*** et trouve 0
  - L'ordonnanceur bascule sur un producteur *(avant que le consommateur n'exécute la fonction **sleep()**)*
  - Le producteur incrémente ***compteur*** (compteur = 1), et constate que le tampon était vide
  - Le producteur lance un wakeup() ''perdu'' car le consommateur n'était pas encore endormi !
- **Solution ?**
  - ''Mémoriser'' le wakeup() ! ⟹ les sémaphores

Le problème du producteur-consommateur

- Le producteur et le consommateur ne peuvent pas accéder en même temps au tampon (une ressource critique)
- Donc, besoin d'un sémaphore binaire

- Résoudre le problème du producteur-consommateur avec les sémaphores

- Gérer l'accès au tampon en utilisant un sémaphore binaire ou une exclusion mutuelle

Le problème du producteur-consommateur

- Solution avec 3 sémaphores :
  - **full**
    - Un sem. à compteur initialisé à 0 avec une liste vide
    - Contrôle le nombre d'éléments insérés dans le tampon
    - La liste contient les processus consommateurs bloqués
  - **empty**
    - Un sem. à compteur initialisé à *n* avec une liste vide
    - Contrôle le nombre de places vides dans le tampon
    - La liste contient les processus producteurs bloqués
  - **exmu**
    - Un sémaphore binaire initialisé à 1
    - Contrôle l'entrée et la sortie en Section Critique

Le problème du producteur-consommateur

init(full, 0), init(empty, n)

Producteur :

```
Tque en_vie
  produire (R)
  P(empty)

  insérer(R,tampon)

  V(full)
ftque
```

Consommateur :

```
Tque en_vie
  P(full)

  R=extraire(tampon)

  V(empty)
  consommer(R)
ftque
```

Le problème du producteur-consommateur

init(full, 0), init(empty, n), init(exmu,1)

Producteur :

```
Tque en_vie
  produire (R)
  P(empty)
  P(exmu)
  insérer(R,tampon)
  V(exmu)
  V(full)
ftque
```

Consommateur :

```
Tque en_vie
  P(full)
  P(exmu)
  R=extraire(tampon)
  V(exmu)
  V(empty)
  consommer(R)
ftque
```

Le problème du producteur-consommateur

```
#define BUFFER_SIZE 100
int buffer[BUFFER_SIZE];
/* Les sémaphores doivent être initialisées avant usage */
sema_t mutex, full, empty;
sema_init(&mutex, 1); sema_init(&full, 0); sema_init(&empty, BUFFER_SIZE);
```

```
void producteur() {
  int item;
  while(TRUE) {
    item = produire_objet();
    down(&empty);
    down(&mutex);
    deposer_objet(item, buffer);
    up(&mutex);
    up(&full);
  }
}
```

```
void consommateur() {
  int item;
  while(TRUE) {
    down(&full);
    down(&mutex);
    item = retirer_objet(buffer);
    up(&mutex);
    up(&empty);
    consommer_objet(item);
  }
}
```