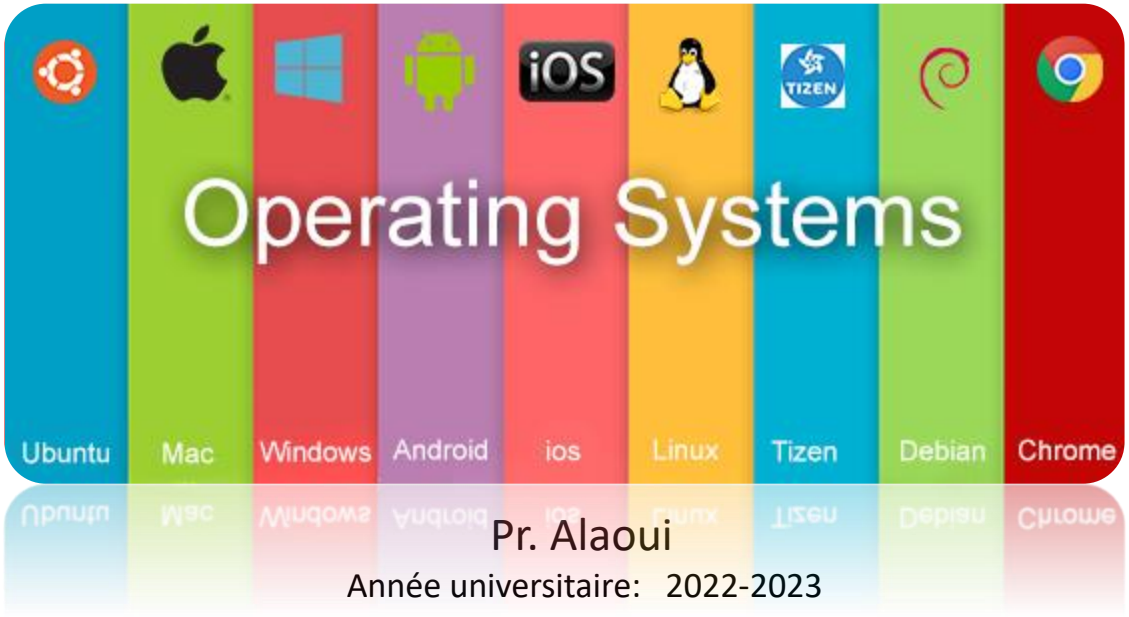
DUT GÉNIE INFORMATIQUE
2ÈME ANNÉE

M11

Les Systèmes d'exploitation
IV. Communication interprocessus & Synchronisation

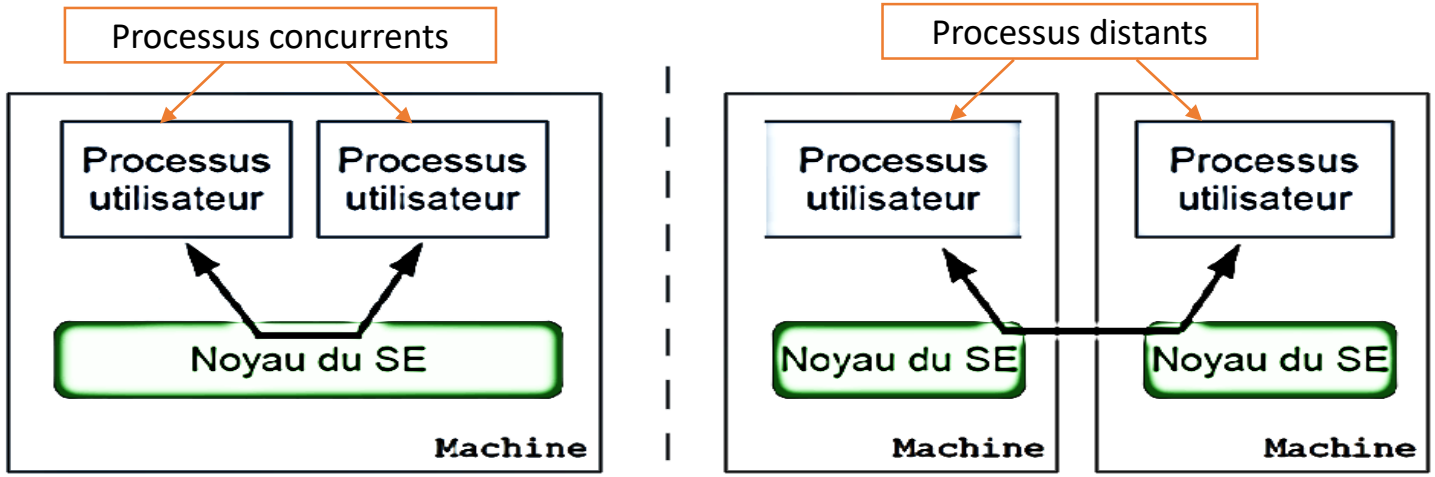


Pr. Alaoui

Année universitaire: 2022-2023

Notions

- Les processus, **concurrents** ou **distants**, sont amenés à **communiquer** et à **synchroniser** durant leur cycles de vie.
- Processus concurrents: sont en compétition pour le partage de **ressources**.
  - Coopérants: qui **partagent des données**, se trouvant en mémoire principale ou en mémoire secondaire, avec d'autres processus, et peuvent **s'affecter mutuellement** en cours d'exécution .
  - Indépendants: **ne partagent pas de données** avec d'autres processus et sont **ordonnancés indépendamment** les uns des autres.



Communications intra-système *Communications inter-système*

Notions: Ressource

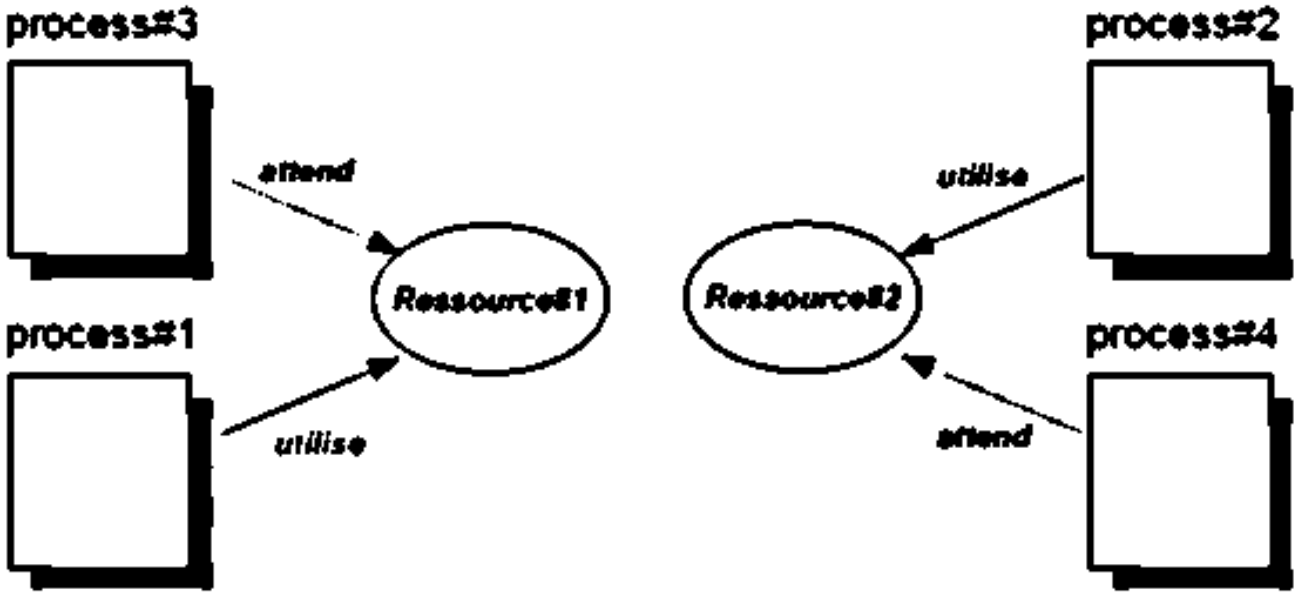
- Ressource: est toute entité dont a besoin un processus pour qu'il puisse évoluer
  - Matérielle: mémoire, UCT, périphériques
  - Logicielle: données, variables
- Ressource locale à un processus:
  - Ne peut être utilisée que par ce processus
  - Doit obligatoirement disparaitre à la destruction de ce processus puisqu'elle n'est plus utilisable.
- Ressource commune: n'est locale à aucun processus.
- Une ressource commune partageable avec n points d'accès (n >= 1): une ressource qui peut être attribuée, au même instant, à n processus au plus.
- Une ressource critique: partageable à un point d'accès (n=1 ou non partageable).
  - Exemple: L'UCT est une ressource à un seul point d'accès.
- Le mode d'accès à une ressource peut évoluer dynamiquement:
  - Un fichier est une ressource à **n points d'accès** quand il est **ouvert en lecture**, **critique** quand il est **ouvert en écriture**.
- Section Critique: Soit une ressource critique ***c***, **la section critique** d'un processus ***p***, pour la ressource ***c***, est une phase du processus ***p*** pendant laquelle il utilise ***c***, qui devient donc inaccessible aux autres processus.

Notions: Ressource

Les processus coopérants sont confrontés à deux grands problèmes : la famine et l'inter-blocage(deadlock).

- Famine: Monopolisation d'une ressource. Si un processus émet un flux constant de requêtes (de lecture par exemple) et si toutes ses requêtes sont satisfaites en premier, il pourrait arriver que les requêtes d'autres processus ne soient jamais satisfaites.

Attente indéfinie



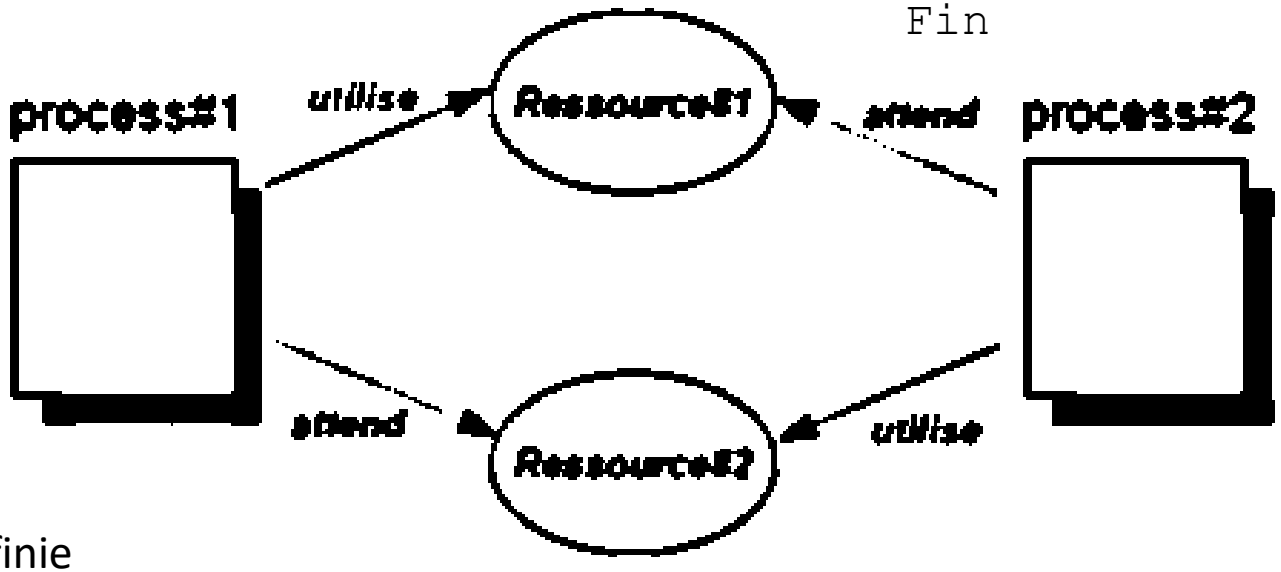
Notions: Ressource

- Inter-blocage: survient lorsque deux ou plusieurs processus demandent à obtenir des ressources en même temps, et que les ressources requises par les uns sont occupées par les autres et vice versa.

On considère deux processus P_1 et P_2 utilisant deux ressources critiques R_1 et R_2 comme suit :

```
Processus P1
Début
acquérir R1
acquérir R2
utiliser R1 et R2
Fin
```

```
Processus P2
Début
acquérir R2
acquérir R1
utiliser R2 et R1
Fin
```

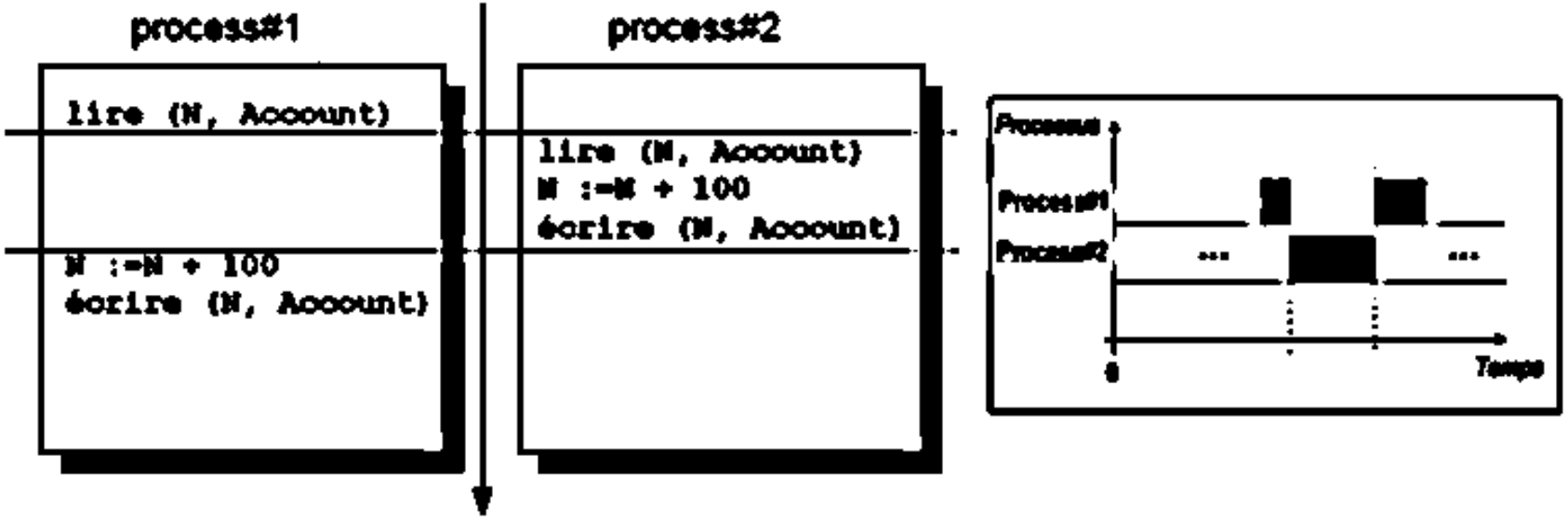


Attente infinie

Notions: Ressource

- Incohérence de données : problème de la synchronisation relative à l'exécution des processus

- **Exemple:** incrémenter un compte client de 100 depuis, deux opérations bancaires simultanées.
- **N:** solde initial, **Account:** numéro du compte client.



Une solution?

Notions: Exclusion mutuelle

- Soit deux processus ***p*** et ***q*** qui produisent des données devant être imprimées sur une imprimante unique. L'emploi de cette imprimante par ***p exclut son emploi par q tant que l'impression pour p n'est pas terminée***.
- **Un mécanisme d'exclusion mutuelle** sert à assurer l'atomicité des sections critiques relatives à une ressource critique.
- Autrement: s'assurer que les ressources non partageables ne soient attribuées qu'à un seul processus à la fois.
- Un processus désirant entrer dans une section critique doit être **mis en attente** si la ressource relative à la section critique n'est pas libre.
- Mais Comment peut-on attendre?
  - **Active :** procédure *entrer_Section_Critique* matérialisée *par* boucle dont la condition est un test qui porte sur des variables indiquant la présence ou non d'un processus en section critique
  - **Non active :** le processus passe dans l'état endormi et ne sera réveillé que lorsqu'il sera autorisé à entrer en section critique.

Notions: Exclusion mutuelle

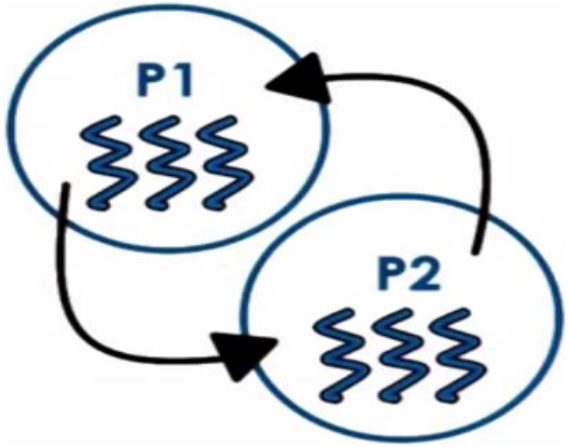
- Un mécanisme d'exclusion mutuelle doit satisfaire les conditions suivantes:
  1. **Exclusion mutuelle:** Si le processus Pi s'exécute dans sa section critique, alors aucun autre processus peut s'exécuter dans sa section critique.
  2. **Interblocage:** aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres d'y entrer.
  3. **Attente bornée:** aucun processus ne doit attendre indéfiniment avant d'entrer en section critique.
  4. Aucune hypothèse ne doit être faite sur les vitesses relatives des processus.

Accès concurrents aux ressources

- Comment gérer les accès concurrents aux ressources ?
- Mécanismes de:
  - Communication: Echange de données entre processus, tout en maintenant la protection ainsi que l'isolation entre processus communicants.
  - Synchronisation: La relation de dépendance logique entre processus qui cadence leur évolution et fixe l'ordre de leur exécution dans le temps (*i.e.* s'affecter mutuellement).

Mécanismes de Communications

- La communication interprocessus (IPC: interprocess communication) comparée à la communication entre ouvriers.



- **Les ouvriers** partagent un espace de travail: un dépôt d'outils et de pièces nécessaires pour la confection de voitures.
- **Les ouvriers** appellent les uns et les autres: expliciter les demandes et les réponses.
- Besoin de synchronisation: l'un commence sa tâche après la fin de celle de l'autre.

- **Les processus** partagent un espace de travail: une mémoire partagée.
- **Les processus** appellent les uns et les autres: passage de messages (message passing).
- Besoin de synchronisation: mécanismes de synchronisation.

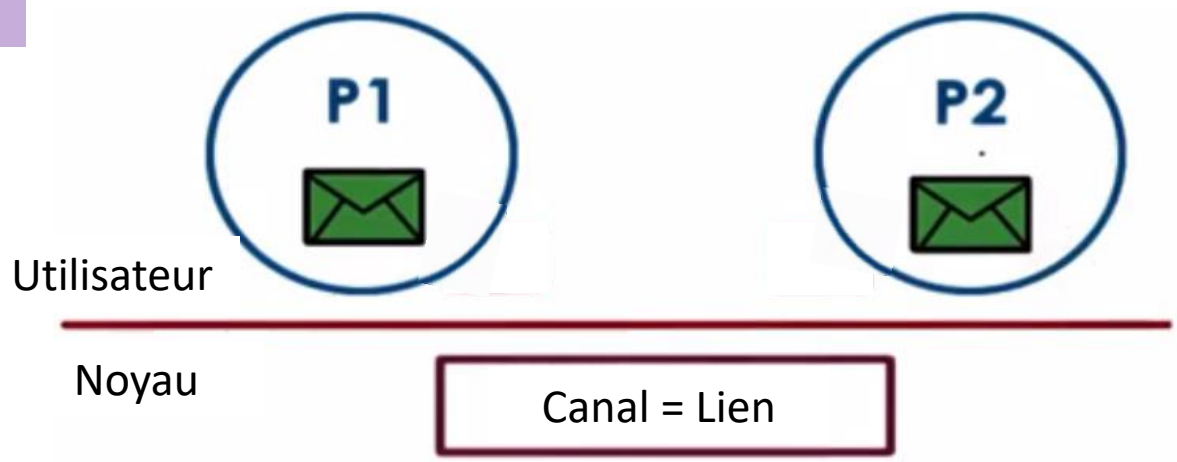
Mécanismes de Communications

- **IPC:** un ensemble de mécanismes que l'OS supporte pour permettre aux processus d'interagir entre eux (coordiner, communiquer).
- Les mécanismes IPC sont catégorisés comme suit:
  - Mécanismes à passage de messages (messages passing)
  - Mécanismes à mémoire partagée (shared memory)
- On verra que les IPC comprennent la notion de synchronisation.

Mécanismes de Communications

1)Message Passing

1.1)Principes

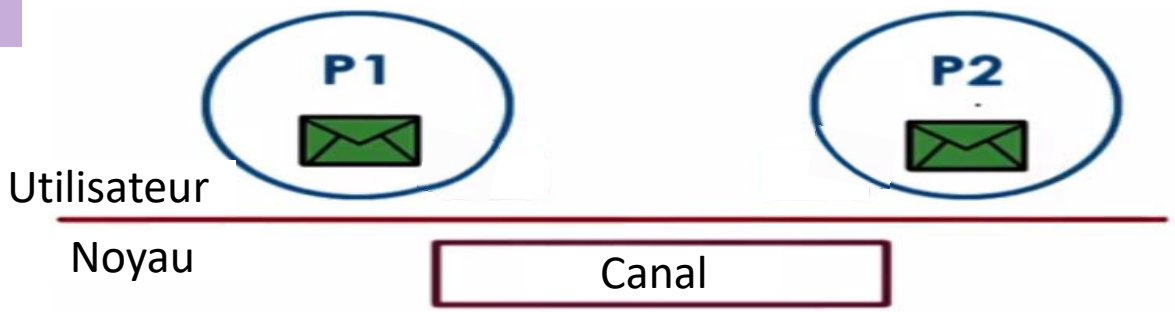


- Les processus créent des messages, puis les **envoient (écrire)** ou **les reçoivent (lire)**.
- Le noyau OS **établi et maintient le canal** qui sera utilisé pour transmettre les messages entre les processus et est requis pour **effectuer toutes les opérations IPC**.
- Le canal (ou le lien), qui peut être implémenté sous forme d'une file d'attente FIFO par exemple, est responsable de **transmettre le message** d'un processus à un autre.
- Ce lien peut être unidirectionnel ou bidirectionnel.

Mécanismes de Communications

1)Message Passing

1.1)Principes



- Coût des opérations:
  - L'envoi: appel système + copie du message depuis l'espace adresse du processus vers le canal.
  - La réception: appel système +copie du message depuis le canal vers l'espace adresse du processus de réception.
  - 1 communication= 4 copies de données + 4 passages user/Kernel.
- Inconvénients:
  - le coût généré (overhead) dû aux multiples copies de données IN/OUT le noyau ainsi que les passages multiples user/kernel.

✓Avantages:

- Le noyau du système d'exploitation prend en charge toutes les opérations, concernant la gestion des canaux.
- La synchronisation: le noyau s'assurera que les données ne sont pas écrasées ou corrompues d'une façon ou d'une autre, quand les processus tentent d'envoyer ou de recevoir en même temps.

Mécanismes de Communications

1)Message Passing

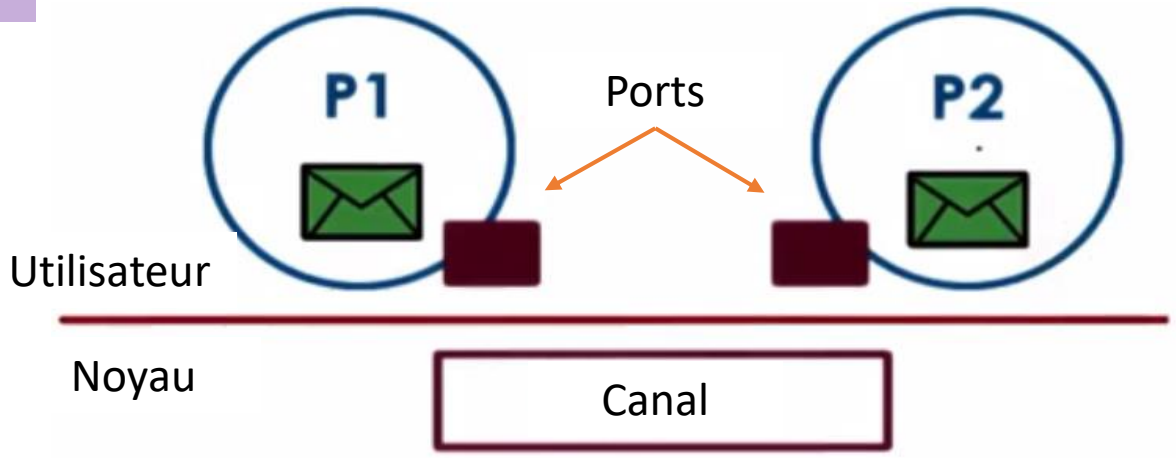
1.1)Principes

- Le Message passing peut être bloquant ou non-bloquant.
- Bloquant est pour dire synchrone
  - Envoie bloquant: l'expéditeur est bloqué jusqu'à la réception du message.
  - Réception bloquante: le récepteur est bloqué jusqu'à ce qu'un message soit disponible.
- Non-bloquant est pour dire asynchrone
  - Envoi non-bloquant: l'expéditeur envoie le message et continue.
  - Réception non-bloquante: le récepteur reçoit sans attendre.
- Si à la fois envoyer et recevoir bloquent, nous avons besoin d'un **rendez-vous**.

Mécanismes de Communications

1)Message Passing

1.1)Principes



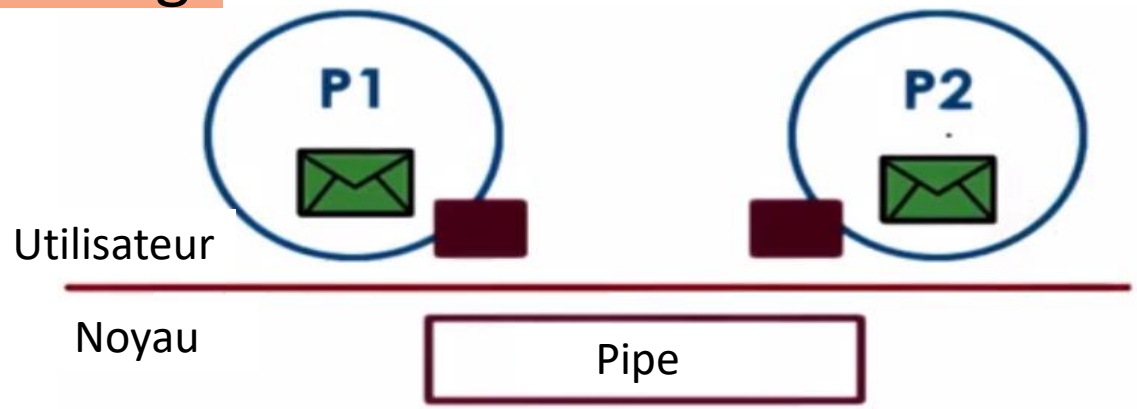
- Le message passing implémente la notion de port.
- Le port est une interface par laquelle un processus peut, entre autres, envoyer ou recevoir un message.

Mécanismes de Communications

1)Message Passing

1.2)Les pipes

a)Pipe ordinaire



- **Les pipes ordinaires** permettent à deux processus de communiquer de la manière standard producteur-consommateur: le producteur écrit à une extrémité du tuyau (WRITE_END) et le consommateur lit à l'autre extrémité (READ_END).
- Il n'y a pas de message en soi mais plutôt un flux d'octets poussés dans le pipe depuis un processus puis reçu dans un autre.
- Les pipes ordinaires sont unidirectionnels.
- Les pipes peuvent être accédées en utilisant les appels système read () et write ().
- Un pipe ordinaire n'est pas accessible depuis l'extérieur du processus qui l'a créé:
  - Le processus parent crée un pipe et l'utilise pour communiquer avec un processus enfant qu'il crée via fork ().
- Les pipes ordinaires peuvent être utilisés uniquement pour la communication entre les processus sur la même machine.
- Une fois que les processus ont fini de communiquer et sont terminés, le pipe ordinaire cesse d'exister.

Mécanismes de Communications

1)Message Passing

1.2)Les pipes

b)Pipe nommé

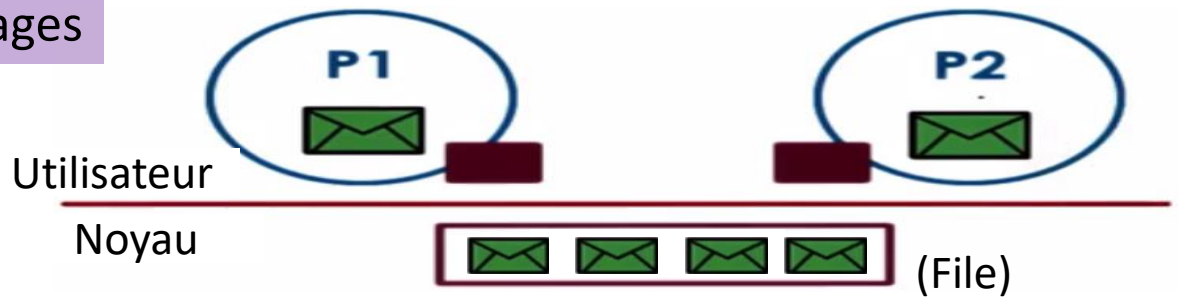
- La communication pour **les pipes nommés** peut être bidirectionnelle et aucune relation parent-enfant n'est requise.
- Une fois qu'un pipe nommé est établi, plusieurs processus peuvent l'utiliser pour la communication.
- Les pipes nommés continuent d'exister après la fin des processus communicants jusqu'à ce qu'ils soient explicitement supprimés du système de fichiers.
- **UNIX:**
  - Seule la transmission half-duplex est autorisée.
  - Si les données doivent passer dans les deux sens simultanément, deux FIFO sont généralement utilisés.
  - Les processus de communication doivent résider sur la même machine.
  - Si une communication intermachine est requise, les **sockets** doivent être utilisées.
- **WINDOWS:**
  - La communication en full-duplex est autorisée.
  - Les processus en communication peuvent résider sur la même machine ou des machines différentes.

Mécanismes de Communications

1)Message Passing

1.3)Les files de messages

a)Principe



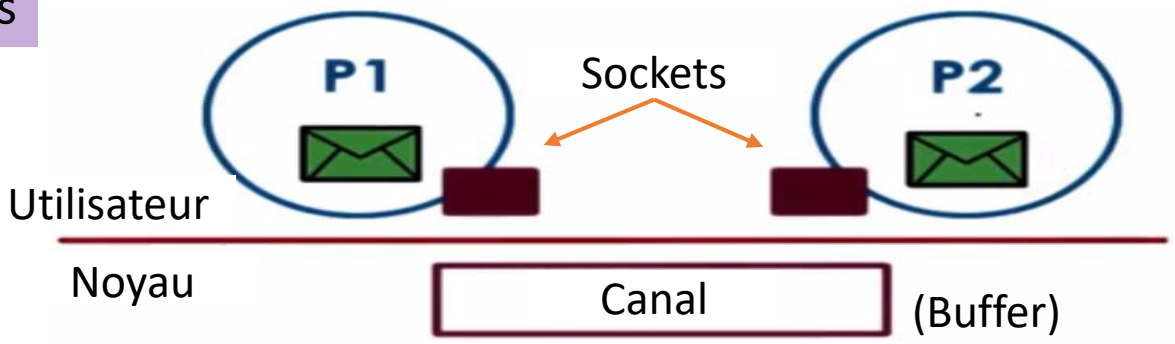
- Un processus émetteur doit envoyer un message correctement formaté au canal, puis le canal fournira un message correctement formaté au processus destinataire.
- Longueur maximale de zéro:
  - La file ne peut contenir aucun message.
  - L'expéditeur doit bloquer jusqu'à ce que le destinataire reçoive le message.
- Capacité bornée:
  - La file d'attente a une longueur finie n.
  - Si la file d'attente n'est pas pleine lorsqu'un nouveau message est envoyé, le message est placé dans la file d'attente et l'expéditeur peut continuer l'exécution sans attendre.
  - Si le lien est plein, l'expéditeur doit bloquer jusqu'à ce qu'un espace soit disponible dans la file d'attente.
- Capacité illimitée:
  - La longueur de la file est potentiellement infinie.
  - L'expéditeur ne bloque jamais.
- L'OS fournit des mécanismes aux niveaux des files d'attente de messages pour intégrer également notion de priorités des messages ou la planification des envoies des messages.

Mécanismes de Communications

1)Message Passing

1.3)Les files de messages

b)Les sockets

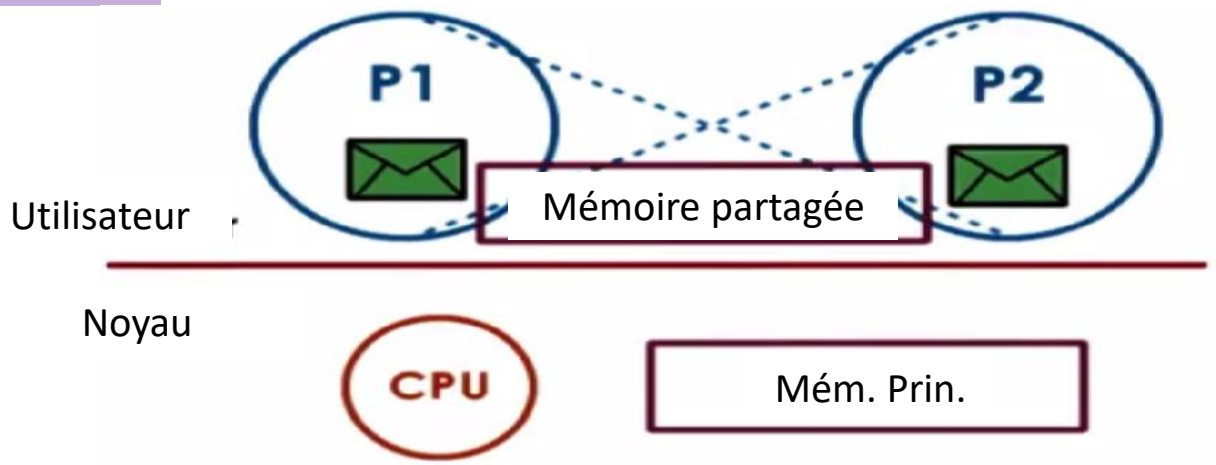


- Les sockets permettent aux processus d'envoyer des messages à l'intérieur et à l'extérieur du buffer de communication dans le noyau.
- Un socket est identifié par une adresse IP concaténée à un numéro de port.
- En général, les sockets utilisent une architecture client-serveur.
- L'appel socket():
  - Crée une mémoire buffer au niveau du noyau.
  - Associe tout le traitement au niveau du noyau nécessaire pour le transmission du message.
- Le socket peut être un socket TCP/IP, ce qui signifie que l'ensemble de la pile de protocoles TCP/IP est associé au mouvement des données dans le noyau.
- Le système d'exploitation est suffisamment intelligent pour comprendre que si deux processus sont sur la même machine, il n'a pas vraiment besoin d'exécuter la pile de protocoles complète pour envoyer les données sur le réseau, puis de le recevoir et le passer au le processus.

Mécanismes de Communications

2)Mémoire partagée

2.1)Principe

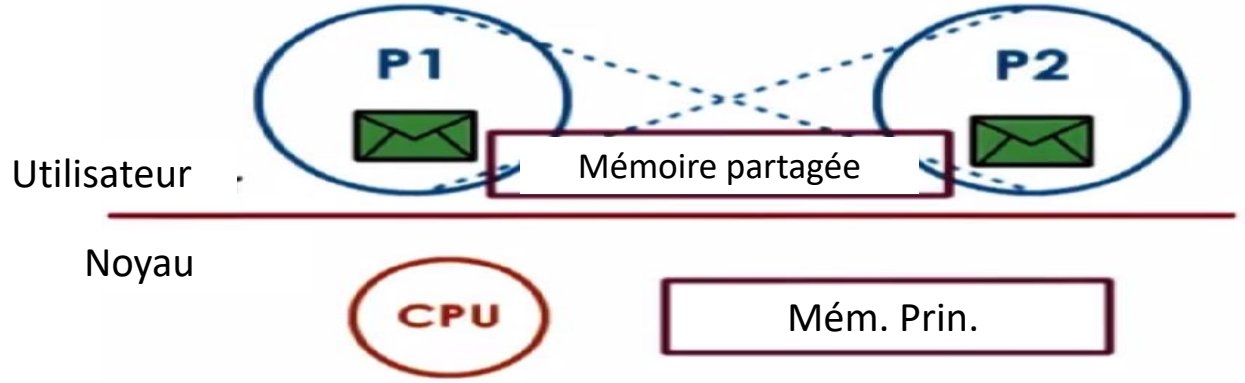


- Les processus **envoient (écrire)** ou **reçoivent (lire)** les messages dans une région partagée de la mémoire.
- Le noyau OS **établi** la mémoire partagée entre les processus.
- Un **même espace de la mémoire physique** peut être accessible par ces processus. C'est à dire qu'une adresse logique de P1 et une autre de P2 vont correspondre à la même adresse physique dans la mémoire principale.
- Normalement, le SE empêche un processus d'accéder à la mémoire des autres processus. Lorsque le mécanisme de mémoire partagée est utilisé, cela nécessite que les processus communicants **suppriment cette restriction**.

Mécanismes de Communications

2)Mémoire partagée

2.1)Principe

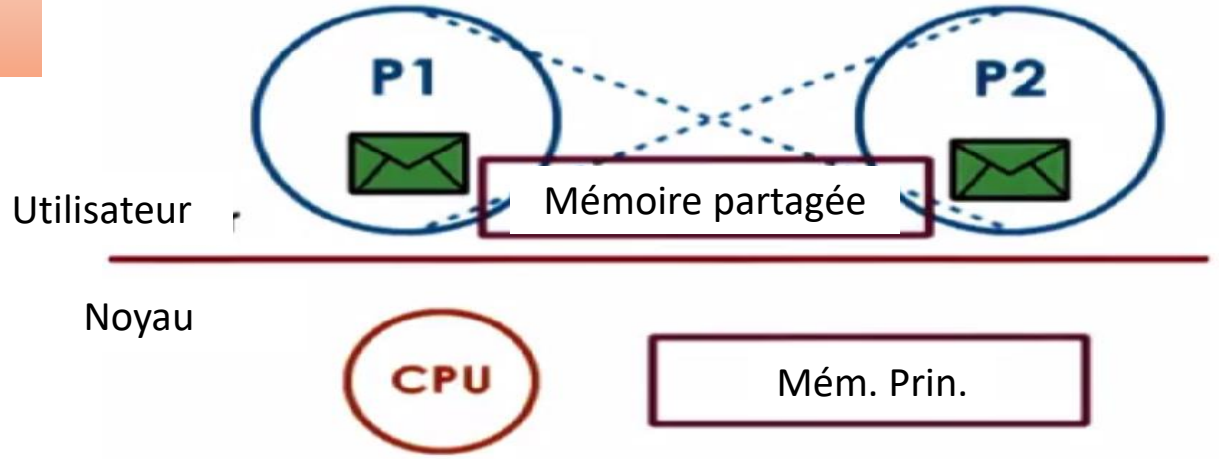


- Une mémoire partagée réside dans **l'espace d'adressage du processus** qui l'a créé. Les autres processus qui souhaitent communiquer à l'aide de cette région doivent la **joindre à leur espace d'adressage**.
- Les processus peuvent ensuite échanger des informations en **lisant et écrivant** des données dans la région partagée.
- Les processus sont également responsables de s'assurer qu'ils n'écrivent pas au même endroit **simultanément**. Cela est géré par les mécanismes de **synchronisation**.

Mécanismes de Communications

2)Mémoire partagée

2.1)Principe



✓Avantages:

- **Plus rapide** que le message passing car les appels système sont requis uniquement pour établir la région de la mémoire partagée.
- **Réduction du nombre de copies** de données. Un processus peut utiliser une donnée dans la mémoire partagée sans avoir besoin de la copier.

- Inconvénient:
  - **C'est au programmeur de gérer** les accès et l'organisation de la mémoire partagée. La difficulté majeur est de gérer la **synchronisation**: les processus doivent synchroniser explicitement leurs accès à la mémoire partagée.

Mécanismes de Communications

2)Mémoire partagée

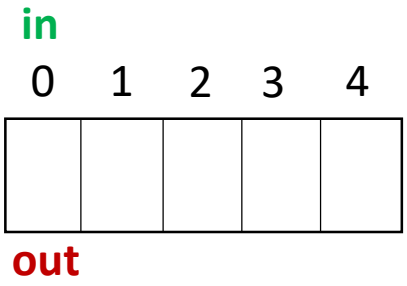
2.2) Problème du producteur/consommateur

- Le problème majeur de la mémoire partagée est la gestion de la synchronisation.
- **La synchronisation** permet de gérer les accès à la mémoire partagée grâce à plusieurs mécanismes.
- Un mécanisme simple est d'utiliser le concept du **problème du producteur/consommateur:**
  - Le processus producteur (**P**) ne peut que produire (**écrire**) des informations.
  - Le processus consommateur (**C**) ne peut que consommer (**lire**) ces informations.
- Déroulement**:**
  - Pour permettre l'exécution simultanée de P et C, un **buffer** est rempli par P et est vidé par C. Le buffer réside dans la région mémoire partagée par P et C.
  - P peut produire une information pendant que C consomme **une autre**.
  - P et C doivent être **synchronisés**, de sorte que C n'essaie pas de consommer une information qui **n'a pas encore été produite**.
  - **Buffer:** tableau circulaire avec deux pointeurs **in** (modifié par P et indique prochaine case vide) et **out** (modifié par C et indique première case pleine).

Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



1) Etat initial:
- buffer vide: **in=out**
- Si P ne produit rien, C ne peut rien consommer

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

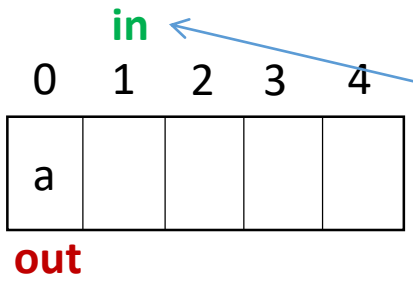
```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



2) P produit l'item a dans next_prod:
- next_prod est mis dans la case 0
- **in** est incrémenté par 1

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

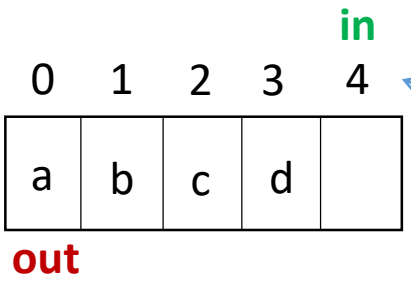
```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



3) P produit 3 autres items:
- A chaque itération, next_prod est mis dans la case **in**
- **in** est incrémentée et atteint case 4

4) P ne pourra par produire un 5ème item: après cela, il devra pointer sur la case 0 (= 5 mod 5) contenant un item pas encore lu par C (pointé par **out**).

```
//mémoire partagée
#define B_S 5 /* taille du buffer
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

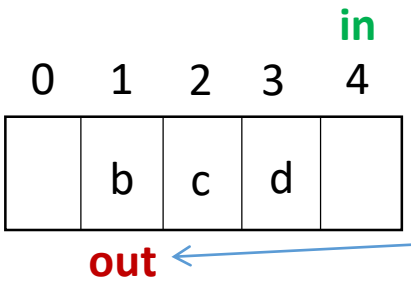
```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



5) C consomme un item:
- L'item à la case pointée par **out** (=0) est mis dans next_cons
- **out** est incrémentée

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

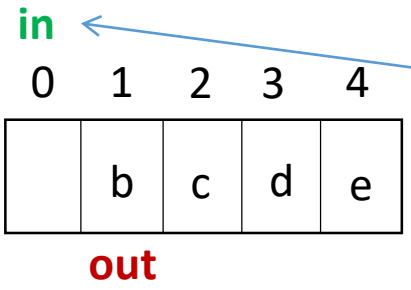
```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



4-bis) P pourra produire un 5ème item: le contenu de la case 0 est consommé, elle est libre d'accès (car **in** ≠ **out**)

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

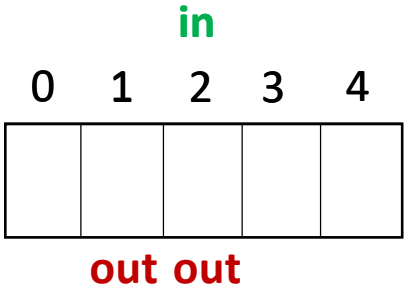
```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

Mécanismes de Communications

2)Mémoire partagée

2.2) Problème du producteur/consommateur



6) Supposant la configuration à gauche (**out** précède **in** de 1) et C consomme le contenu de la case 1:
- out pointe alors sur case 2 (= **in**).
-C ne pourra plus consommer d'item (car **in=out)**

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
```

```
//Producteur P
item next_prod;
while (true) {
/* Tester si production est possible*/
while(((in + 1) % B_S) == out);
/* si buffer plein, ne rien faire */
buffer[in] = next_prod; //produire
in = (in + 1) % B_S; }
```

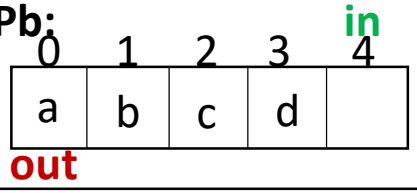
```
//Consommateur C
item next_cons;
while (true) {
while (in == out) ;/* si buffer
vide, ne rien faire */
next_cons = buffer[out];//consommer
 out = (out + 1) % B_S;
}
```

Mécanismes de Communications

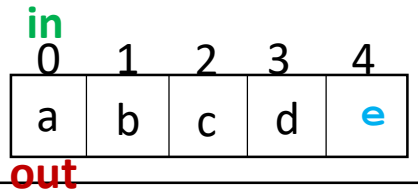
2)Mémoire partagée

2.2) Problème du producteur/consommateur

Pb:



Sol:



Problème de l'algo: Si B_S = 5, P ne peut déposer que **4 items à la fois** (étape 4).

Solution: Introduire une variable globale (**counter**):

- initialisée à 0
- utilisée dans les boucles while.
- incrémentée à chaque production
- décrémentée à chaque consommation.

```
//mémoire partagée
#define B_S 5 /* taille du buffer */
typedef struct {
. . .
} item;
item buffer[B_S];
int in = 0;
int out = 0;
int counter=0;
```

```
//Producteur P
item next_prod;
while (true) {
/* produire un item dans  next_prod */
While( counter == B_S ); //buffer
plein, ne rien faire
buffer[in] = next_prod;
in = (in + 1) % B_S;
counter++; }
```

```
//Consommateur C
item next_cons;
while (true) {
while ( counter == 0 );//buffer
vide, ne rien faire
next_cons = buffer[out];
out = (out + 1) % B_S;
counter--;
/* consommer un item dans
next_cons */ }
```

Mécanismes de synchronisation

1) Introduction

- **Synchronisation(Remix):** elle se présente comme un ensemble de mécanismes qui permettent aux processus d'accéder à leur section critique en garantissant l'exclusion mutuelle.
- Les mécanismes de synchronisation sont catégorisés comme suit:
  - Mécanismes logiciels
  - Mécanismes matériels
  - Combinaison des deux

On introduit la synchronisation à travers la solution proposée (i.e. variable `counter`) au problème rencontré dans l'algorithme du producteur/consommateur.

Mécanismes de synchronisation

1) Introduction

1.1) Problème de l'algo du producteur/consommateur

- L'incrémentation et décrémentation de counter est exécutée sous forme de 3 instructions en assembleur:

counter++ :
- `reg1=counter`
- `reg1 = reg1+1`
- `counter = reg1`

counter-- :
- `reg2 = counter`
- `reg2 = reg2-1`
- `counter = reg2`

- On se positionne dans le cas où les processus s'exécutent en parallèle et que P exécute counter++ et C counter--. Considérons que counter = 5 et que les instructions s'exécutent dans cet ordre:
  - Inst0: P exécute reg1 = counter {reg1 = 5}
  - Inst1: P exécute reg1 = reg1 + 1 {reg1 = 6}
  - Inst2: C exécute reg2 = counter {reg2 = 5}
  - Inst3: C exécute reg2 = reg2 – 1 {reg2 = 4}
  - Inst4: P exécute counter = reg1 {counter = 6 }
  - Inst5: C exécute counter = reg2 {counter = 4} **Cette valeur doit être = 5**

- **Problème:** les processus P et C **accèdent en même temps** à une variable partagée counter
- **Solution:** introduire une **section critique** pour l'accès à counter

Mécanismes de synchronisation

1) Introduction

1.2) Section critique

- La structure générale d'un programme utilisant une section critique :

```
do{
        Entrée section critique

          // code section critique

        Sortie section critique

          // code non critique

}while{true}
```

- **Rappel:** Lorsqu'on introduit une section critique, il faut s'assurer de satisfaire les conditions d'exclusion mutuelle:
  1. Exclusion mutuelle
  2. Pas d'Interblocage
  3. Attente bornée
  4. Pas d'hypothèse sur les vitesses relatives des processus.

Mécanismes de synchronisation

2) Mécanisme logiciel

2.1) Introduction

- **Synchronisation par mécanisme logiciel:** l'accès à la section critique est contrôlé par un algorithme uniquement et n'a pas besoin de circuit spécial.
- **La Solution de Peterson** en est un exemple:
  - Elle est adaptée au cas de deux processus
  - Les processus partagent deux variables :
    - Int **turn**: indique l'indice du processus (1 ou 2) qui entre dans la section critique
    - Boolean **flag[2]**: flag[k]==True indique que le processus « k+1 » souhaite entrer dans la section critique
  - Remarque: *i* est l'indice du processus courant, *j* est l'indice de l'autre processus

Mécanismes de synchronisation

2) Mécanisme logiciel

2.2) Solution de Peterson

```
do{// le processus
    flag[i-1] = true; // inst 1
    turn = j; // inst 2
    while(flag[j-1] && turn = = j);  // inst 3
// code section critique
    flag[i-1] = false; // inst 4
// code non critique
}while{true}
```

- 2 processus P1 et P2 s'exécutent en même temps
- Initialement: `turn=0 | flag=[F,F]`
- P1 puis P2 →inst 1 `turn=0 | flag=[T,T]`
- P1 → inst 2: `turn=2 | flag = [T,T]`
- P1 → inst 3: `While(T), P1 boucle et ne peut pas accéder à sa SC`
- P2 → inst 2: `turn=1 | flag = [T,T]`
- P1 → inst 3: `While(F), P1 entre en SC`
- P2 → inst 3: `While(T), P2 boucle et ne peut pas accéder à sa SC`
- P1 termine SC → inst 4: `turn=1 | flag=[F,T]`
- P2 → inst 3: `While(F), P2 entre en SC`
- P1 entre en section non critique, puis termine
- P2 termine SC → inst 4: `turn=1 | flag=[F,F]`
- P2 entre en section non critique, puis termine

Les conditions de l'exclusion mutuelle sont satisfaites:

1) Pas d'accès simultané à la SC
2) Pas d'interblocage
3) Les processus ont accédé à leur SC.
4) Pas de supposition sur la vitesse des processus

Mécanismes de synchronisation

2) Mécanisme logiciel

2.2) Solution de Peterson

- La Solution de Peterson peut **poser un problème** lorsqu'on dispose d'un processeur superscalaire (comporte plusieurs unités de calcul).
- Ce processeur peut exécuter plusieurs instructions simultanément parmi une suite d'instructions. Soit les instructions suivantes traitées par processeur qui exécute 2 instructions à la fois :

```
1. Mov eax,0
2. Mov ebx,1
3. Mov edx,2
4. Inc edx
5. Mov ecx,3
```

```
1. Mov eax,0
2. Mov ebx,1
3. Mov edx,2
5. Mov ecx,3
4. Inc edx
```

- Le processeur choisit à chaque fois 2 instructions qui n'agissent pas sur les mêmes registres.
- Il peut alors exécuter : (1,2); (3,5); (4)
- Cela produit un changement dans l'ordre des instructions (**memory reordering**).

Mécanismes de synchronisation

2) Mécanisme logiciel

2.2) Solution de Peterson

- Si la solution de Peterson rencontre un changement d'ordre des instructions. On peut imaginer le scénario suivant:

```
do{// le processus
   flag[i-1] = true; // inst 1
   turn = j; // inst 2
   while(flag[j-1] && turn = = j)
   ; // inst 3
// code section critique
   flag[i-1] = false; // inst 4
// code non critique
}while{true}
```

- Initialement: `turn=0 | flag = [F,F]`
- P1 →inst 2 puis P1 → inst 3:

`turn=2 | flag=[F,F] et P1 entre en SC`

- P2 → inst 2 puis P2 → inst 3:

`turn=1 | flag=[F,F] et P2 entre en SC`

- P1 → inst 1 `turn=1 | flag = [T,F]`
- P2 → inst 1: `turn=1 | flag = [T,T]`
- …..

Une condition de l'exclusion mutuelle n'est pas satisfaite:

1) accès simultané à la SC par les deux processus

Solution: Mécanisme matériel

Mécanismes de synchronisation

3) Mécanisme matériel

- **Synchronisation par mécanisme matériel:** l'accès à la section critique est contrôlé par un circuit spécial atomique qui ne peut pas être interrompu pendant son exécution.
- Le principe commun des mécanismes matériels est d'utiliser un **verrou** (lock) pour bloquer/débloquer l'accès à la section critique.

```
do{
        Acquire Lock (verrouillage)

            // code section critique

        Release Lock (déverrouillage)

            // code non critique

}while{true}
```

- Il existe plusieurs mécanismes matériels, parmi eux:
  - La solution test_and_set()
  - Mutex (API)

- **Note:** les algorithmes présentés ci-après des solutions matérielles décrivent le comportement du circuit atomique utilisé. Ce circuit ne peut être utilisé par deux ou plusieurs processus à la fois.

Mécanismes de synchronisation

3) Mécanisme matériel

3.1) Solution test_and_set()

```
boolean test_and_set(boolean
*oldLock) // circuit atomique
{
  boolean newLock=*oldLock;
  *oldLock=true;
  return newLock;
}
```

- lock est une variable globale et son passage se fait par référence (&lock).

```
boolean lock = false; //initialisation
```

```
do{// le processus
  while(test_and_set(&lock)) ;//inst 1

// code section critique
    lock = false; // inst 2

// code non critique
}while{true}
```

Cas de deux processus:

- P1 →inst 1 `lock=T | while(F)` P1 dans SC.
- P2 →inst 1 `lock=T | while(T)` P2 attend.
- P1 →inst 2 `lock=F` P1 dans SNC
- P2 →inst 1 `lock=T | while(F)` P2 dans SC
- P2 →inst 2 `lock=F` P2 dans SNC

Les conditions de l'exclusion mutuelle sont satisfaites:

1) Pas d'accès simultané à la SC
2) Pas d'interblocage
3) Les processus ont accédé à leur SC.
4) Pas de supposition sur la vitesse des processus

Mécanismes de synchronisation

3) Mécanisme matériel

3.1) Solution test_and_set()

```
boolean test_and_set(boolean *oldLock)
// circuit atomique
{
  boolean newLock=*oldLock;
  *oldLock=true;
  return newLock;
}
```

```
boolean lock = false; //initialisation
```

```
do{// le processus
  while(test_and_set(&lock)) ;//inst 1
// code section critique
    lock = false; // inst 2
// code non critique
}while{true}
```

Cas de trois processus:

- P1 →inst 1 `lock=T | while(F)` P1 dans SC.
- P2 →inst 1 `lock=T | while(T)` P2 attend.
- P1 →inst 2 `lock=F` P1 dans SNC.
- P2 →inst 1 `lock=T | while(F)` P2 dans SC
- P3→inst 1 `lock=T | while(T)` P3 attend.
- P1→inst 1 `lock=T | while(T)` P1 attend.
- P2 →inst 2 `lock=F` P2 dans SNC.
- P1→inst 1 `lock=T | while(F)` P1 dans SC.
- ……

Une condition de l'exclusion mutuelle n'est pas satisfaite:
3) P3 n'a pas accédé à sa SC, alors que P1 y a accédé deux fois.

Solution:
Réécrire le code en ajoutant d'autres variables pour que les processus puissent accéder autant de fois les uns et les autres

Mécanismes de synchronisation

3) Mécanisme matériel

3.2) Mutex

- Pour simplifier l'utilisation des solutions matérielles aux programmeurs, les concepteurs des SE ont proposé des outils logiciels (API) permettant l'appel de ses solutions.
- Le verrou mutex en fait partie. Il permet d'appeler un mécanisme de synchronisation matériel tel que test_and_set():
  - La fonction acquire() permet d'établir le verrou sur la section critique
  - La fonction release() libère le verrou.

```
do{
        acquire ()

        // code section critique

        release ()

        // code non critique

}while(true);
```

```
acquire()
{
while(test_and_set(&lock));
// busy wait (attente active)
}
```

```
release()
{
  lock=true;
}
```

Mécanismes de synchronisation

3) Mécanisme matériel

3.3) Problème du busy wait

- Lorsqu'un processus entre dans la section critique, les autres processus sont occupés à attendre (**busy wait**) sa libération:
  - Dans le cas de la solution `test_and_set()`, la boucle `while(test_and_set(&lock));` occupe le processeur tous le temps d'attente de la libération de la section critique
- Inconvénient: gaspille des cycles processeur.

✓Avantage:

- Pas de changement de contexte.
- Lorsque la section critique est de courte durée, cela peut être plus avantageux que de céder le processeur à un autre processus et gaspiller du temps en changement de contexte.

- Le problème du busy wait est résolu par **les sémaphores**

Mécanismes de synchronisation

4) Les sémaphores

4.1) Définition

- Dans le domaine ferroviaire, un sémaphore est un signal permettant de déterminer si l'accès à la voie est libre (sémaphore ouvert) ou non (sémaphore fermé).
- Pour résoudre le problème du busy wait, on utilise un **sémaphore** représenté par une variable entière qui garantit l'exclusion mutuelle, accompagnée d'une liste de processus en attente pour accéder à la SC.
- En dehors de l'initialisation, le sémaphore n'est accessible que par deux opérations `wait()` et `signal()` (aussi notées dans la littérature `P()` et `V()`, `down()` et `up()`).

Fermé

Ouvert

Mécanismes de synchronisation

4) Les sémaphores

4.2) Solution au busy wait

```
while(true){//Processus
  wait(&S); //inst 1
 //section critique
  signal(&S); //inst 2
//section non critique
}
```

```
typedef struct{
int value;
struct process *list;
} semaphore;
```

```
wait(semaphore *S){
 S.value--;
 if (S.value < 0){
 add_Proc(S.list);
 block();}}
```

```
signal(semaphore *S){
 S.value++;
 if (S.value <= 0){
Proc=remove_Proc(S.list);
 wakeup(Proc);}}
```

`Semaphore S | S.value=1 | S.list=Ø //initialisation`

- P1 →inst 1 `S.value=0` P1 dans SC.
- P3 →inst 1 `S.value=-1 | S.list=P3` P3 est block.
- P2 →inst 1 `S.value=-2 | S.list=P3,P2` P2 est block.
- P1 →inst 2 `S.value=-1 | S.list=P2` 1er proc (P3) de S.list wakeup et `RQ_UCT=P3` . P1 accède à SNC et termine.
- P3 accède à l'UCT `RQ_UCT= Ø` et entre en SC.
- P3 →inst 2 `S.value=0 | S.list=Ø` P2 wakeup `RQ_UCT=P2` et P3 accède à SNC et termine.
- P2 accède à l'UCT `RQ_UCT= Ø` et entre en SC.
- P2 →inst 2 `S.value=1` P2 accède à SNC et termine.

Mécanismes de synchronisation

4) Les sémaphores

4.2) Solution au busy wait

- **Le busy wait est annulé:**
  - Lorsque un processus est placé dans S.list, cela implique un changement de contexte.
  - L'appel de la fonction block() empêche le processus de demander l'accès à l'UCT.
- **Les conditions de l'exclusion mutuelle sont satisfaites:**

  1. Pas d'accès simultané à la SC
  2. Pas d'interblocage
  3. Les processus ont accédé à leur SC: S.list permet d'ordonner l'accès à la file d'attente prêt de l'UCT (pas comme dans test_and_set() cas 3 processus).
  4. Pas de supposition sur la vitesse des processus
- Le sémaphore est **une solution logicielle** pour le problème du busy wait, il y a donc un risque de **changement d'ordre des instructions.**

Mécanismes de synchronisation

5) Solution combinée: Sémaphore & mutex

```
while(true){//Processus
 acquire();
  wait(&S); //inst 1
 release();
 //section critique
  signal(&S); //inst 2
//section non critique
}
```

```
typedef struct{
int value;
struct process *list;
} semaphore;
```

```
wait(semaphore *S){
 S.value--;
 if (S.value < 0){
 add_Proc(S.list);
 block();}}
```

```
signal(semaphore *S){
 S.value++;
 if (S.value <= 0){
Proc=remove_Proc(S.list);
 wakeup(Proc);}}
```

```
Semaphore S | S.value=1 | S.list=Ø //initialisation
```

- Pour résoudre le problème du memory reordering, on **combine le sémaphore (solution logicielle) avec une solution matérielle.**
- On inclut donc un mutex autour de l'appel du wait(&S)
- Rappelons que nous avons introduit le sémaphore pour **résoudre le problème du busy wait** provoqué par mutex et les solutions matérielles en général.
- En réalité, établir mutex autour de wait(&S) permet de **réduire le degré du busy wait** (la fonction n'est composée que de 3 instructions), comparé à celui engendré par mutex autour de la section critique (généralement beaucoup plus longue).

Mécanismes de synchronisation

6) Problème d'interblocage

- **Interblocage:** deux ou plusieurs processus attendent indéfiniment l'arrivée d'un événement qui ne peut être engendré que par un des processus en attente.
- **Exemple:** Soit P0 et P1 deux processus partageant deux sections critiques, l'une contrôlée par le sémaphore S et l'autre par le sémaphore Q:

  - P0 exécute wait(S), puis P1 éxécute wait(Q).
  - P0 exécute wait(Q) mais il est block(), puis P1 exécute wait(S) et il est block().
  - Aucun processus ne peut avancer dans son exécution, car il a besoin que l'autre processus le débloque par signal(). c'est **l'interblocage**
- Il est donc nécessaire de bien programmer les mécanismes de synchronisation pour éviter ce problème.