# Module SEL

# Synchronisation

- Exclusion mutuelle
- Sémaphores
- Schémas classiques de synchronisation
- Bestiaire des outils de synchronisation

## Exclusion mutuelle : introduction

- Coopération entre processus ⇒ communication via la mémoire, qui devient une ressource partagée
- Les opérations élémentaires (ex: affectation) peuvent conduire à des incohérences
- ⇒ L'accès à des ressources partagées entre plusieurs processus nécessite des synchronisations explicites pour être correct
  - Mise en évidence de ces problèmes d'incohérence
  - Définition de la notion de section critique

## Exemple 1 : ressource matérielle partagée

- Soient deux programmes P1 et P2 désirant imprimer sur la même imprimante (mode caractère)
- La fonction *ecrire* est supposée être une opération atomique d'impression d'un caractère

**Processus** P1 :
**char** t1[4]='abcd';
**for** (i=0; i<4;i++) {
        ecrire(t1[i]);
}

**Processus** P2 :
**char** t2[4]='ABCD';
**for** (i=0; i<4;i++) {
        ecrire(t2[i]);
}

Module SEL 3

## Exemple 1 : ressource matérielle partagée

- Comportement en l'absence de synchronisation explicite
  - On peut voir imprimer n'importe quelle séquence de caractère respectant l'ordre d'exécution de chaque programme ('abcdABCD''aAbBcCdD', 'abcABCDd', etc.)
  - On voudrait voir imprimée la séquence 'abcdABCD' ou 'ABCDabcd'}

Module SEL 4

## Exemple 2 : ressource logicielle (variable) partagée

- Soit deux exécutions concurrentes P1 et P2 du même programme de réservation de places dans un avion
- Le code traduit le fait que *libres* n'est jamais négatif

```
int libres = 1;
if (libres > 0)  {
      libres--;
      réserverPlace(client);
}
```

## Exemple 2 : ressource logicielle (variable) partagée

- Considérons l'entrelacement suivant
  - (P1) libres==1 (décide de réserver)
  - (P2) libres==1 (décide de réserver)
  - (P2) libres--; ⇒ libres=0;
  - (P1) libres--; ⇒ libres=-1;
- On a un surbooking par erreur (et bientôt un client mécontent)

## Exemple 2 : ressource logicielle (variable) partagée

- Dans ce cas, on souhaiterait rendre atomique la section  de code manipulant la variable *libres*
- Atomique : même résultat que si la section était exécutée séquentiellement
- On ne peut pas savoir a priori le résultat de l'exécution  parallèle de P1 et P2. Le parallélisme introduit de l'indéterminisme

## Section critique et exclusion mutuelle

- Exécution acceptable
  - Une exécution parallèle est acceptable si elle produit le même résultat qu'une exécution séquentielle possible
- Section critique
  - La partie du programme qui peut rendre inacceptable le résultat de l'exécution parallèle est une section critique.
  - Une section critique est une portion de code que l'on veut rendre indivisible (atomique)

## Section critique et exclusion mutuelle

- Atomique (définition)
  - Résultat équivalent à une exécution non interrompue.
- Indivisible (définition)
  - On ne voit pas de valeur intermédiaire de la variable
  - Résultat équivalent à une exécution non interrompue (équivalent à atomique)
- Non interruptible (définition)
  - Exécution ne sera jamais interrompue

## Section critique et exclusion mutuelle

- Remarques
  - Non interruptible $\Rightarrow$ atomique, mais pas le contraire
  - Réalisations de sections critiques ne rendent pas nécessairement les sections critiques non interruptibles

## Section critique et exclusion mutuelle

**Toute ressource partagée (matérielle, variable) doit être manipulée en section critique**

## Section critique et exclusion mutuelle

- Ressource critique (définition)
  - Une ressource critique est une ressource dont la manipulation doit être effectuée dans une section critique
- Exclusion mutuelle (définition)
  - Quand deux processus manipulent une ressource dans une section critique, on dit que la ressource est manipulée en exclusion mutuelle (i.e. les deux processus n'y touchent pas en même temps)

## Section critique et exclusion mutuelle

- Structure du code assurant l'atomicité

  <entrée en section critique>;
  <section critique>;
  <sortie de section critique>

## Section critique et exclusion mutuelle

- Remarques
  - Exclusion mutuelle et section critique ne sont pas absolues mais relatives à une ressource précise.
  - Ne s'applique qu'aux processus utilisant cette ressource
  - Définir la durée d'une section critique n'est pas une tâche aisée
    - Trop courte peut être incorrect
    - Trop longue peut être peu performant

## Propriétés à assurer

- Propriétés obligatoires
  - Sûreté : deux processus ne doivent pas être en même temps dans leur section critique
  - Vivacité (absence de blocage inutile) si aucun processus n'est en section critique, aucun processus ne doit être bloqué par le mécanisme de contrôle de la section critique
- Propriété individuelle
  - Absence de privation : aucun processus ne doit être obligé d'attendre indéfiniment avant de pouvoir entrer en section critique

## Solution triviale (mais fausse ...)

```
char occup = 0; // 1 si un processus est en SC
while (occup == 1) { ; } // rien
occup = 1;
<section critique>;
occup = 0;
```

- La solution triviale ne fonctionne pas (si deux processus essaient d'entrer simultanément en section critique, les deux voient *occup* à *0* et entrent en section critique)
- Le problème vient de l'absence d'atomicité de la séquence test-modification de *occup*

## Solution triviale (mais fausse ...)

- Problème supplémentaire
  - Inefficace (boucle qui monopolise le processeur pendant l'attente)
- On fait comment alors ?
  - On essaie d'améliorer la solution triviale ? Non !
    - Ou alors, on sait ce qu'on fait, …
    - Algorithmes spécifiques, instructions read-modify-write, cf module NOY !
  - On utilise ce que nous offre notre système d'exploitation ? Oui !

## Exemple d'interface: mutex de la librairie pthreads

- pthread_mutex_init(&m, ...)
  Création d'un mutex
- pthread_mutex_destroy(m)
  Destruction d'un mutex
- pthread_mutex_lock(m)
  Acquisition d'un mutex (ou attente)
- pthread_mutex_unlock(m)
  Relâchement d'un mutex
- pthread_mutex_trylock()
  Tentative d'acquisition d'un mutex (sans attente)

## Mutex : mode d'emploi

- Une ressource partagée ⇒ un mutex
  - Sauf quand deux variables sont toujours manipulées en même temps
- pthread_mutex_trylock()
  - Attention aux problèmes d'atomicité !

```
mutex_t m;
If (pthread_mutex_trylock(&m) != 0) {
    // On a le mutex, on peut manipuler la variable
    ….
} else {
    // Que peut on dire et faire ?
}
```

## Sémaphore

- Introduit par Edsger Dijkstra en 1965
- Sémaphore dit à compteur (il existe d'autres sortes de sémaphores)
- Un outil parmi d'autres (conditions, signaux, moniteurs, etc.)
- Accent mis sur la définition et l'interface d'accès aux sémaphores, pas leur réalisation

## Sémaphore

- Un sémaphore *s* est un composé :
  - D'un compteur *s.e*
    - Valeur initiale du compteur ≥ 0
  - Primitives d'accès atomiques
    - P (ou Wait)
    - V (ou Signal)
    - P et V viennent de termes Hollandais (nationalité de Dijkstra, P =Proberen = test, V=Verhogen = increment)
    - P = Puis-je ?, V = Vas-y !

## Sémaphore

Description des primitives d'accès P et V

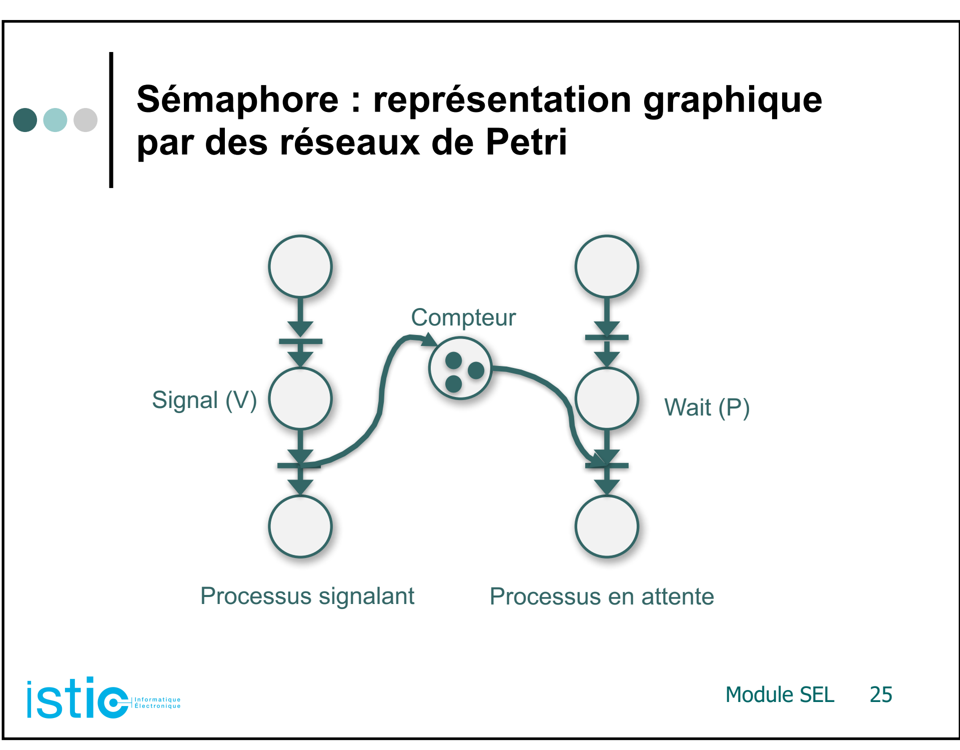
```
typedef struct {int e} sémaphore;
void P (sémaphore *s) {
    when (s → e>0)
        s → e = s → e-1; //Tant que pas vrai, on attend
}
void V (sémaphore *s) {
    s → e = s → e+1;
}
```

## Sémaphore

- Si un processus *p* appelle P(s) alors que s.e est égal à 0, *p* attend jusqu'à ce que s.e devienne strictement supérieur à 0. *p* est bloqué sur le sémaphore
- L'opération P peut conduire au blocage de *p*. Elle matérialise un test sur une condition de synchronisation : « Puis-je continuer ? »
- L'opération V peut libérer un processus. Elle matérialise une évolution de la valeur des conditions de synchronisation. Elle ne bloque personne : « Vas-y, continues !' »

## Sémaphore

- Si plusieurs processus sont bloqués sur un sémaphore, rien n'est dit sur le processus qui sera libéré par un V
- Le sémaphore doit être partagé par les processus désirant se synchroniser (variable globale)
- Dans l'opération P il y a une attente. Dans la mise en œuvre il faudra faire attention à ce que cela n'entraîne pas d'attente active sur le processeur.




## Sémaphore
## Description équivalente

Description équivalente de P et V :

```
typedef struct {int e, file f} sémaphore;
void P (sémaphore *s) {
    s → e = s → e-1;
    if (s → e < 0) {
            insérer(s → f,p) ; bloquer p;
    }
}
void V (sémaphore *s) {
    if (s → e<0) {
            p = ôter(s.f); réveiller(p);
    }
    s → e = s → e+1;
}
```

## Sémaphore
## Description équivalente

- Le compteur peut maintenant être négatif
- Même fonctionnement qu'au préalable
- s → f met en file les processus bloqués en attente du sémaphore (plus proche de la mise en œuvre)
- Dans l'opération P il y a une attente. Dans la mise en œuvre il faudra veiller à ce que cela n'entraîne pas d'attente active
- Réveillé ne veut pas dire exécuté
- Code pour assurer l'atomicité n'est pas donné

## Sémaphore
## Description équivalente

- Signification de l'attribut s → e :
  - Si s → e ≥ 0, s → e est égal au nombre de processus pouvant appeler P sans se bloquer
  - Si s → e < 0, s → e est égal au nombre de processus bloqués

## Exemples de synchronisation Exclusion mutuelle

- Propriété à assurer : au plus un processus en section critique à la fois
  - Entrée en section critique peut conduire à un blocage ⇒ utilisation d'un P
  - Valeur initiale du sémaphore = 1 (initialement un processus et un seul peut faire P sans se bloquer)
- Code

```
sema mutex(1); // valeur initiale du sémaphore
P(mutex);
<section critique>;
V(mutex);
```

## Exclusion mutuelle : propriétés de la solution

- On nomme *nc* le nombre de processus en section critique)
- Sûreté (exclusion)
  - D'après la structure du code, on a nc=nf(mutex)-nv(mutex)
  - D'après (3), on a nf(mutex)=min(np(mutex), 1+nv(mutex))
  - ou en reformulant : nf(mutex) ≤ 1+nv(mutex) et donc nc ≤ 1+nv(mutex) - nv(mutex), soit nc ≤ 1

## Exclusion mutuelle : propriétés de la solution

- Vivacité (absence de blocage inutile)
  - Supposons qu'aucun processus ne soit en section critique.
  - D'après la structure du code, on a nc=0, ou encore nf(mutex)=nv(mutex)
  - Ceci peut se ré-écrire nf(mutex)<nv(mutex)+1
  - D'après (3), on a nf(mutex)=np(mutex), donc on n'a pas de processus bloqué
- Absence de privation : non vérifié en l'absence de règle sur le processus libéré par un V (entre autres)

## Exclusion mutuelle et interblocage

- L'imbrication de sections critiques peut conduire à une situation où tous les processus sont bloqués indéfiniment (interblocage)

```
sema mutex1(1); // contrôle de la section critique SC1
sema mutex1(1); // contrôle de la section critique SC1
Processus P1:              Processus P2
      ....                       ....
      P(mutex1); (1)             P(mutex2) (3)
      P(mutex2); (2)             P(mutex1) (4)
      V(mutex2)                  V(mutex1)
      V(mutex1)                  V(Mutex2)
      ...                  ...
```

- Si on exécute, dans l'ordre (1), (3), (2), (4) les deux processus sont bloqués définitivement

istic

## Exclusion mutuelle et interblocage

- Une situation d'interblocage peut se rencontrer dans de nombreux cas  d'allocation de ressources
- Situation typique d'interblocage : P en section critique
- L'emboitement des sections critiques peut être ``cachée'' par des appels de procédure
- Solutions aux problème d'interblocage vues plus tard

## Exemples de synchronisation
## Attente d'événement

- Un processus doit attendre qu'un événement extérieur se produise avant de poursuivre son exécution au-delà d'un certain point (exemple : attente de fin d'E/S)
- L'événement attendu peut être arrivé avant que le processus ne se mette en attente ⇒ dans le cas l'arrivée de l'événement doit être mémorisée
- Code

  **sema** sbloc(0);
  Attente de l'événement : P(sbloc)
  Arrivée de l'événement : V(sbloc)
- De nombreux autres cas seront vus en TD

## Interface Unix

- Sémaphores System V
  - int semget(key_t key, int nsems, int flag);
    Création d'un groupe de sémaphores
  - int semop(int semid, struct sembuf *array, size_t nops);
    Opérations sur groupe de sémaphores (P/V). La valeur à ajouter/retirer au compteur et le sémaphore concerné dans le groupe sont des arguments de l'appel système (array)
  - int semctl(int semid, int semnum, int cmd, ...);
    Opérations de contrôle sur le sémaphore (récupération compteur, destruction)

## Interface Unix

- Le sémaphore est un objet système, problème d'accord sur l'identificateur du sémaphore
  - Utilisation du paramètre *key*
  - *ftok* : Production d'une clé unique
  - key_t ftok (char*pathname, int proj_id);
    Génération d'une clé à partir d'un nom de fichier et d'un numéro (non nul)
    - Mêmes arguments par deux processus : mêmes résultats
    - Possibilité de collisions

## Interface Unix

- Sémaphores Posix :
  - sem_init (&s, ..., val) : initialisation d'un sémaphore
  - sem_wait(&s) : attente (P)
  - sem_post(&s) : signalement (V)
  - sem_getvalue(&s) : récupération de la valeur du compteur
  - sem_destroy(&s) : destruction du sémaphore

Module SEL 37

## Interface Unix

- Interface plus facile d'utilisation que les sémaphores System V (pas de groupe de sémaphores)
- Permettent uniquement de se synchroniser entre threads d'un processus

⇒ Utilisation des sémaphores System V pour se synchroniser entre processus différents

⇒ Pas de problème d'accord sur le nommage

Module SEL 38

## Sémaphores à compteur vs sémaphore d'exclusion mutuelle (lock)

- Etat sémaphore d'exclusion mutuelle : free/busy (binaire)
- Etat d'un sémaphore : quand free, compte le nombre de P possibles avant blocage (comptage)

⇒ Sémaphores à compteurs peuvent être utilisés pour des problèmes de synchronisations plus généraux que les locks (dont l'exclusion mutuelle)

⇒ Locks dédiés à l'exclusion mutuelle, doivent être associés à d'autres outils de synchronisation

## Futex

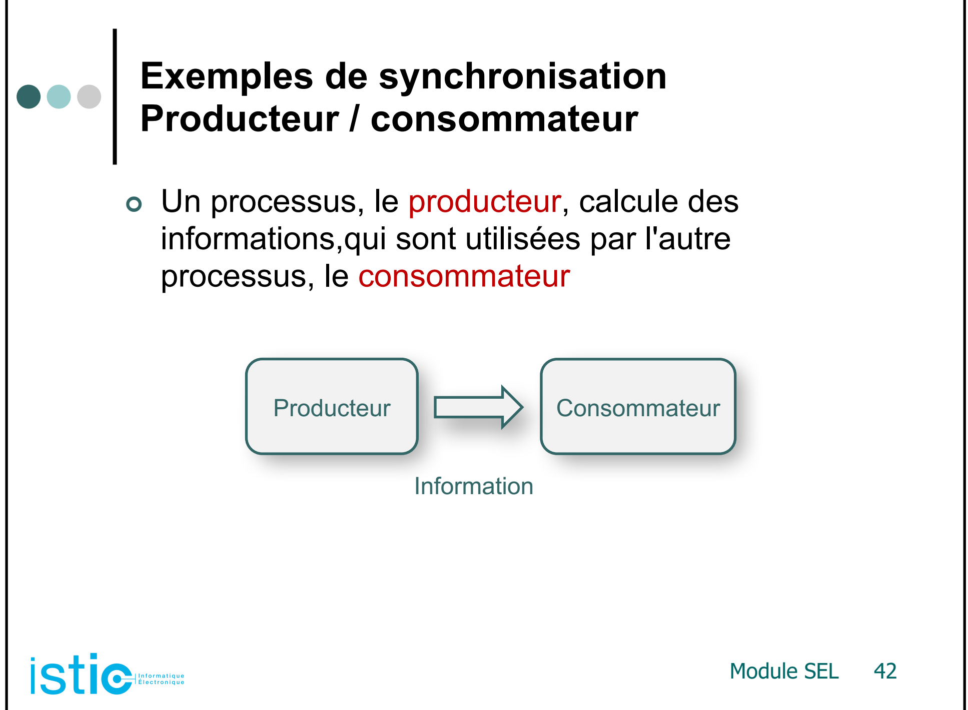
- Spécificité Linux (au moins à la base)
  - Apparus dans la version 2.5.7
  - Fast Userspace muTEX
- Objectif : synchronisation rapide
  - Entier en mode utilisateur, manipulé par instructions atomiques
  - File d'attente en mode système
  - Intérêt : appels systèmes uniquement en cas de blocage, utilisables pour implémenter des sémaphores / mutex posix

# Futex

- Interface
  - WAIT(addr, val)
    - If the value stored at the address addr is val, puts the current thread to sleep.
  - WAKE(addr, num)
    - Wakes up num number of threads waiting on the address addr.

# Exemples de synchronisation Producteur / consommateur

- Un processus, le producteur, calcule des informations,qui sont utilisées par l'autre processus, le consommateur



## Producteur / consommateur Contraintes de synchronisation

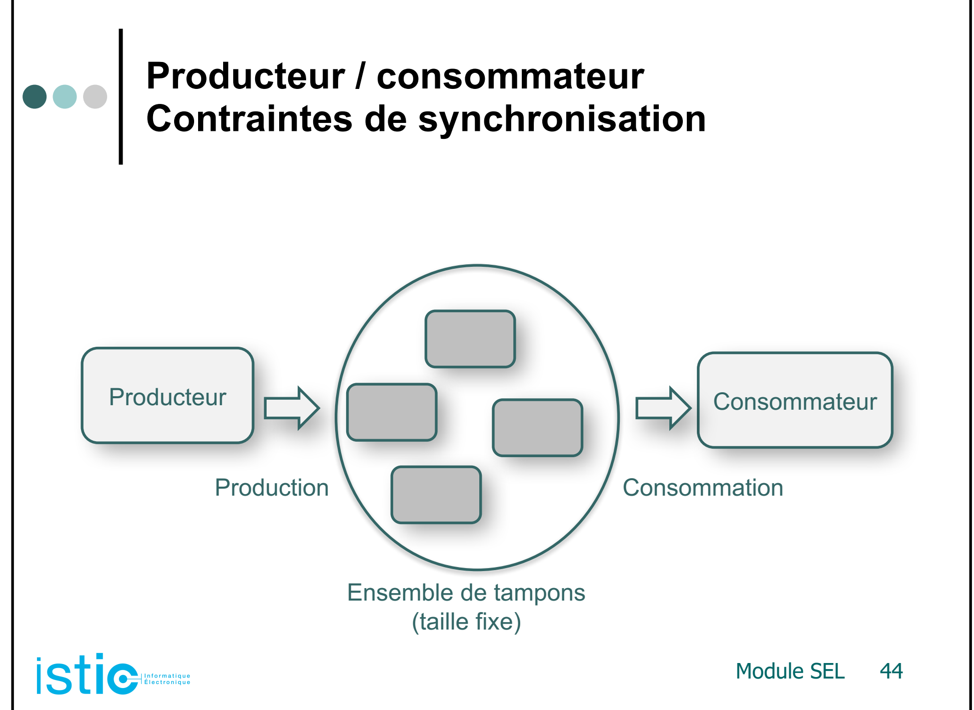
- Le consommateur ne peut pas aller plus vite que le producteur : il ne peut s'exécuter que s'il y a des informations produites et non consommées
- Le producteur peut aller plus vite que le consommateur ⇒ pour ne par le freiner, on mémorise l'information dans un ensemble de tampons (de nombre borné)
- Un tampon est plein s'il contient une information produite et pas encore consommée, sinon il est vide
- Le producteur ne peut s'exécuter que s'il a au moins un tampon vide

istic Module SEL 43



## Producteur / consommateur Contraintes de synchronisation






istic Module SEL 44

## Producteur / consommateur Contraintes de synchronisation

- On ne s'intéresse pas ici à la façon dont est géré l'ensemble de tampons (tableau, liste, ...)
- Procédures d'accès aux tampons :
  - *produire(d)* recopie une information d dans un tampon vide et de passer au tampon suivant, le tampon passe alors dans l'état plein (résultat indéterminé s'il n'existe pas de tampon vide) ;
  - *consommer()* permet de rendre en résultat le contenu d'un tampon plein et de passer au suivant, ce qui a pour effet de "vider » le tampon
- Ces deux procédures ne comportent pas de synchronisation, elles permettent juste de masquer la gestion des tampons (listes, tableaux).

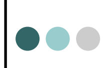
## Producteur / consommateur Contraintes de synchronisation

```
processus producteur
{
info d;
while (1)  {
        d = calcul;
        attendre "tampon vide";
        produire(d);
}
```

```
processus consommateur
{
info d;
while (1) {
        attendre "tampon plein"
        d = consommer();
        calcul(d);
}
```

## Producteur / consommateur
## Contraintes de synchronisation

- Condition pour produire : nb tampon vide > 0
  - Or, nb tampon vide = nb tampon - nb production + nb consommation
  - Formule analogue à celle donnant la valeur du compteur d'un sémaphore s : $s.e = s.e_0 - nf(s) + nv(s)$
  - Un sémaphore s_vide compte le nombre de tampons vides}
  - On effectue un P sur ce sémaphore avant chaque production
  - On effectue un V sur ce sémaphore après chaque consommation
- Condition pour consommer : nb tampon plein > 0
  - Un raisonnement similaire amène à utiliser un sémaphore s_plein qui compte le nombre de tampons pleins
  - P sur ce sémaphore avant chaque consommation
  - V sur ce sémaphore après chaque production



## Producteur / consommateur
## Contraintes de synchronisation

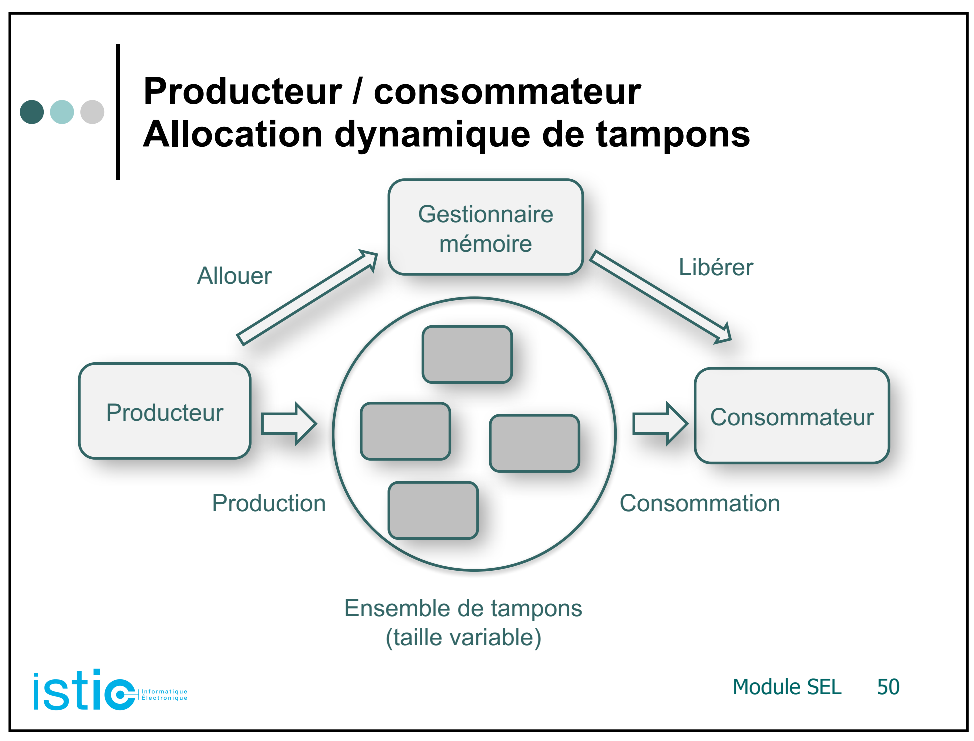
```
sema s_plein(0);
sema s_vide(NB_TAMPONS);
processus producteur
{
info d;
while (1)  {
        d = calcul;
        P(s_vide);
        produire(d);
        V(s_plein);
}

processus consommateur
{
info d;
while (1) {
        P(s_plein);
        d = consommer;
        calcul(d);
        V(s_vide);
}
```

## Producteur / consommateur
## Contraintes de synchronisation

- Exclusion mutuelle sur chaque tampon ?
  - Superflue, déjà assuré par la synchronisation existante
- Plusieurs producteurs / consommateurs ?
  - Soit être modifié pour gérer l'exclusion mutuelle sur les variables partagées identifiant les tampons où produire/consommer (procédures produire/consommer)
- Raisonnement ici avec un nombre fixe de tampons
  - Peut être modifié pour allouer dynamiquement les tampons dans lesquels produire



## Producteur / consommateur
## Allocation dynamique de tampons



## Producteur / consommateur
## Allocation dynamique de tampons

```
sema s_plein(0);
```

```
processus producteur
{
info d;
while (1)  {
        d = calcul;
        allouer();
        produire(d);
        V(s_plein);
}
```

```
processus consommateur
{
info d;
while (1) {
        P(s_plein);
        d = consommer();
        calcul(d);
        libérer();
}
```

## Producteur / consommateur
## Allocation dynamique de tampons

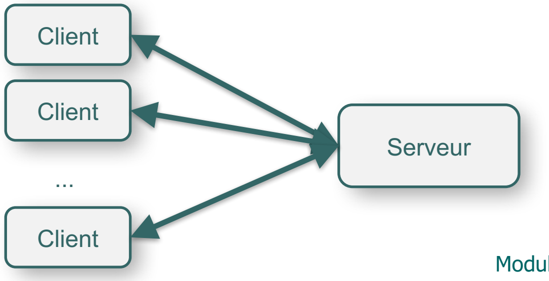
- Le sémaphore *s_vide* disparaît du code des processus
- Cela ne signifie pas qu'il n'y aura plus jamais de blocage dans le producteur
  - Blocage s'il n'y a plus d'espace mémoire
  - Ou pénurie mémoire

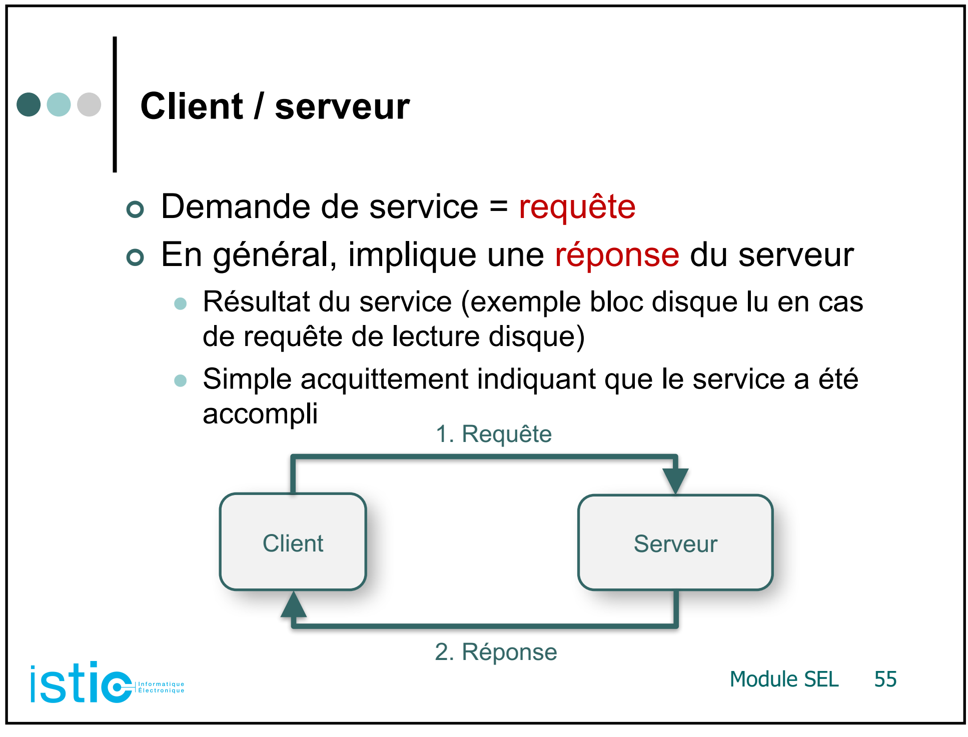
## Producteur / consommateur

- Remarques
  - Si les processus producteurs et consommateurs ont des vitesses constantes, les performances maximales sont obtenues avec deux tampons (production d'un tampon en parallèle avec consommation sur l'autre)
    - ⇒Le débit du couple producteur/consommateur est fixé par le processus le plus lent
    - ⇒ Augmenter le nombre de tampons ne fait qu'augmenter la consommation mémoire
  - A vitesse variable l'existence de plus de deux tampons permet de traiter les a-coups, sans obliger un processus à attendre

## Client/serveur

- Principe
  - Ensemble de processus (clients) qui demandent un travail à un processus spécialisé (serveur)
  - Type de travail : traitement de requêtes d'E/S disque, allocation de ressource, etc.






## Client / serveur

- Remarques
  - Double producteur/consommateur ?
    - Vrai pour les requêtes
    - Faux pour les réponses : la réponse à une requête particulière doit être envoyée au client ayant émis cette requête

## Client / serveur

- Bloc de requête contenant à la fois
  - La description du travail demandé (requête)
  - La réponse du serveur (vide au départ)
- File des requêtes : mémorisation des demandes de services demandées et pas encore traitées
  - Procédures d'accès *déposerReq* et *prendreReq* assurant que la file est accédée en exclusion mutuelle
  - Pas de capacité de stockage maximale dans la file de requête,
  - Même espace mémoire pour client et serveur



## Client / serveur

## Client / serveur
## Contraintes de synchronisation

- (1) Le serveur ne peut s'exécuter s'il n'y a pas de requête à traiter
- (2) Le client ne peut pas utiliser la réponse tant que le serveur ne la lui a pas communiquée

## Client / serveur
## Contraintes de synchronisation

- Contrainte (1)
  - Sémaphore de comptage des requêtes, s_nbreq
  - P lors du retrait par le serveur
  - V lors du dépot par le client
- Contrainte (2)
  - Il faut réveiller le client qui a demandé le service
  - On utilise donc un sémaphore par client
  - Valeur initiale du sémaphore = 0 (sémaphore bloquant
  - Transmission dans le bloc de requête

## Client / serveur
## Contraintes de synchronisation

```
typedef demande d, sema attRep(0), reponse r blocReq;
sema s_nbreq (0);  sema mutexFreq (1);
file blocReq fileReq;

void deposerReq (pointeur blocReq ptb) {
        P(mutexFreq);
        mettre le bloc de requête repéré par ptb dans fileReq;
        V(mutexFreq);
}
blocReq prendreReq (void) {
        P(mutexFreq);
        enlever un bloc de fileReq et rendre son adresse;
        V(mutexRreq);
}
```

Module SEL 61

## Client / serveur
## Contraintes de synchronisation

```
processus client {
blocReq rq;
...
rq.d = initialisation de la
requête;
deposerReq(\&rq);
V(nbreq);
P(rq.attRep);
exploiter la réponse(rq.r);
}
```

```
processus serveur {
blocReq *ptb;
while (1) {
P(nbreq);
ptb:=prendreReq();traitement
requête (ptb → d);
ptb → r = reponse ;
V(ptb → attRep);
}
```

Module SEL 62

## Exemples de synchronisation Sémaphores privés

- Un sémaphore privé est un sémaphore sur lequel un seul processus (le propriétaire) peut faire un P.
- Intérêt :
  - Quand on fait un V on sait exactement quel processus on réveille (le processus propriétaire du sémaphore)

  ⇒ Il est possible d'exprimer des contraintes de synchronisation complexes, délicates à écrire directement avec des sémaphores ``standard''

## Autres exemples de synchronisation

- Rendez-vous, ou barrières de synchronisation
- Lecteur-rédacteur
- Utilisation d'une ressource parmi N

- Seront examinés en TD

## Bestiaires des outils de synchronisation

- Messages
- Sémaphores avec messages
- Sémaphores d’exclusion mutuelle
- Verrous lecteur/rédacteur
- Moniteurs
- Transactions

## Messages

- Associe synchronisation et transfert d'information
- Interface typique :
  - **void** envoyer(**identité** destinataire, **message** m) provoque l'émission du message *m* vers l'entité désignée par *destinataire*, elle est non bloquante (⇒ stockage des messages dans une file)
  - **void** recevoir(**identité** *source, **message*** m) rend un message et l'identité de l'entité émettrice, bloque le processus demandeur s'il n'a pas de messages émis

## Messages : client/serveur

**processus** client {
**requête** rq;
**réponse** r;
**identité** id;
...
rq =init requête;
envoyer(serveur, rq);
recevoir(id, &r);
exploiter la réponse(r);
}

**processus** serveur {
**requête** rq;
**réponse**} r;
**identité**} id;
**while** (1) {
recevoir(id, &rq);
traiter requête(rq);
r = calcul réponse;
envoyer(id, r);
}}

## Messages : producteur / consommateur

**processus** producteur {
**info** m;
**while** (1) } {
m =calcul;
envoyer(consommateur,m);
}}

**processus** consommateur {
**info** m;
**while** (1) {
recevoir(producteur,&m);
calcul(m);
}}

## Messages : producteur / consommateur

- Il n'y a pas dans cette version de contrôle à l'émission

⇒ Si la file de messages en attente n'est pas de taille bornée, le consommateur sera submergé de messages

⇒ Dans ce cas, on peut prévoir une circulation de messages dans l'autre sens ("autorisation » de produire)

## Messages

- Remarques
  - Boîte à lettres : file par processus. Dans ce cas, *source* et *destinataire* désignent le processus
  - Port : file par port, ports créés selon les besoins.
    - Permet de distinguer les sources de messages, et éventuellement d'attendre un message d'une source particulière.
    - Dans ce cas, *source* et *destinataire* désignent le port
  - Gestion mémoire : recopies de messages lors de l'envoi d'un message (à optimiser !)

## Messages

- Remarques
  - Emission bloquante/non-bloquante : émission non bloquante suppose qu'il y a toujours de la place pour accueillir le message dans la file.
  - Une mise en œuvre avec file de taille bornée implique un cas de blocage à l'émission (cas de file pleine)
  - Ce mécanisme reste de bas niveau : il n'est pas toujours aisé à manipuler pour exprimer des contraintes de synchronisation complexes

## Messages : interface POSIX

- Message queues :
  - int msgget(key_t key, int msgflg);
    Création d'un file de messages
  - int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
    Envoi d'un message sur une file
  - size_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
    Réception d'un message dans une file
  - int msgctl(int msqid, int cmd, struct msqid_ds *buf);
    Opération de contrôle sur une file de message (destruction, consultation taille de file, etc)

## Sémaphores d'exclusion mutuelle (verrous, locks)

- Optionnellement
  - Pas de blocage dans le cas d'acquisition de verrou par le même processus que celui qui le possède déjà

```
lock(l);
….
lock(l); // potentiellement masqué par appel fonction
unlock(l);
…
unlock(l);
```

  - Test du fait que l'acquisition et la libération du verrou sont réalisés par le même processus

## Verrous lecteurs/rédacteurs

- Schéma de synchronisation très courant lors de l'accès aux données (fichiers, bases de données)
- Distinction de deux types d'acquisition du verrou :
  - Lecture seule : plusieurs accès simultanés en lecture sont autorisés
  - Ecriture : jamais deux écritures en même temps ou une lecture en même temps qu'une écriture

## Verrous lecteurs/rédacteurs – interface POSIX

- Interface bibliothèque pthread (extrait)
  - int pthread_rwlock_init(pthread_rwlock_t *lock, const pthread_rwlockattr_t *attr);
    Initialize a read/write lock object.
  - int pthread_rwlock_rdlock(pthread_rwlock_t *lock)
    Lock a read/write lock for reading, blocking until the lock can be acquired.
  - int pthread_rwlock_wrlock(pthread_rwlock_t *lock)
    Lock a read/write lock for writing, blocking until the lock can be acquired.
  - int pthread_rwlock_unlock(pthread_rwlock_t *lock)
    Unlock a read/write lock.

## Variables de conditions POSIX

- Fonctionnement
  - Lock pour protéger une variable partagée, utilisés normalement
  - Fonctions d'attente (*pthread_cond_wait*) et de signal (*pthread_cond_signal*)
    - Mutex automatiquement et atomiquement relâché pendant l'attente
    - Mutex automatiquement et atomiquement verrouillé lors du réveil
  - Signal ne fait rien quand aucun thread n'est bloqué (faire un signal avant un wait est une erreur)

## Variables de conditions POSIX

- Interface bibliothèque pthread (extrait)
  - int pthread_cond_signal(pthread_cond_t *cond)
    Unblock of the threads blocked on the specified condition variable.
  - int pthread_cond_broadcast(pthread_cond_t *cond)
    Unblock all threads currently blocked on the specified condition variable.
  - int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *mutex)
    Unlock the specified mutex, wait for a condition, and relock the mutex.

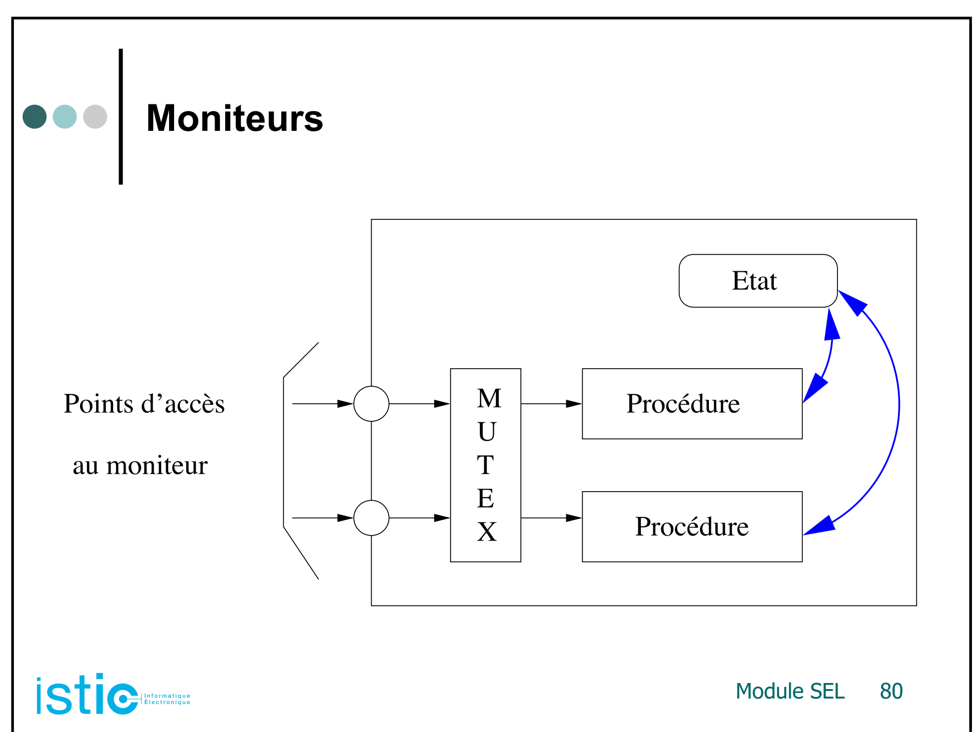
## Moniteurs

- Notion de moniteur : Hoare 1974, Brinch Hansen 1975
- Outil de plus haut niveau que les sémaphores, destiné à faciliter l'écriture de programmes parallèles corrects
  - Concept du niveau langage
  - Peut être mis en œuvre en utilisant des mécanismes de synchronisation de plus bas niveau, tels que les sémaphores

## Moniteurs

- Moniteur constitué
  - D'un ensemble de variables d'état, inaccessibles directement aux utilisateurs du moniteur
  - D'un ensemble de procédures manipulant ces variables, et accessibles aux utilisateurs du moniteur.
  - Ces procédures s'exécutent en exclusion mutuelle

## Moniteurs



## Moniteurs

- Conditions : variable d'état d'un type particulier
  - Opérations sur une condition *c* (*p* désigne le processus qui s'exécute)
    - c.attendre (wait) : bloque le processus p, on dit que *p* est "en attente" de *c*
    - c.vide (empty) : rend *vrai* si aucun processus n'est "en attente" de c
    - c.signaler (notify) : réveille un des processus en attente de *c*, s'il en existe (c.vide *vrai*), ne fait rien sinon
  - Conditions manipulées en exclusion mutuelle

## Moniteurs

- Remarques
  - Contrairement aux sémaphores, une condition ne mémorise pas.
    - L'opération signaler effectuée sur une condition quand aucun processus n'est bloqué n'a aucun effet.
    - Un appel à attendre est toujours bloquant
  - Similitudes avec les sémaphores privés
  - Exclusion mutuelle + blocage sur les conditions ⇒ relâcher l'exclusion mutuelle pendant l'attente (sinon, personne ne peut exécuter le signaler)
    - ⇒ Mise en œuvre non triviale

## Java : parallélisme et synchronisation

- Thread : objet particulier
  - Par héritage de la classe *Thread*
  - Par définition d'une classe qui implémente l'interface *Runnable*
- Ordonnancement des threads
  - Préemptif
  - Fonctions de contrôle explicite (yield, sleep, join)
  - Priorités

## Java : parallélisme et synchronisation

- Verrous (attribute synchronized)
  - Attribut d'une méthode ou d'un bloc
  - Empêche tout appel concurrent à une autre méthode ou bloc *synchronized* du même objet
  - Assure l'exclusion mutuelle
    - Uniquement sur les méthodes / blocs *synchronized*
  - Verrous réentrants : un même thread peut entrer dans plusieurs blocs *synchronized* du même objet sans blocage

## Java : parallélisme et synchronisation

```
Class Color {
        private int red, green, blue;
        public synchronized void put (int r, int g, int b) {
                // en exclusion mutuelle
                red = r ; green = g; blue = b;
        }
        public synchronized int transfoTSL() {
                // en exclusion mutuelle
                ...
        }
}
```



## Java : parallélisme et synchronisation

- Sémaphores
  - Classe Sémaphore : sémaphores à compteur
  - Méthodes
    - acquire (= P)
    - release (= V)
  - Erreurs faciles
    - Mécanisme de bas niveau
    - Synchronisations disséminées dans le code

## À retenir

- Parallélisme : synchronisation obligatoire
- Outils de synchronisation peu nombreux (sémaphores, verrous d'exclusion mutuelle, etc.)
- Synchronisation non triviale
- Mais schémas de synchronisation typiques peu nombreux