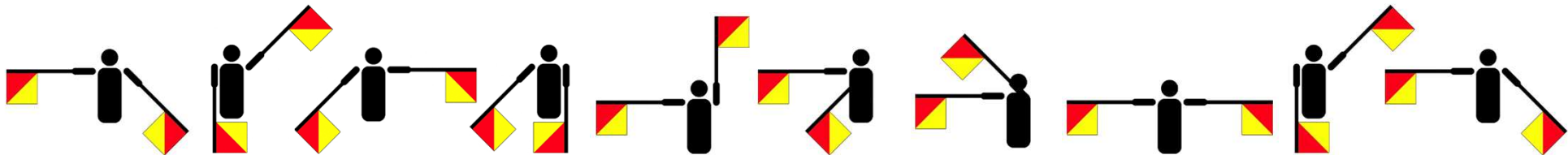


Les sémaphores



❑ Concept introduit par Dijkstra en 1965 pour **synchroniser** des activités parallèles (processus ou threads)

- Section critique
- Exclusion mutuelle
- Synchronisation répondant à la logique d'une application

❑ **Principe** : un sémaphore représente un distributeur de tickets

- Un processus peut
 - ❖ **Prendre** un « ticket » (ou « jeton »)
 - ❖ **Déposer** un ticket
- En l'absence de ticket, un processus demandeur est **bloqué**
- Lorsqu'un ticket est déposé
 - ❖ S'il y a un processus en attente de ticket, celui-ci est **débloqué**
 - ❖ S'il n'y a personne en attente, le ticket sera pris par le prochain demandeur

Un sémaphore vu comme un objet (Syntaxe « à la C++ »)

- Un sémaphore est le regroupement d'un **compteur** et d'une **file d'attente**

```
class Semaphore {
```

```
    private :                // valeur > 0 = nombre de tickets disponibles
        int valeur;          // valeur < 0 = nombre de processus en attente d'un ticket
        File processusEnAttente; // les processus demandeurs bloqués en attente d'un ticket
};
```

- Trois **opérations** (**publiques**) y sont associées

```
void initialiser (Semaphore &S, unsigned int nbTicketsInitial);
```

```
    // fixer le nombre initial de tickets au démarrage de l'application
```

```
void P (Semaphore &S);    // prendre un ticket :
```

```
    // le processus appelant ne termine cette opération que lorsqu'il a obtenu un ticket
```

```
void V (Semaphore &S);    // déposer un ticket :
```

```
    // ce qui peut conduire à débloquer un processus bloqué en attente d'un ticket
```

// Fixer le nombre initial de tickets au démarrage de l'application

void initialiser (Semaphore &S, unsigned int nbTicketsInitial) {

S.valeur = nbTicketsInitial;

}

// Le processus courant prend un ticket

// Il ne termine cette opération que lorsqu'il a obtenu un ticket

void P (Semaphore &S) {

// Tenter de prendre un ticket

S.valeur = S.valeur – 1;

if (S.valeur < 0) {

// Il n'y avait pas de ticket disponible, le processus courant est bloqué

bloquer(processusCourant(), S.processusEnAttente);

}

// Le processus a obtenu un ticket, immédiatement ou après avoir attendu

}

// Déposer un ticket, ce qui peut débloquent un processus en attente d'un ticket

void V (Semaphore &S) {

// Un ticket de plus dans le distributeur

S.valeur = S.valeur + 1;

if (S.valeur <= 0) {

// La valeur était donc négative, traduisant qu'au moins un processus attendait un ticket :

// débloquent le processus en tête de la file

debloquent(processusEnTete(S.processusEnAttente));

// Le processus demandeur a été débloquent :

// il peut avoir aussi pris l'UC au processus qui a déposé le ticket

}

}

```
void P (semaphore &S) {  
    if (S.valeur > 0)  
        S.valeur = S.valeur - 1;  
    else  
        Bloquer le processus courant dans S.processusEnAttente;  
}
```



```
void V (semaphore &S) {  
    if (S.processusEnAttente non vide)  
        Débloquer le processus en tête de la file S.processusEnAttente;  
    else  
        S.valeur = S.valeur + 1;  
}
```

Comment est implanté un sémaphore ?

- ❑ Le système gère les **opérations** sur un même objet sémaphore en **exclusion mutuelle**, afin de garantir la cohérence de cet objet **partagé**
- ❑ Comment ?
 - En utilisant les mécanismes d'exclusion mutuelle précédemment présentés
- ❑ On construit donc un mécanisme de synchronisation « évolué » avec **attente passive** en utilisant des mécanismes primitifs existants. Par exemple :
 - Masquage des interruptions durant l'exécution de ces opérations en monoprocesseur
 - Problèmes dans le cas des multiprocesseurs
 - ❖ Action atomique avec mémoire partagée
 - ❖ Instruction test-and-set
 - Si un bit est à 0 alors met à 1 et retourne 0
 - Sinon retourne 1
 - ❖ Instruction swap entre 1 et le bit de verrou
 - Si le bit est à 0 alors OK
 - Sinon re-tester ultérieurement

Le but dans ce cours n'est pas d'implanter
un tel mécanisme
mais d'utiliser des sémaphores
(via les opérations P/V présentées)
pour synchroniser des applications parallèles

Utilisation par 2 activités parallèles d'un sémaphore n'ayant aucun ticket initialement

```
Semaphore sem; // Variable supposée partagée  
initialiser(sem, 0);
```

```
void P1 {  
    ...  
    V(sem);  
    ...  
}
```

```
void P2 {  
    ...  
    P(sem);  
    ...  
}
```


Utilisation par 2 activités parallèles d'un sémaphore possédant un ticket initialement

```
Semaphore mutex; // Variable supposée partagée  
initialiser(mutex, 1);
```

```
void P1 {  
    ...  
    P(mutex);  
    Section critique;  
    V(mutex);  
    ...  
}
```

```
void P2 {  
    ...  
    P(mutex);  
    Section critique;  
    V(mutex);  
    ...  
}
```

Exemple 3 – Gestion d'un pool d'imprimantes

```
class GI {           // Gestion d'Imprimantes  
private  
    ...  
public  
    // Demander une imprimante  
    // et récupérer le numéro de l'imprimante allouée  
    int demanderImprimante (void);  
  
    void rendreImprimante (int numImp);  
};
```

Exemple 3 – Synchronisation des accès aux imprimantes

```
#define NB_IMP ...  
{  
    // Variables supposées partagées  
    bool occupe[NB_IMP];  
}
```

```
int demanderImprimante (void) {  
    int i;  
  
    i = 1;  
    while (occupe[i]) i++;  
    occupe[i] = true;  
    return (i);  
}
```

```
void rendreImprimante (int numImp) {  
    occupe[numImp] = false;  
}  
  
void init (void) {  
    int i;  
  
    for (i = 1; i < NB_IMP; i++)  
        occupe[i] = false;  
}
```

Exemple 3 – Synchronisation des accès aux imprimantes

```
#define NB_IMP ...  
{  
    // Variables supposées partagées  
    bool occupe[NB_IMP];  
    Semaphore ImpLibre;  
}
```

```
int demanderImprimante (void) {  
    int i;  
  
    P(ImpLibre);  
    i = 1;  
    while (occupe[i]) i++;  
    occupe[i] = true;  
    return (i);  
}
```

```
void rendreImprimante (int numImp) {  
    occupe[numImp] = false;  
    V(ImpLibre);  
}
```

```
void init (void) {  
    int i;  
    initialiser(ImpLibre, NB_IMP);  
    for (i = 1; i < NB_IMP; i++)  
        occupe[i] = false;  
}
```

Cela marche-t-il ?

- ☐ **Concept de bas niveau**
- ☐ **L'utilisation reste simple dans le cas de l'exclusion mutuelle**
- ☐ **Une utilisation systématique peut être source d'erreurs**
 - **Un seul oubli ou un appel mal situé perturbe l'application complète**
 - **L'exclusion peut ne pas être assurée ou un interblocage peut apparaître dans un cas très rare**
- ☐ **Des outils plus structurés sont nécessaires dans les cas plus complexes**
- ☐ **Aucun langage évolué ne s'appuie uniquement sur les sémaphores**

☐ Sémaphore booléen

- Équivalent à un sémaphore à un seul jeton
- Utilisation par une alternance de P et de V

☐ Sémaphore privé

- Seul le processus propriétaire se bloque derrière ce sémaphore (opération P)
- La file d'attente n'est plus nécessaire
- Tout processus peut effectuer une opération V sur le sémaphore

❑ Exercices théoriques avec P et V

- Modèle du producteur-consommateur
- Rendez-vous

❑ Comment synchroniser des threads Posix [utilisé en TP]

- Avec des sémaphores d'exclusion mutuelle Posix ou verrous : type `pthread_mutex_t`
- Avec des sémaphores à compteur Posix : type `sem_t`

❑ Comment synchroniser des Processus Unix [Hors programme]

- Présentation de la bibliothèque des IPC (InterProcess Communication) Unix System V
 - ❖ Segment de mémoire partagée : pour enfin partager des données entre processus (parents ou non) !
 - ❖ Sémaphores : pour synchroniser des processus
 - ❖ Files de messages : pour aller plus loin que les tubes en échangeant des informations entre processus non parents

Exercices « théoriques »

**[Ces exercices seront aussi concrètement réalisés en TP
en considérant que les activités parallèles sont des threads Posix]**

Exercice 1 – Affichage alterné

- ❑ On suppose qu'une activité parallèle possède le comportement suivant :

Activité Afficheur {

while (1) {

Effectuer un traitement ;

Afficher un message de plusieurs lignes à l'écran ;

Effectuer un traitement ;

}

}

- ❑ On souhaite synchroniser, à l'aide de sémaphores, plusieurs activités de ce type pour qu'elles alternent leurs messages à l'écran
- ❑ Proposer une solution pour 2 activités parallèles
- ❑ Généraliser la solution à N activités parallèles

Exécution souhaitée pour 2 activités parallèles :

...

Activité 1 : début de mon message

Activité 1 : fin de mon message

Activité 2 : début de mon message

Activité 2 : fin de mon message

Activité 1 : début de mon message

Activité 1 : fin de mon message

Activité 2 : début de mon message

Activité 2 : fin de mon message

...

...

Exercice 2 – Modèles des producteurs/consommateurs

- ☐ On considère un buffer partagé par des activités parallèles de deux types :
 - Des producteurs qui déposent des messages dans ce buffer
 - Des consommateurs qui retirent des messages de ce buffer
- ☐ Le buffer comporte N cases et est géré circulairement
- ☐ Variante de base
 - Les dépôts se font dans l'ordre croissant des indices de cases, de manière circulaire
 - Les retraits se font dans l'ordre des dépôts, de manière circulaire aussi
- ☐ Proposer une solution utilisant des sémaphores pour que ces activités parallèles déposent et retirent leurs messages de manière cohérente

Exercice 2 – Modèles des producteurs/consommateurs

☐ Variante 1

☐ Les messages sont de deux types (0/1, blanc/noir, recto/verso...)

☐ On veut que les dépôts soient alternés dans le buffer

- Un message d'un 1^{er} type, un message du second, etc.
- Les dépôts se font toujours dans l'ordre croissant des indices de cases, de manière circulaire
- Les retraits se font toujours dans l'ordre des dépôts, de manière circulaire aussi

☐ Proposer une solution utilisant des sémaphores pour que ces activités parallèles déposent et retirent leurs messages de manière cohérente

Exercice 2 – Modèles des producteurs/consommateurs

☐ Variante 2

- ☐ Les messages sont de deux types (0/1, blanc/noir, recto/verso...)
- ☐ On n'impose plus de contrainte sur les dépôts
 - Les dépôts se font dans l'ordre croissant des indices de cases, de manière circulaire
 - Les retraits se font dans l'ordre des dépôts, de manière circulaire aussi
- ☐ Mais, les consommateurs précisent quel type de message ils veulent retirer
 - Et sont bloqués s'ils ne peuvent retirer un message du type attendu
- ☐ Proposer une solution utilisant des sémaphores pour que ces activités parallèles déposent et retirent leurs messages de manière cohérente

- ☐ On considère N activités parallèles qui doivent réaliser un rendez-vous
- ☐ Une activité arrivant au point de RdV doit se mettre en attente s'il existe au moins une autre activité qui n'y est pas arrivé
- ☐ Toutes les activités bloquées sur cette « barrière » peuvent la franchir lorsque la dernière y est arrivée
- ☐ Une activité a le comportement suivant
 - Je fais un certain traitement
 - J'arrive au point de RdV (j'attends les autres si elles n'y sont pas...)
 - Et je continue mon traitement
- ☐ Proposer une solution de synchronisation à l'aide des sémaphores

Synchronisation de threads par sémaphores

□ Sémaphores booléens d'exclusion mutuelle (ou verrous)

➤ Normaux

- ❖ Un P est bloquant si le sémaphore a déjà été franchi par un autre thread

➤ Récursifs

- ❖ Un thread qui a franchi le sémaphore en devient propriétaire et peut effectuer plusieurs P sans se bloquer

□ Sémaphores avec compteur

➤ Sémaphores classiques avec compteur (Dijkstra)

Sémaphores d'exclusion mutuelle POSIX : pthread_mutex_t

- ❑ Synchronisation de base : **mutex** et condition
- ❑ Types (**opaques**) : <sys/types.h>
 - pthread_t, pthread_key_t
 - **pthread_mutex_t**
 - pthread_cond_t
 - pthread_once_t
- ❑ Attributs : standard + propres à l'implantation
 - pthread_attr_t
 - **pthread_mutexattr_t**
 - pthread_condattr_t
- ❑ Bibliothèque ➔ #include <pthread.h> + compilation : -lpthread
- ❑ Gestion des erreurs
- ❑ Primitives « thread-safe »

❑ Création avec les caractéristiques par défaut

- Macro, utilisée lors de la déclaration

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

❑ Création en fixant les caractéristiques

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

- mutex = descriptif du mutex, mis à jour
- attr = attributs optionnels pour l'initialisation
- Erreurs : EAGAIN, ENOMEM, EPERM, EBUSY, EINVAL

❑ Initialisé dans l'état **non** verrouillé (avec un ticket) !

❑ Éviter de faire une copie de mutex...

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

➤ mutex = descriptif du mutex à détruire

- ☐ Possible si non verrouillé
- ☐ Ressources libérées
- ☐ Retourne 0 en cas de succès, un code d'erreur sinon
- ☐ Erreurs : EINVAL, EBUSY

- ❑ Changer les caractéristiques par défaut
- ❑ Créer et initialiser par défaut un attribut de mutex

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

- attr = attribut du mutex (valeur par défaut)
- Erreurs : ENOMEM

- ❑ Détruire un attribut de mutex

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

- attr = descriptif
- Objet devient non initialisé
- Erreurs : EINVAL

□ Positionner le type d'un mutex

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

➤ attr = attributs du mutex (valeur par défaut)

➤ type = type du mutex :

- ❖ PTHREAD_MUTEX_NORMAL
- ❖ PTHREAD_MUTEX_ERRORCHECK
- ❖ PTHREAD_MUTEX_RECURSIVE
- ❖ PTHREAD_MUTEX_DEFAULT

➤ Erreurs : EINVAL

□ Obtenir le type d'un mutex

```
int pthread_mutexattr_gettype(pthread_mutexattr_t *attr, int *type);
```

➤ Erreurs : EINVAL

☐ Verrouiller un mutex (opération P)

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

➤ Erreurs : EINVAL, (EAGAIN, EDEADLK)

☐ Tenter de verrouiller un mutex (opération P non bloquante)

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

➤ Erreurs : EINVAL, EBUSY, (EAGAIN, EDEADLK)

☐ Déverrouiller un mutex (opération V)

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

➤ Erreurs : (EINVAL, EPERM)

Sémaphores avec compteurs POSIX : sem_t

❑ Sémaphores Posix (1003.1b)

- `#include <semaphore.h>`
- Type « opaque » : `sem_t`

❑ Créer et initialiser un sémaphore Posix

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

- `sem` : identificateur du sémaphore créé
- `pshared` : 0 ➔ partage entre threads, > 0 ➔ partage entre processus (si supporté)
- `value` : Valeur initiale
- Succès : 0. Échec : -1 + `errno` : EINVAL, ENOSYS

❑ Détruire un sémaphore Posix

- Succès : 0. Échec : -1 + `errno` : (ENOSYS)

```
int sem_destroy(sem_t *sem);
```


❑ Opération P bloquante

```
int sem_wait(sem_t *sem);
```

- Demande un ticket sur le sémaphore référencé
- Succès : 0. Échec : -1 + errno : EINVAL, EINTR, ENOSYS

❑ Opération P non bloquante

```
int sem_trywait(sem_t *sem);
```

- Succès : 0. Échec : -1 + errno : EAGAIN

❑ Opération P temporisée

```
int sem_timedwait (sem_t *sem, const struct timespec *abs_timeout);
```

- Succès : 0. Échec : -1 + errno : ETIMEDOUT

□ Opération V

```
int sem_post(sem_t *sem);
```

- Dépose un ticket dans le sémaphore référencé
- Succès : 0. Échec : -1 + errno : EINVAL

❑ Principe

- Le compteur est systématiquement décrémenté lors d'une opération P

❑ La sémantique du compteur est alors la suivante :

- S'il est positif, le compteur indique le nombre de jetons disponibles
- S'il est négatif, la valeur absolue du compteur indique le nombre de processus en attente derrière le sémaphore

❑ Invariant

- $S.C = S_C_Initial + Nombre_Total_V(S) - Nombre_Total_P(S)$

❑ Obtenir la valeur du compteur d'un sémaphore Posix

```
int sem_getvalue(sem_t *sem, int *sVal);
```

- Succès : 0. Échec : -1 + errno : EINVAL
- sVal ne reflète pas forcément la réalité !

Communication et synchronisation entre processus InterProcess Communication (IPC) UNIX System V

Pour aller plus loin qu'en TP...
en faisant se synchroniser des processus

(Hors cours)

□ Permettent à des processus sans lien de parenté

- De partager des données (via des segments de mémoire partagée – *shared memory*)
- De communiquer par des échanges de messages (via des files de messages – *queues*)
- De se synchroniser (via des sémaphores généraux)

□ Bibliothèques de base

- `#include <sys/types.h>`
- `#include <sys/ipc.h>`

□ Un IPC possède

- Un identificateur externe ou clé (unique sur la machine) de type `key_t`
- Un identificateur interne (unique au sein d'un processus) de type `int`
- Des droits d'accès (`user/group/others`)
- La possibilité de restreindre son utilisation à la descendance de son créateur (clé : `IPC_PRIVATE`)

□ Un IPC est une ressource **persistante** sur une machine

➤ Le nombre d'IPC est **limité**

- ❖ Globalement
- ❖ Pour chaque utilisateur

□ Nécessité de **détruire** un IPC quand il n'est plus utilisé

➤ Par commande Unix (voir le man)

- ❖ **ipcrm** [-m id] [-q id] [-s id]
 - -m : détruire le segment de mémoire partagée identifié par id
 - -q : détruire la file de messages identifiée par id
 - -s : détruire l'ensemble de sémaphore identifié par id

➤ Par programmation

- ❖ Primitive associée à chaque type d'IPC
 - shmctl, msgctl, semctl

□ Visualisation des IPC existant sur une machine

➤ Commande UNIX **ipcs** (voir le man)

Partager des variables entre processus

Segments de mémoire partagée

UNIX System V

- ❑ **#include <sys/shm.h>**
- ❑ **Avant de manipuler un segment de mémoire partagée, un processus doit**
 - Le créer ou l'ouvrir (si un autre processus l'a déjà créé) afin d'obtenir un identificateur interne pour ce segment
 - « L'attacher » à son espace d'adressage i.e. obtenir une adresse référençant ce segment
 - ❖ Par exemple, pour une zone de mémoire partagée représentant un entier, il attache le segment à l'adresse int *p
- ❑ **Il accède ensuite à cette zone partagée via cette adresse**
 - Il peut consulter ou modifier la valeur de la zone de mémoire partagée
 - ❖ Par exemple : `printf("Valeur partagée = %d\n", *p);` ou : `(*p)++;` ou : `*p = *p * 10;`
- ❑ **Quand il ne veut plus manipuler ce segment, il libère l'adresse le référençant en « détachant » ce segment**

Création d'un segment de mémoire partagée

```
int shmget(key_t key, size_t size, int shmflg);
```

➤ key = nom externe associé à ce segment de mémoire partagée

- ❖ Unique, obtenu notamment par `ftok()` [voir man], si des processus non parents veulent y accéder
- ❖ Privé, égal à `IPC_PRIVATE`, si l'utilisation du segment est restreinte aux seuls descendants du créateur du segment

➤ size = taille en octets allouée à ce segment

- ❖ Dépend de l'information à partager
 - Exemple : `sizeof(int)` pour partager une variable entière
 - Exemple : `sizeof(struct maStructure)` pour partager différentes informations regroupées dans une structure

➤ shmflg = indicateur, suite de bits comprenant

- ❖ `IPC_CREAT` : création d'un nouveau segment
- ❖ `IPC_EXCL` : indique si une éventuelle création doit échouer ou non
 - 1 : la création échoue si le nom externe est déjà utilisé
 - 0 : le processus obtient le numéro interne d'un segment déjà créé (ouverture)
- ❖ 9 bits de faible poids spécifiant les droits d'accès au segment si on le crée

❑ **Retourne l'identificateur interne ou -1 (+ errno) si erreur**

Ouverture d'un segment de mémoire partagée

```
int shmget(key_t key, size_t size, int shmflg);
```

➤ **key = nom externe associé au segment**

❖ Le segment doit exister

➤ **size = taille en octets**

❖ Peu importe, fixé à la création

➤ **shmflg = indicateur**

❖ Droits fixés à la création, doivent permettre à l'appelant d'utiliser le segment

❖ IPC_EXCL et IPC_CREAT ne doivent pas être tous les deux positionnés

❑ **Retourne l'identificateur interne ou -1 (+ errno) si erreur**

Contrôle d'un segment de mémoire partagée (1)

- ☐ Changer les droits d'accès, le propriétaire, le groupe . . .
- ☐ Consulter les caractéristiques
 - Propriétaire, groupe
 - Droits
 - Dernière modification
 - Taille . . .
- ☐ Verrouiller / déverrouiller le segment en mémoire centrale
- ☐ Détruire le segment de mémoire partagée

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Contrôle d'un segment de mémoire partagée (2)

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- **shmid** = identificateur interne du segment
- **cmd** = action de contrôle
 - ❖ **IPC_STAT** : récupérer le descripteur du segment (et donc ses caractéristiques) dans la zone pointée par buf
 - ❖ **IPC_SET** : modifier les caractéristiques du segment à partir du descripteur à l'adresse buf
 - ❖ **SHM_LOCK** : verrouiller le segment en mémoire centrale
 - ❖ **SHM_UNLOCK** : déverrouiller le segment en mémoire centrale
 - ❖ **IPC_RMID** : détruire le segment de mémoire partagée
 - ❖ ...

❑ **Retourne 0 si succès, -1 sinon + errno**

Attacher un segment de mémoire partagée

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- shmid = identificateur interne du segment
- shmaddr = adresse à laquelle attacher le segment ou NULL si on laisse choisir le système
- shmflg = indicateur (les 2 valeurs sont possibles : |)
 - ❖ SHM_RND : arrondir ou non l'adresse
 - ❖ SHM_RDONLY : segment en lecture seule ou non

☐ Retourne l'adresse d'attachement ou -1 (+ errno) en cas d'échec

☐ On peut voir cette étape d'attachement comme un « malloc » : le processus doit obtenir une adresse (dans son espace d'adressage) référençant le segment de mémoire partagée (situé en mémoire centrale) avant de pouvoir le manipuler

Détacher un segment de mémoire partagée

```
int shmdt(const void *shmaddr);
```

➤ shmaddr : adresse d'attachement

- ☐ Retourne 0 en cas de succès, -1 (+ errno) sinon
- ☐ On peut voir cette étape de détachement comme un « free » : le processus libère l'adresse (dans son espace d'adressage) référençant le segment de mémoire partagée (situé en mémoire centrale) car il n'a plus d'accès à y faire

Synchronisation de processus par sémaphores

IPC UNIX System V

- ❑ `#include <sys/sem.h>`
- ❑ Un IPC représente un **ensemble de sémaphores** et non un unique sémaphore !
- ❑ Économie de ressources
 - Regrouper tous les sémaphores utiles à une même application dans **un seul IPC**
- ❑ Un sémaphore dans cet ensemble est identifié par le couple :
(numéro interne de l'ensemble, numéro du sémaphore dans l'ensemble)
 - Le 1^{er} sémaphore de l'ensemble porte le numéro 0

Création d'un ensemble de sémaphores

```
int semget (key_t key, int nsems, int semflg);
```

➤ key = nom externe associé à cet ensemble de sémaphores

- ❖ Unique, obtenu notamment par `ftok()` [voir man], si des processus non parents veulent y accéder
- ❖ Privé, égal à `IPC_PRIVATE`, si l'utilisation de l'ensemble est restreinte aux seuls descendants du créateur

➤ nsems = nombre de sémaphores dans cet ensemble

- ❖ Dépend de la synchronisation à mettre en place pour l'application
 - Exemple : 4 pour la version de base du modèle producteurs/consommateurs

➤ semflg = indicateur, suite de bits comprenant

- ❖ `IPC_CREAT` : création d'un nouvel ensemble de sémaphores
- ❖ `IPC_EXCL` : indique si une éventuelle création doit échouer ou non
 - 1 : la création échoue si le nom externe est déjà utilisé
 - 0 : le processus obtient le numéro interne d'un ensemble déjà créé (ouverture)
- ❖ 9 bits de faible poids spécifiant les droits d'accès à l'ensemble si on le crée

❑ **Retourne l'identificateur interne ou -1 (+ errno) si erreur**

Ouverture d'un ensemble de sémaphores

```
int semget(key_t key, int nsems, int semflg);
```

➤ **key** = nom externe associé à l'ensemble de sémaphores

❖ L'ensemble doit exister

➤ **nsems** = nombre de sémaphores

❖ Peu importe, fixé à la création

➤ **semflg** = indicateur

❖ Droits fixés à la création, doivent permettre à l'appelant d'utiliser l'ensemble

❖ IPC_EXCL et IPC_CREAT ne doivent pas être tous les deux positionnés

❑ **Retourne l'identificateur interne ou -1 (+ errno) si erreur**

Contrôle d'un ensemble de sémaphores

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

- **semid** = identificateur interne de l'ensemble de sémaphores
- **semun** = numéro d'un sémaphore de l'ensemble
- **cmd** = action de contrôle
 - ❖ **SETVAL** : Initialiser la valeur du sémaphore de numéro **semnum** de l'ensemble de sémaphores
 - ❖ **SETALL** : Initialiser les valeurs des différents sémaphores de l'ensemble (la valeur de **semnum** importe peu)
 - ❖ **IPC_RMID** : Détruire l'ensemble de sémaphores
 - ❖ **IPC_STAT**, **IPC_SET**, **GETALL** : cf. man
- **arg** = union de types à définir (*à recopier du man*)

```
union semun {  
    int          val;      /* Type utilisé si SETVAL          */  
    struct semid_ds *buf;   /* Type utilisé si IPC_STAT & IPC_SET */  
    ushort_t     *array;   /* Type utilisé si GETALL & SETALL   */  
}
```

❑ **Retourne >0 si succès (dépend de cmd) ou -1 (+ errno) si échec**

Initialiser les valeurs d'un ensemble de sémaphores

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

☐ Initialiser la valeur d'un sémaphore d'un ensemble

- cmd = SETVAL
- Initialiser la valeur de arg.val avec le nombre de jetons voulu
- arg.val sera affectée à la valeur du sémaphore identifié par (semid, semnum)

☐ Initialiser les valeurs des différents sémaphores d'un ensemble

- cmd = SETALL
- Réserver la place mémoire pour le tableau arg.array (malloc en fonction du nombre de sémaphores de l'ensemble)
- Initialiser ses valeurs (ce doivent être des entiers courts non signés) avec les nombres de jetons voulus (dans l'ordre des sémaphores de l'ensemble)
- Les valeurs de arg.array seront affectées aux valeurs des sémaphores de l'ensemble

Description d'une opération de blocage / déblocage

```
struct sembuf {  
    u_short_t  sem_num; /* Numéro du sémaphore dans l'ensemble */  
    short      sem_op;  /* Opération à réaliser sur ce sémaphore */  
    short      sem_flg; /* Indicateurs */  
}
```

□ Le champ `sem_op` décrit l'opération à réaliser

- `sem_op > 0` : dépôt de `sem_op` jetons supplémentaires dans le sémaphore `sem_num`
jetons éventuellement consommés par des processus bloqués en attente
➔ opération V
- `sem_op < 0` : retrait de `|sem_op|` jetons du sémaphore de numéro `sem_num`
blocage éventuel tant que tous les jetons ne sont pas disponibles
➔ Opération P
- `sem_op = 0` : blocage de l'appelant tant que le nombre de jetons n'est pas nul

```
int semop (int semid, struct sembuf *array, size_t nops);
```

- **semid** = identificateur interne de l'ensemble de sémaphores
 - **array** = tableau dont chacune des nops cases décrit une opération (P ou V) à réaliser sur l'ensemble
 - **nops** = nombre d'opérations (P ou V) à réaliser (sans ressortir du mode noyau)
- ☐ **Retourne 0 si succès, -1 (+ errno) sinon**

Les files de messages Ou boîtes aux lettres IPC UNIX System V

- ☐ **#include <sys/msg.h>**
- ☐ **Permet d'échanger, de manière synchronisée, des messages via une file**
- ☐ **Une file de messages possède une capacité limitée**
 - **Fixée à MSGMNB octets par défaut**
 - **Pouvant être modifiée par une opération de contrôle**
- ☐ **Un processus qui veut « poster » un message est bloqué s'il n'y a pas la place**
- ☐ **Un processus qui veut « retirer » un message d'un certain type est bloqué si aucun message de ce type n'est disponible**
- ☐ **Possibilité de rendre non bloquante, les opérations de dépôts et de retrait (IPC_NOWAIT)**

Création / Ouverture d'une file de messages

```
int msgget (key_t key, int msgflg);
```

➤ key = nom externe associé à cette file

- ❖ Unique, obtenu notamment par `ftok()` [voir man], si des processus non parents veulent y accéder
- ❖ Privé, égal à `IPC_PRIVATE`, si l'utilisation de la boîte à lettres est restreinte aux seuls descendants du créateur

➤ msgflg = indicateur, suite de bits comprenant

- ❖ `IPC_CREAT` : création d'une nouvelle file de messages
- ❖ `IPC_EXCL` : indique si une éventuelle création doit échouer ou non
 - 1 : la création échoue si le nom externe est déjà utilisé
 - 0 : le processus obtient le numéro interne d'une boîte à lettres déjà créée (ouverture)
- ❖ 9 bits de faible poids spécifiant les droits d'accès à la file si on la crée

☐ Retourne l'identificateur interne ou -1 (+ `errno`) si erreur

☐ Pour l'ouverture

- La file de messages doit avoir été créée
- `IPC_EXCL` et `IPC_CREAT` ne doivent pas être tous les deux positionnés

```
int msgctl(int msgid, int cmd, struct msgid_ds *buf);
```

- msgid = identificateur interne de la file de messages
- cmd = action de contrôle
 - ❖ IPC_RMID : Détruire la file de messages
 - ❖ IPC_STAT, IPC_SET, IPC_INFO, MSG_INFO, MSG_STAT : cf. man
- Buf = adresse d'un descripteur de file de messages pour récupérer ou positionner des caractéristiques pour la file identifiée par msgid

❑ Retourne >0 si succès (dépend de cmd) ou -1 (+ errno) si échec

□ Message défini par

- Un pointeur sur le premier octet de la suite (texte du message)
- La longueur du message
- Le type du message...

□ Exemples de structures légales

```
struct msgbuf {  
    long mtype;           /* type du message */  
    char mtext[ ];        /* texte du message */  
}
```

```
struct msgbuf {  
    long mtype;           /* type du message */  
    int n;  
    char mtext[5];  
}
```

```
int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg);
```

- **msgid** = identificateur interne de la file de messages
- **msgp** = adresse du message à envoyer (structure définie par l'utilisateur)
- **msgsz** = taille du message (en octets)
- **msgflg** = indicateur
 - ❖ **IPC_NOWAIT = 0** : Bloquer l'appelant jusqu'à ce que le dépôt soit effectué
 - ❖ **IPC_NOWAIT = 1** : Laisser l'appelant poursuivre son exécution en l'avertissant que son message n'a pas été déposé (errno = EAGAIN)

- ☐ **S'il n'y a assez de place dans la file pour déposer, l'appelant est bloqué (sauf si IPC_NOWAIT positionné) jusqu'à ce qu'il y ait assez de place**
- ☐ **Retourne 0 si succès, -1 (+ errno) sinon**

Retrait d'un message dans une file (1)

```
ssize_t msgrcv(int msgid, void *msgp, size_t msgsz,  
               long msgtyp, int msgflg);
```

- **msgid** = identificateur interne de la file de messages
- **msgp** = adresse du message récupéré
- **msgsz** = taille maximale du message attendu (en octets)
- **msgtyp** = type du message attendu
 - ❖ 0 : premier message de la file
 - ❖ > 0 : premier message de la file du type msgtyp
 - ❖ < 0 : premier message de la file d'un type inférieur ou égal à msgtyp
- **msgflg** = indicateur
 - ❖ IPC_NOWAIT : non bloquant si pas de message du bon type disponible
 - ❖ MSG_EXCEPT : Premier message qui diffère de msgtyp si msgtyp > 0
 - ❖ MSG_NOERROR : Tronquer le message s'il est plus long que msgsz

```
ssize_t msgrcv(int msgid, void *msgp, size_t msgsz,  
               long msgtyp, int msgflg);
```

- ❑ S'il n'y a assez de message du bon type disponible, l'appelant est bloqué jusqu'à ce qu'un message puisse être retiré
 - En tenant compte des indicateurs positionnés dans msgflg
- ❑ Retourne la taille effective du message retiré si succès, -1 (+ errno) sinon
- ❑ Remarque
 - Si on récupère le message dans une variable du type struct msgbuf *
 - ❖ Le champ mtype contient le type du message
 - ❖ Le champ mtext contient le texte du message