

HMIN105M : Principes de la programmation concurrente et répartie

Responsable : Hinde Bouziane (bouziane@lirmm.fr)
Intervenants : H. Bouziane et T. Lorieul

UM - LIRMM

1 Chapitre 2 : Communications Évoluées entre Processus (IPC)

- Généralités
- Files de Messages
- Mémoires Partagées
- Ensembles de sémaphores

Besoins

- On veut des moyens de communications offrant d'**autres possibilités** que les tubes ou les sockets :
 - Échange de messages, partage d'espaces de mémoire communs, synchronisation.
- Les connus : communications dites **IPC - SV** :
 - **Files de messages** : envoi/réception de messages entre plusieurs processus ; l'unité transmise entre processus est un message !
 - **Mémoires partagées** : mémoire commune accessible à plusieurs processus, donc hors de l'espace de chacun !
 - Ne pas confondre avec le partage d'un espace mémoire d'un seul processus par plusieurs threads.
 - **Ensembles de sémaphores** : outils et opérations évolués pour résoudre les conflits d'accès et la synchronisation.

Caractéristiques globales

- Mécanismes externes aux processus
- Gestion par le système d'exploitation (table dédiée).
- Chaque objet (file, ensemble de sémaphore ou segment de mémoire partagée) dispose d'un identifiant ($id \geq 0$) interne à un processus.
 - Analogie : descripteur de fichier pour un tube ou une socket.
 - Nécessaire pour son utilisation par les processus.
 - Question : comment obtenir l'identifiant interne ?
- De l'extérieur, identification par mécanisme de **clé**. Cette clé permet à un processus l'obtention d'un identifiant interne.

Visualisation - exemple

Par la commande `ipcs` on peut obtenir ce tableau :

----- Shared Memory Segments -----

key	shmid	owner	perms	bytes	nattch	status
0x00000000	131072	jms	600	393216	2	
0x00000000	163841	jms	600	393216	2	dest

----- Semaphore Arrays -----

key	semid	owner	perms	nsems	status
0xcbc384f8	0	jms	600	1	

----- Message Queues -----

key	msqid	owner	perms	used-bytes	messages
0x7a094087	32768	jms	666	160	20

Une file de messages dont l'identifiant est 32768 existe, avec le propriétaire et droits indiqués et elle contient actuellement 20 messages de longueur totale 160 octets.

Identification

Un objet IPC peut-être privé ou publique. Lorsqu'il est publique, il faut donner à tous les processus autorisés à y accéder un moyen d'obtenir l'identifiant, c.à.d. leur fournir une **clé**.

Une clé est une suite binaire permettant **indirectement** d'obtenir l'identifiant d'un objet IPC.

Comment ? par un calcul.

Calcul sur quoi ? sur des paramètres convenus à l'avance entre le créateur de la file et l'ensemble des utilisateurs.

Une fonction dédiée est `ftok()`. Elle a pour syntaxe :

```
key_t uneClef=ftok(const char * chemin, int entier)
```

Identification - suite

Le paramètre `chemin` est le chemin d'un fichier, et `entier` un entier quelconque (un caractère pour certains systèmes).

Ces deux paramètres seront toujours utilisés pour obtenir l'identifiant d'un objet IPC publique, `ftok()` faisant un calcul sur l'*inode* du fichier et l'entier.

Concrètement, pour les programmeurs des processus il faut :

- décider d'un nom de fichier ; le créer et **ne plus le toucher** tant que la file existe.
- décider d'un entier déterminé à utiliser pour un objet IPC déterminé.

Identification - on termine

Exemple 1 : Tous les processus utilisateurs font :

```
key_t sesame = ftok("./readme.txt", 10)           et ensuite  
int id_obj = ? (sesame, ....)                     qui donnera
```

systématiquement le même résultat pour la même clé.

Plus tard, le symbole **?** sera remplacé par un nom de fonction qui dépend de l'objet IPC manipulé.

Exemple 2 : Tous les processus utilisent un logiciel commun /opt/jeux/solitaire. On peut utiliser la clé `ftok("/opt/jeux/solitaire",'A')`.

1 Chapitre 2 : Communications Évoluées entre Processus (IPC)

- Généralités
- **Files de Messages**
- Mémoires Partagées
- Ensembles de sémaphores

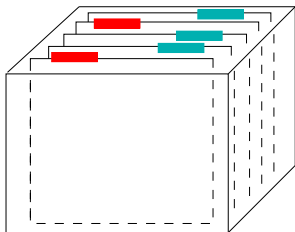
Files de messages - concepts

Une **file de messages** est une structure en mémoire centrale, faite pour communiquer entre processus, l'unité d'échange étant un **message**.

Un **message** est une structure de données quelconque **sans** pointeurs.

Question : pourquoi ?

Les messages peuvent porter une **étiquette**.



Analogies :

file de messages \equiv boîte contenant des fiches

un message \equiv une fiche

une étiquette \equiv un onglet.

Actions possibles

Chaque processus peut :

- déposer un message,
- extraire un message de plusieurs façons : le premier disponible ou le premier portant une étiquette spécifique ; par exemple le premier message portant une étiquette rouge.

et aussi :

- créer une file, lui affecter des droits d'accès,
- utiliser une file existante (si possible),
- la détruire.

Remarques :

- Un message extrait disparaît de la file.
- Il n'y a pas de notion d'ouverture/fermeture.
- La durée de vie de la file va de sa création jusqu'à la destruction, donc au delà de la vie des processus accédant.

Une mémoire faillible

Question : lorsque la file n'est pas vide et que tous les processus accédant se terminent, est-ce que la file conserve son contenu ?

Réponse : oui, . . . tant que la mémoire système n'a pas été nettoyée, par une action de redémarrage du système ou par celle d'un administrateur ou de l'utilisateur créateur de la file.

Conséquence : Les objets créés sont *persistants* : leur existence est indépendante des processus.

Synchronisation

Dans les files de messages, la protection et la synchronisation d'accès sont prises en charge par le système.

Ce que le système prend en charge :

- chaque message est déposé de façon atomique, c'est-à-dire sans mélange possible entre messages ; lorsque le système dépose un message, il termine le dépôt avant de passer au dépôt suivant.
- Si la file est pleine, tout processus voulant déposer un message (on parlera d'écrivain) est endormi ; le réveil aura lieu dès qu'il y aura de la place pour déposer un message.
- Si la file est vide, tout processus voulant extraire un message (on parlera de lecteur) est endormi ; le réveil aura lieu dès qu'un message **correspondant à sa requête** sera déposé.

Attention

Attention à la possibilité de famine, l'attente éternelle n'est pas prise en charge par le système.

Exemple : Un processus voulant extraire un message portant une étiquette rouge, sera réveillé uniquement lorsqu'un tel message sera déposé, quel que soit le contenu de la file par ailleurs.

Remarque : Des solutions d'extraction non bloquante sont possibles (non ou rarement utilisées dans ce cours). Elles sont utilisables en fonction du contexte mais attention car elles peuvent être contre-productives. **Question** : Donner un exemple.

Opérations sur les files

Les opérations réalisables sur les files sont :

- 1 la création d'une file ;
- 2 l'identification d'une file existante ;
- 3 le dépôt et l'extraction d'un message ;
- 4 la destruction d'une file ;
- 5 On peut aussi connaître et gérer plusieurs paramètres d'une file (droits, taille et nombre de messages, etc).

Remarque : Dès qu'une file existe, son identifiant est fixé pour toute sa durée de vie ! Tout processus voulant y accéder **doit** le connaître ou l'obtenir.

Création et identification d'une file

Le même appel système, *msgget()*, permet de créer une file ou uniquement d'obtenir son identifiant.

Syntaxe :

int	msgget	(key_t uneClef,	int droits)
↑		↑	↑
identifiant		clé attachée à la file	droits attachés à la file ou accès demandé

Les droits s'énoncent comme pour la création de fichiers.

Exemples

```
int f_id = msgget(cle, IPC_CREAT|0666)
```

permet de créer une file de messages avec les droits d'accès de lecture et écriture à tout processus de tout utilisateur.

```
int f_id = msgget(cle, O_RDONLY)
```

est une demande d'accès en lecture seule.

Création et identification d'une file - file privée

Un processus voulant créer une file privée peut le faire par le truchement de la constante `IPC_PRIVATE` :

Par exemple :

```
int f_id = msgget(IPC_PRIVATE, 0666)
```

permet de créer une file de messages privée avec les droits d'accès indiqués.

Privée ne veut pas dire que seul le processus créateur peut utiliser la file. Pour partager la file par plusieurs processus, ces derniers doivent obtenir directement son identifiant.

Moyens possibles : du bouche à oreille (très moyen), en consultant la liste des objets ipc existants, par héritage, à l'aide d'un moyen de communication : tubes, files publiques, mémoire partagée, etc.

Structure d'un message

La structure du message est décrite ainsi dans le manuel :

```
struct msgbuf {  
    long mtype;      /* message type, must be > 0 */  
    char mtext[1]; /* message data */  
};
```

Interprétation : Une structure contenant l'étiquette comme première variable, suivie de variables dont la taille globale est > 0 . Ces variables ne doivent pas contenir de pointeurs.

Question : Pourquoi ?

Conclusion : on peut déposer toute structure complexe, sans pointeurs.

Exemple de message

```
struct strMonMsg {  
    long monetiquette ;  
    int num[10] ;  
    char nom[30] ;  
} ;  
  
...  
struct strMonMsg monMsg, *ptrContenu ;
```

et déposer/extraire de tels messages de la file. Le contenu qui suit l'étiquette peut aussi être une `struct`.

Accès - extraction

Ce qu'on veut faire :

```
extraire(idFile, tamponRécupération, uneEtiquette)
```

Concrètement, l'appel système est **msgrcv()**, de syntaxe :

ssize_t	msgrcv(
	int identifiant,	⇐ résultat de msgget()
	struct msgbuf *ptrmsg,	⇐ pointeur tampon réception
	size_t lgmsg,	⇐ longueur max acceptée
	long étiquette,	⇐ quelle étiquette
	int flags)	⇐ 0 pour l'instant

- Le résultat est le nombre d'octets lus hors étiquette.
- étiquette > 0 : lecture du premier message disponible avec l'étiquette e = étiquette.
- étiquette = 0 : lecture du premier message disponible.
- étiquette < 0 : lecture premier message disponible avec la plus petite étiquette $e \leq |\text{étiquette}|$.

Exemple

```
struct sMsg {long etiq; char mot[12];} vmsg;
```

```
int ret = msgrcv(f_id, &vMsg,  
    (size_t)sizeof(vMsg.mot), (long) monPid, 0);
```

demande à extraire de la file dont l'identifiant est `f_id`, le premier message portant l'étiquette `monPid`, et de copier ce message dans `vMsg`.

Rappel : le message va disparaître de la file.

Accès - dépôt

Ce qu'on veut faire :

```
déposer(idFile, message, uneEtiquette (utile?))
```

Concrètement, l'appel système est **msgsnd()**, de syntaxe :

```
int msgsnd(  
    int identifiant,           ⇐ résultat de msgget()  
    struct msgbuf *ptrmsg,     ⇐ pointeur sur tampon à déposer  
    size_t lgmsg,             ⇐ longueur du message  
    int flags)                 ⇐ 0 pour l'instant
```

- L'étiquette est absente de cet appel, car elle fait partie de la structure `msgbuf` et le message est déposé avec cette étiquette.
- Le résultat indique si l'opération a réussi ou échoué.
- Attention, une valeur négative ou nulle pour l'étiquette, est forcément une erreur ! Penser au lecteur pour s'en convaincre.

Suppression d'une file

La suppression d'une file peut se faire par la commande `ipcrm`, ou par l'appel système :

```
int msgctl(int f_id, int op,  
           .../*struct msqid_ds *entreeTable*/).
```

L'appel système est en fait très général et permet de gérer tous les paramètres de la file. On se contente ici de donner la forme permettant la suppression seule :

```
int res = msgctl(  
    identifiant,  ⇐ résultat de msgget()  
    IPC_RMID,    ⇐ constante pour la destruction  
    NULL)        ⇐ pointeur si gestion de paramètres
```

1 Chapitre 2 : Communications Évoluées entre Processus (IPC)

- Généralités
- Files de Messages
- **Mémoires Partagées**
- Ensembles de sémaphores

Principe

- Jusque là, chaque processus dispose de son propre espace mémoire, protégé inaccessible à tout autre processus.
- La communication consiste à transférer (copier) des données, dans un espace géré et synchronisé par le système (tubes, fichiers, files de messages, etc.).
- Une **mémoire partagée** consiste à disposer d'un espace **commun** de mémoire, accessible à plusieurs processus.
- Chaque processus pourra y « travailler » comme sur toute donnée propre.
- Avec restrictions possibles : certains processus pourront lire et écrire, d'autres ne pourront que lire ou n'auront aucun droit d'accès.

Problèmes

- Cet espace ne peut pas faire partie de l'espace d'un des processus accédant. En effet, un processus doit rester maître de son propre espace, il peut s'arrêter et un autre arriver et partager l'accès.

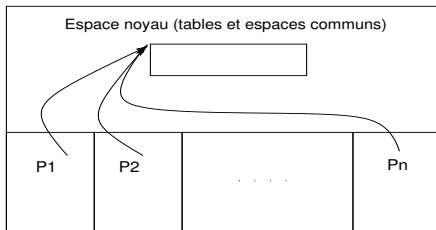
Questions : où faut-il localiser cet espace et comment y autoriser l'accès ? Qui peut le créer et quelle sera sa durée de vie ?

- Le système d'exploitation ne gère pas la synchronisation quand plusieurs processus accèdent en lecture et écriture.

Question : qui gère la synchronisation et par quels moyens ?

Caractéristiques - localisation

- Localisation dans l'espace alloué « au système ».



- L'espace alloué (on parlera de **segment**) sera persistant : son existence sera indépendante des processus qui y accèdent.

Caractéristiques - accès et synchronisation

- Un espace, ou segment, de mémoire partagée sera créé par un processus.
- Chaque processus voulant y accéder demandera à s'**attacher** l'espace ; après vérification des droits, il disposera d'un pointeur vers cet espace.
- Les processus accédant devront gérer la synchronisation : exclusion et protection. Classiquement, ils utiliseront des *sémaphores*.
- La destruction de l'espace devra être faite par un processus ayant le droit de destruction. En cas d'arrêt du système, l'espace sera perdu : fonctionnement identique à celui des files de message.

Opérations sur un segment de mémoire partagée

Les opérations réalisables sur un espace mémoire partagée :

- ❶ création d'un segment ;
- ❷ demande d'attachement (obtention d'un pointeur) ;
- ❸ détachement (abandon d'accès) ;
- ❹ contrôle des paramètres dont suppression (comme pour les files de messages).

Remarque : l'accès en lecture/écriture se fait de manière classique, en utilisant un pointeur. Il n'y a donc pas de primitives dédiées.

Création et identification d'un segment

Comme pour les files de messages, le même appel système, *shmget()*, permet de créer un segment ou uniquement d'obtenir son identifiant.

Syntaxe :

int	shmget	(key_t uneClef,	size_t taille,	int optEtDroits)
↑↑		↑↑	↑↑	↑↑
identifiant		clé associée au segment	taille demandée	droits et/ou type accès

- Le principe d'obtention de la clef est celui déjà vu avec `ftok()`.
- Les droits s'énoncent comme pour la création de fichiers.
- La taille est arrondie au multiple supérieur de la taille d'une page. Lorsque le segment existe, on demande une taille inférieure ou égale (0 est une bonne solution)

Exemples

```
struct uneChaine{  char c;  
                   int x, y;  
                   struct uneChaine *suiv;  
                   };  
  
int  sh_id=shmget(  sesame,  
                  size_t(30*sizeof(unChaine)),  
                  IPC_CREAT|0666);
```

permet de créer un segment avec les droits d'accès de lecture et écriture à tout processus de tout utilisateur.

```
int  sh_id=shmget(sesame,size_t(0),O_RDONLY);
```

est une demande d'accès en lecture seule, en supposant que le segment existe.

Demande d'accès : attachement

Pour accéder à un espace de mémoire partagé, un processus demande l'*attachement* de cet espace ; il consiste à obtenir un pointeur dans son espace propre, vers cet espace extérieur.



syntaxe :

<code>void *</code>	<code>shmat</code>	<code>(int idMem,</code>	<code>const void * adrForce,</code>	<code>int options)</code>
↑↑		↑↑	↑↑	↑↑
adresse ou (void *) -1		identifiant obtenu	NULL sauf exception	type accès

- `adrForce` permet, s'il est différent de `NULL` de spécifier une adresse résultat choisie (forcée) par l'utilisateur : rare.
- Le type d'accès par défaut est en lecture et écriture. `options` permet de le modifier, par exemple de demander l'accès en lecture seule avec `SHM_RDONLY`.

Abandon - détachement

On abandonne l'accès en détachant l'espace commun. **syntaxe** :

int	shmdt	(const void * adrAtt)
		
0 : réussite		adresse
-1 : échec		d'attachement

- À la fin du processus, tous les segments préalablement attachés, dans ce processus, sont détachés.
- **Question** : Pourquoi faut-il donner une adresse d'attachement pour détacher et non l'identifiant ?

Exemples

Si on a créé un tableau d'entiers :

```
int * tab;  
if((tab = (int *)shmat(idMem, NULL, 0))==(int *)-1){  
    perror("shmat");  
    //suite ...}
```

Ou la structure vue précédemment :

```
struct uneChaine * p_att;  
p_att = (struct uneChaine *)shmat(idMem, NULL, 0);  
if ((void *)p_att == (void *)-1){  
    perror("shmat");  
    //suite ...}
```

Détachement :

```
int dtres = shmdt((void *)p_att);
```

Suppression

La suppression est similaire à celle des files de messages :

```
int shmctl(  
    int identifiant,    ⇐ résultat de shmget()  
    IPC_RMID,          ⇐ constante pour la destruction  
    NULL)              ⇐ pointeur si gestion de paramètres
```

Mais encore :

La syntaxe complète de `shmctl()` ou `msgctl()` permet en fait de récupérer un pointeur sur une structure contenant les caractéristiques du segment ou de la file.

On pourra regarder dans le manuel comment récupérer les caractéristiques courantes ou modifier celles qu'on peut modifier.

Pour terminer

En utilisant des segments de mémoire partagée, on travaille directement dans les segments, sans recopie de données de l'espace du processus vers ce segment.

- Dans le cas de partage de gros volumes de données, c'est une structure bien adaptée.
- Qui plus est, le partage peut se faire entre processus non issus d'un même parent par `fork()`.

Question ouverte : comme la taille d'un segment est arrondie au multiple supérieur d'une page, peut-on utiliser l'espace dans la dernière page, au delà de la demande du processus créateur ?

1 Chapitre 2 : Communications Évoluées entre Processus (IPC)

- Généralités
- Files de Messages
- Mémoires Partagées
- Ensembles de sémaphores

Rappels

Un sémaphore est un mécanisme de synchronisation de processus. Il s'agit d'une structure de données qui comprend :

- un entier non négatif donnant le nombre de ressources disponibles
- une file d'attente de processus

Et manipulée au travers de trois opérations :

- Init (sémaphore sem, int nombre_de_ressources)
- P(sémaphore sem, int nb_ressources) : bloque l'appelant si le nombre de ressources de sem demandées est supérieur au nombre de ressources disponibles, sinon décrémente le nombre de ressources de sem.
- V(sémaphore sem, int nb_ressources) : libère un nombre de ressources obtenues et débloquent un ou des processus en attente s'il en existe.

Sémaphores SV

L'implantation SV des sémaphores permet de gérer un ensemble de sémaphores, de sorte à pouvoir contrôler

k_1	exemplaires de la ressource	R_1
\dots	\dots	\dots
k_i	exemplaires de la ressource	R_i

avec des opérations (pour chaque sémaphore de l'ensemble)

P_n	qui bloque l'appelant si la valeur du sémaphore est inférieure à $ n $
V_n	qui incrémente la valeur du sémaphore de $ n $ et débloquent les attentes
Z	qui attend que le sémaphore soit nul afin de réaliser des rendez-vous

et possibilité de réaliser plusieurs opérations P_n et V_n atomiquement.

Création

La création ressemble à celle des files de messages et mémoires partagées.

int	semget	(key_t uneClef,	int nbSem,	int opt)
↑		↑	↑	↑
identifiant		clé associée à l'ensemble	nombre sémaphores	droits et options

Permet de créer un **tableau** de *nbSem* sémaphores ou de récupérer l'identifiant d'un tableau existant. Attention, L'objet IPC ici est le tableau. Il s'agit d'un « vrai » tableau C.

Exemple :

```
int idSem = semget(cleSem, 1, IPC_CREAT|0666);
```

crée et/ou récupère l'identifiant d'un tableau à un seul sémaphore, associé à *cleSem*.

Opérations

On souhaite réaliser une combinaison d'opérations P , V et Z sur un (sous-)ensemble de sémaphores.

Concrètement, on utilise la fonction :

```
int semop(  
    int idSem,                ⇐ résultat de semget()  
    struct sembuf *tabOp,     ⇐ ensemble d'opérations  
                                à réaliser  
    int nbOp)                 ⇐ nombre d'opérations  
                                dans ce tableau
```

Le résultat est 0 (réussite) ou -1 (échec).

Où, toute opération (P , V ou Z) sur un sémaphore est décrite par une structure `sembuf` et est propre à ce sémaphore. L'**ensemble** des *nbOp* opérations demandé sera réalisé de façon atomique.

Opérations - suite

```
struct sembuf {  
    unsigned short  sem_num; /* Numéro du sémaphore */  
    short          sem_op;   /* Opération sur le sémaphore */  
    short          sem_flg;  /* Options par exemple SEM_UNDO */  
};
```

- Les numéros commencent à 0.
- La valeur n de `sem_op` détermine l'opération
 - si $n < 0$ l'opération est P avec comme valeur $|n|$: tentative de décrémenter le sémaphore numéro `sem_num` de $|n|$;
 - si $n > 0$ l'opération est V : incrémentation de n avec réveil des processus en attente ;
 - si $n = 0$ l'opération est Z : attente que la valeur du sémaphore soit 0 (voir rendez-vous).

Exemples

Pour un sémaphore unique (à la Dijkstra), on peut définir :

Une opération P :

```
struct sembuf opp;
opp.sem_num=0;
opp.sem_op=-1;
opp.sem_flg=SEM_UNDO;
semop(idSem, &opp, 1);
```

Une opération V :

```
struct sembuf opv;
opv.sem_num=0;
opv.sem_op=+1;
opv.sem_flg=SEM_UNDO;
semop(idSem, &opv, 1);
```

Ou encore :

```
struct sembuf op[]={
    {(u_short)0, (short)-1, SEM_UNDO},
    {(u_short)0, (short)+1, SEM_UNDO}  };
```

puis : semop(idSem, op, 1) **pour** P,
et semop(idSem, op+1, 1) **pour** V.

Initialisation

C'est la primitive système `semctl()` qui est utilisée pour l'initialisation d'un ensemble de sémaphores, toujours atomiquement.

Remarque : c'est cette même primitive qui sera utilisée pour détruire un ensemble de sémaphores ou obtenir des informations sur ces derniers.

Le prototype est défini ainsi :

```
int semctl(int semid, int semnum, int cmd, ...);
```

Interprétation : on veut faire telle commande sur le sémaphore numéro `semnum`, de l'ensemble `semid`. Pour les pointillés, le manuel dit ceci :

La fonction a trois ou quatre arguments, selon la valeur de `cmd`. Quand il y en a quatre, le quatrième est de type union `semun`. Le programme appelant doit définir cette union de la façon suivante :

Initialisation - suite du calvaire

```
union  semun {  
    int val ;                               /* cmd = SETVAL */  
    struct semid_ds *buf ;                 /* cmd = IPC_STAT ou IPC_SET */  
    unsigned short *array ;               /* cmd = GETALL ou SETALL */  
    struct seminfo *_buf ;                 /* cmd = IPC_INFO (sous Linux) */  
};
```

Conséquences :

- Il faut réapprendre ce qu'est une structure de type *union*
- Dédurre qu'on peut certainement faire beaucoup d'opérations intéressantes.

Exemple - initialisation simple d'un sémaphore

En supposant qu'on a déclaré dans le programme une `union semun` comme celle décrite, on peut initialiser un sémaphore à 1 comme suit :

```
semun egCtrl;  
egCtrl.val=1;  
if(semctl(idSem, 0, SETVAL, egCtrl) == -1){  
    perror("'problème init'");  
    //suite  
}
```

Revoir la structure `semun` : on peut initialiser de façon atomique un tableau de sémaphores (*semnum* devient le nombre d'éléments), obtenir des valeurs courantes ou encore gérer des caractéristiques relatives à l'ensemble de sémaphores.

Destruction

Enfin, la **destruction** d'un ensemble se fera classiquement avec l'appel :

```
semctl(idSem, 0, IPC_RMID)
```

Elle réveillera tous les processus en attente, s'il en existe.

A retenir (chapitres 1 et 2)

- Différences entre processus et threads.
- Partage de ressources inter-processus et inter-threads.
- Synchronisation entre processus et threads (exclusion mutuelle, attente d'un événement).
- Faire attention aux problèmes liés à la synchronisation, en particulier les situations d'interblocage. Exemple : ne jamais effectuer un blocage dans une section critique sans libérer la section critique.
- Conseils de programmation :
 - bien initialiser les objets/variables utilisés,
 - terminaison "propre" : libération de l'espace mémoire alloué, nettoyage des tables IPC, terminaison des threads, etc,
 - traitement des retours de fonction et gestion des erreurs,
 - etc.