Multitâches

3 septembre 2007

Table des matières

1 Introduction **2**
1.1 Pourquoi un système d'exploitation multi-tâches ? 2
1.2 Du multi-utilisateurs 3
1.3 Exemple de multitâche applicatif : pile TCP/IP et paramétrage 3
1.4 Du multi-tâches mono application : calculateur sous VxWorks 4

2 Concepts de base du multi-tâches **6**
2.1 Définition d'une tâche 6
2.1.1 Les composantes d'une tâche 6
2.1.2 Multitâches, et programme réentrant 7
2.2 Etats d'une tâche dans VxWorks 7
2.2.1 Symboles utilisés dans WindView 8
2.2.2 Etats, et groupements d'états SCEPTRE 8
2.3 Ordonnancement (scheduling) 9
2.3.1 Systèmes préemptifs (preemptive scheduling), coopératifs (no preemptive scheduling) 9
2.3.2 Algorithme d'ordonnancement par priorité (algorithme par défaut dans VxWorks) 10
2.3.3 Exercice : Exploitation d'un suivi WindView 11
2.3.4 Etude du résultat obtenu 11
2.3.5 Algorithme d'ordonnancement en tourniquet (round robin) 11
2.3.6 Le tourniquet avec priorités (algorithme possible d'ordonnancement du noyau Wind) 11
2.3.7 Exercice : exploitation d'un suivi WindView 12
2.3.8 Influence du blocage d'une tâche sur la charge du processeur 13
2.3.9 La tranche de temps, et le rendement 13
2.4 Portabilité : les primitives POSIX 13
2.5 Multi-tâches ou multi-threads ? 13

3 Exclusion entre processus **14**
3.1 Les différentes formes d'implémentation de l'exclusion 14
3.1.1 Verrouillage des IT 14
3.1.2 Verrouillage de la préemption 14
3.1.3 Sémaphores d'exclusion mutuelle (voir 4.3.6 page 33) 14
3.1.4 Traces typiques WindView 14
3.2 Pourquoi chercher à éviter les sections critiques ? 15
3.3 Comment éviter les sections critiques ? 16
3.4 Comment identifier une ressource critique ? 16

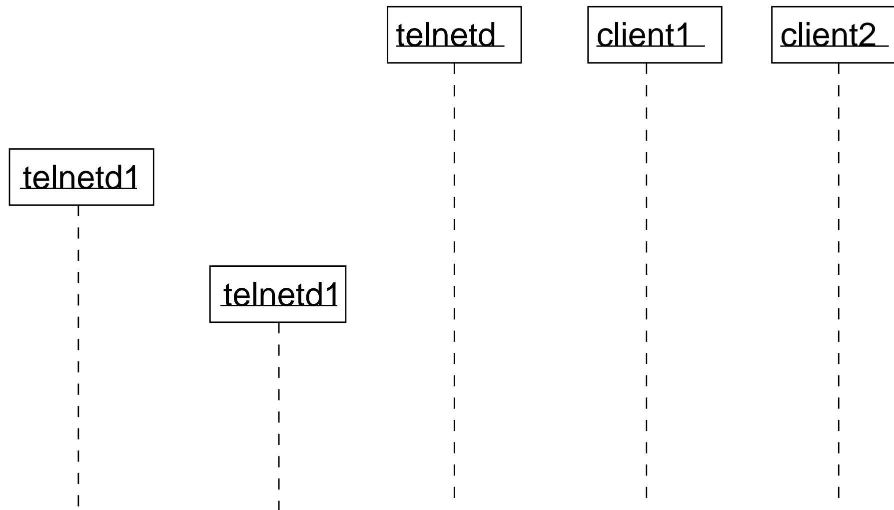
4 Communication inter-processus (IPC) 17
4.1 Le tube de communication (pipe) : un périphérique virtuel . . . 18
4.1.1 Caractéristiques universelles d'un tube de communication . . . 18
4.1.2 Les tubes nommés dans VxWorks . . . 18
4.1.3 Cas d'utilisation des tubes `VxWorks` . . . 18
4.1.4 Les tubes nommés dans Unix . . . 19
4.2 Les files de messages . . . 20
4.2.1 Files de messages Unix . . . 20
4.2.2 Les files de messages VxWorks . . . 24
4.2.3 Symbolisation WindView des événements associés aux files de messages . . 25
4.3 Les sémaphores . . . 26
4.3.1 Sémaphore booléen d'usage général . . . 26
4.3.2 Sémaphores de synchronisation . . . 28
4.3.3 Suivi WindView . . . 30
4.3.4 Exemple de sémaphore de synchronisation POSIX (linux pthread) . . . 31
4.3.5 Exemple de synchronisation sur la mort du premier enfant . . . 32
4.3.6 Sémaphores d'exclusion mutuelle dans VxWorks . . . 33
4.3.7 Sémaphores à compte . . . 35
4.3.8 Options spéciales des sémaphores dans le noyau WIND . . . 35
4.3.9 Comparaison entre les sémaphores POSIX, et WIND . . . 36
4.4 La mémoire partagée . . . 37
4.4.1 Mémoire partagée dans un environnement de processus lourds (shared memory) . . . 37
4.4.2 Mémoire partagée dans un environnement de processus légers (thread) . . . 37
4.5 Les signaux . . . 38
4.5.1 Fonctionnement général des signaux . . . 38
4.5.2 Les signaux prédéfinis (`kill -l`) . . . 38
4.5.3 Un cas particulier : les alarmes . . . 39
4.5.4 Timers et Chiens de garde dans VxWorks . . . 40
4.5.5 L'horloge auxiliaire dans VxWorks . . . 40
4.5.6 POSIX Clocks and Timers . . . 41

1 Introduction

1.1 Pourquoi un système d'exploitation multi-tâches ?

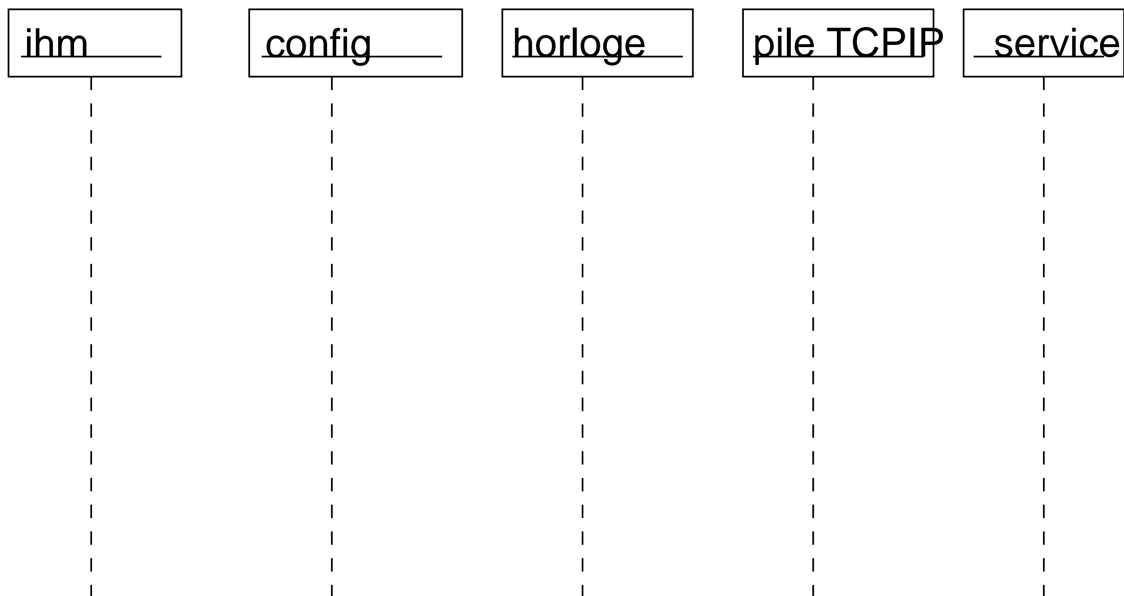
- La machine est un serveur d'applications, sur lequel plusieurs applications peuvent tourner ensemble, ou plusieurs utilisateurs utilisent simultanément une même application (réentrante) . Ces applications sont indépendantes, et n'ont pas d'objectif commun (telnetd, sshd, bash, g++ ...) -> applications
- La machine est un serveur multiservices (serveur Web, serveur telnet, serveur ftp, ...)
- La machine a été dimensionnée pour l'application, qui se compose de plusieurs activités simultanées, conçues pour un objectif unique (activités)
- Combien de processeurs ? combien de temps perdu ?
- Facilité de développement, de maintenance.

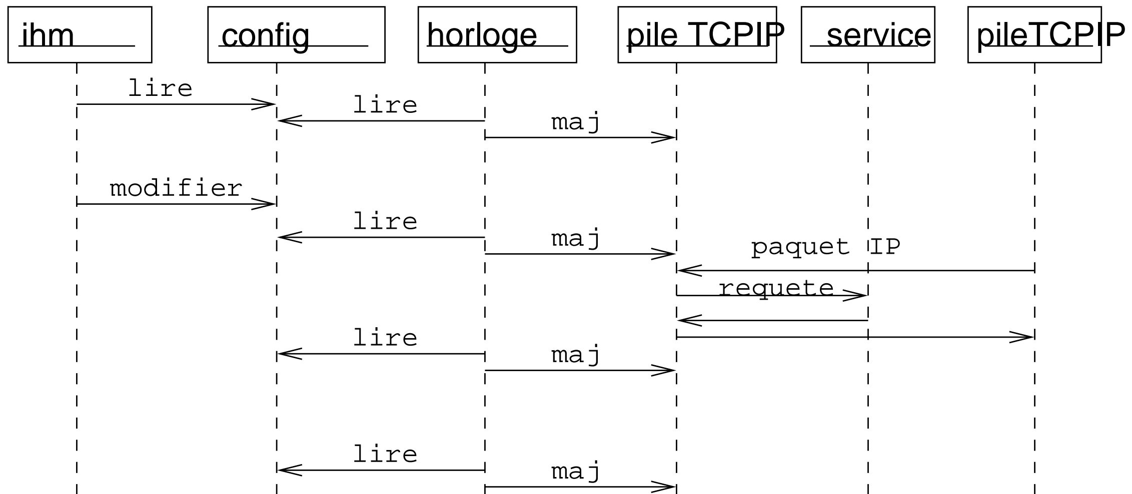
1.2 Du multi-utilisateurs



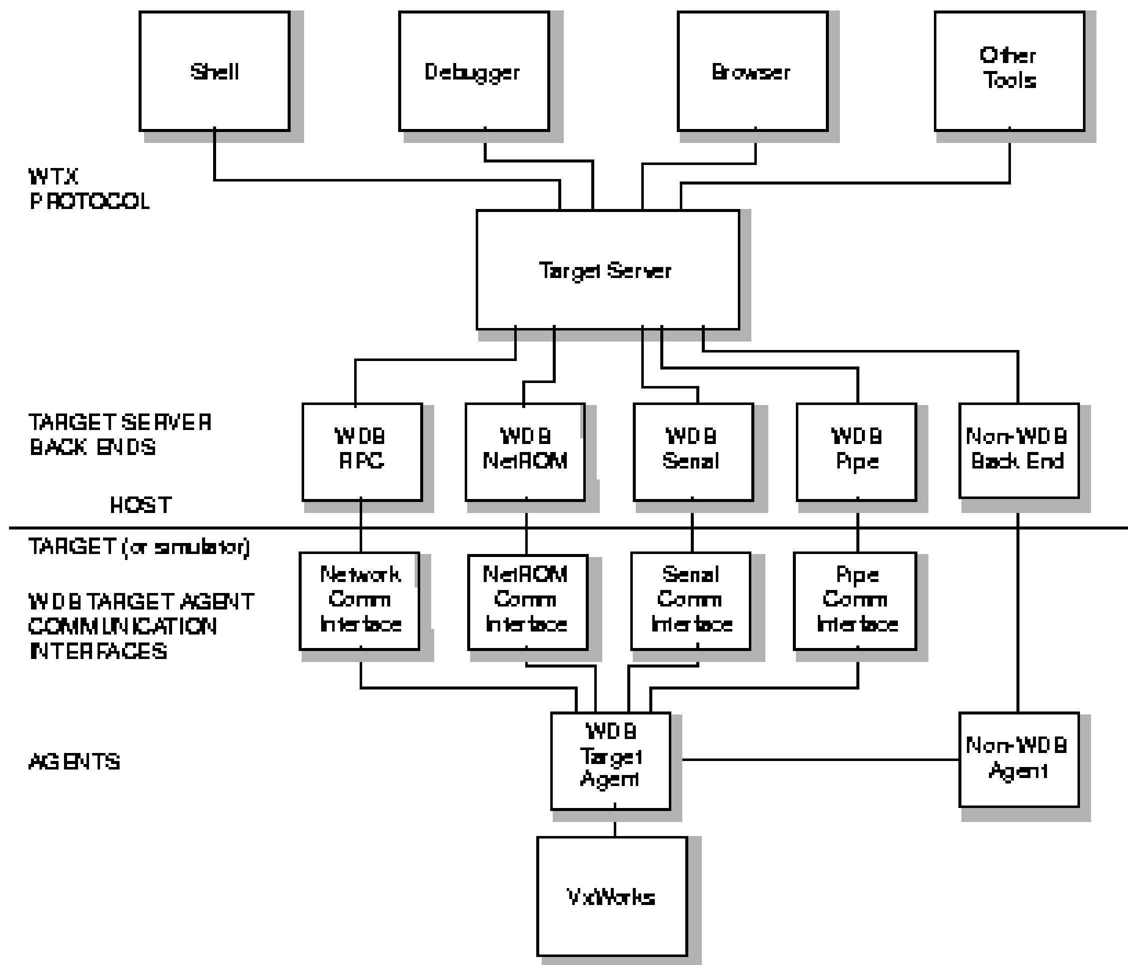
1.3 Exemple de multitâche applicatif : pile TCP/IP et paramétrage

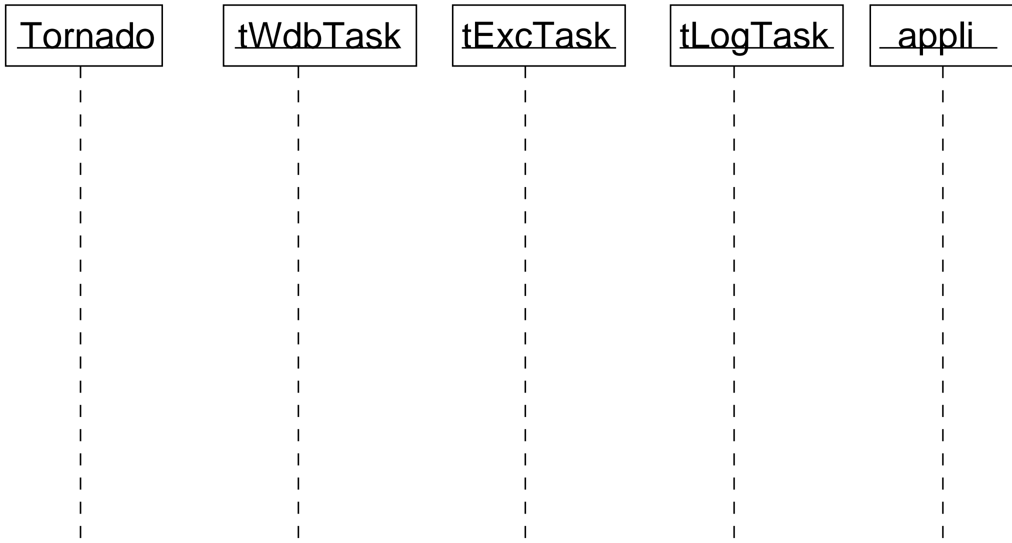
Sous Win2K, comme sous Linux, on peut paramétrer la pile TCP/IP dynamiquement





1.4 Du multi-tâches mono application : calculateur sous VxWorks




Tornado
tWdbTask
tExcTask
tLogTask
appli

2 Concepts de base du multi-tâches

2.1 Définition d'une tâche

Une tâche (task), contrairement à un programme, est une entité essentiellement .
Elle met en œuvre de manière séquentielle un ou plusieurs programmes (le code), en vue de la réalisation d'une activité par un processeur.

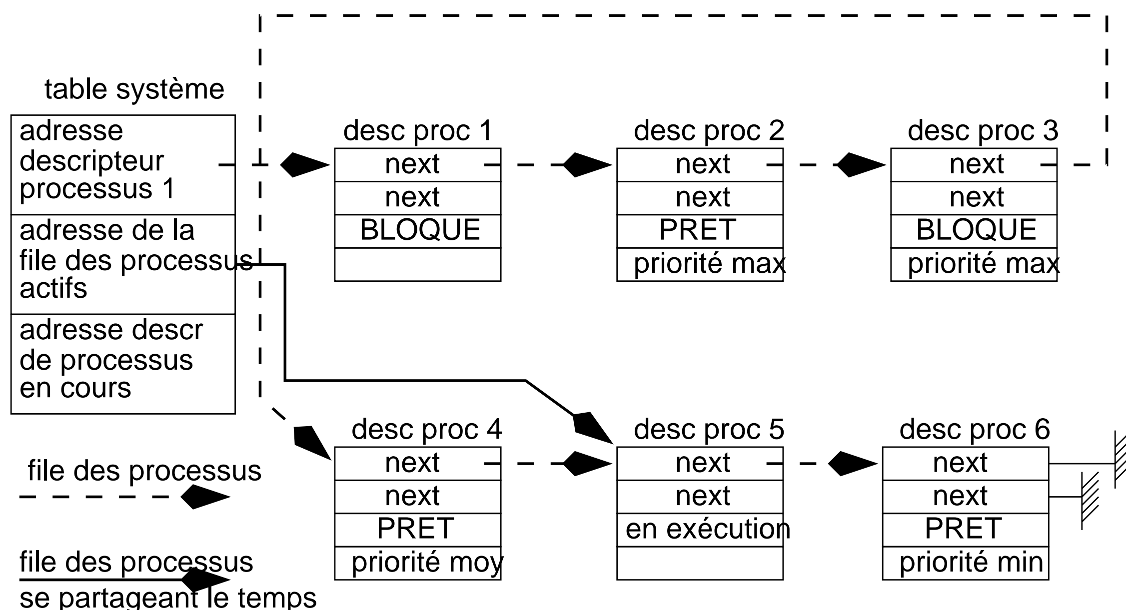
Les tâches soumises à de fortes contraintes de temps sont liées à des interruptions matérielles (elles sont activées par un événement physique) : elles sont appelées tâches matérielles ou .

Les tâches qui ont moins d'exigence en temps de réponse sont appelées tâches (ou différées, car elles ne s'exécutent pas immédiatement dès qu'elles sont actives). Elles sont programmées soit dans une boucle de scrutation, soit de façon indépendante.

2.1.1 Les composantes d'une tâche

Une tâche a un état qui varie dans le temps, en fonction des instructions du programme qu'elle exécute, et des événements extérieurs auxquels elle est sensible. Elle comporte différentes zones mémoires :

- une zone (pointée par le pointeur de programme)
- une zone de (pointée par le pointeur de pile)
- une zone de données (pointée par le pointeur de données)
- un descripteur de processus, élément d'une chaîne, contenant les valeurs de tous ces pointeurs, les valeurs des registres du processeur, un nom, un identifiant, une priorité, un état, ... Tous ces éléments faisant partie de ce qu'on appelle le .



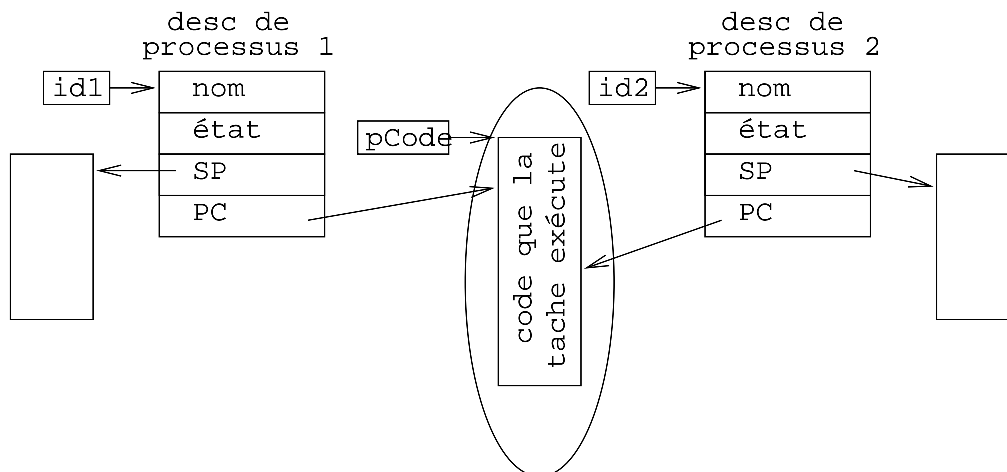
Dans VxWorks, un descripteur de processus s'appelle
Extrait de `taskLib.h` :

```
typedef struct windTcb /* WIND_TCB - task control block */
{
Q_NODE qNode ; /* 0x00 : multiway q node : rdy/pend q */
Q_NODE tickNode ; /* 0x10 : multiway q node : tick q */
```

```
Q_NODE activeNode ; /* 0x20 : multiway q node : active q */
OBJ_CORE objCore ; /* 0x30 : object management */
char * name ; /* 0x34 : pointer to task name */
int options ; /* 0x38 : task option bits */
UINT status ; /* 0x3c : status of task */
UINT priority ; /* 0x40 : task's current priority */
UINT priNormal ; /* 0x44 : task's normal priority */
UINT priMutexCnt ; /* 0x48 : nested priority mutex owned */
...
FUNCPTR entry ; /* 0x74 : entry point of task */
char * pStackBase ; /* 0x78 : points to bottom of stack */
char * pStackLimit ; /* 0x7c : points to stack limit */
...
struct __sFILE * taskStdFp[3] ; /* 0xc4 : stdin,stdout,stderr fps */
int taskStd[3] ; /* 0xd0 : stdin,stdout,stderr fds */
} WIND_TCB ;
```

TaskIdSelf() retourne l'identifiant de la tâche qui l'appelle (adresse du TCB)

2.1.2 Multitâches, et programme réentrant



Un code, pour être réentrant, ne doit pas avoir de variable globale.

Les tâches id1 et id2, qui exécutent le même code, peuvent-elles être dans des états différents ?

2.2 Etats d'une tâche dans VxWorks

La figure 1 page 8 montre succintement les 3 états principaux d'une tâche gérée par le noyau WIND.

La stratégie de passage de l'état PRÊTE à l'état EN EXÉCUTION dépend de l'algorithme d'ordonnancement (voir ordonnancement, section 2.3 page 9.

L'événement attendu peut être :

– une donnée d'entrée disponible (par exemple un octet arrivé sur un port série)

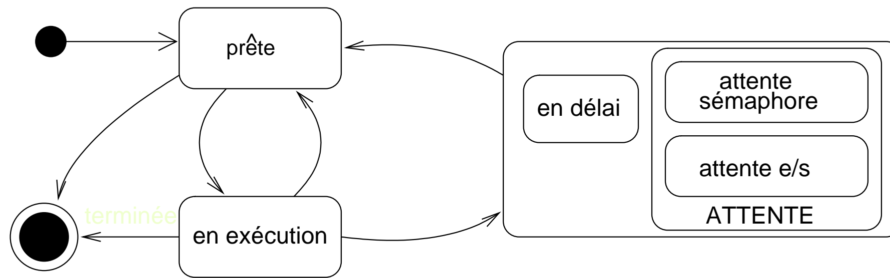


FIG. 1 – états d'une tâche gérée par le noyau

- une donnée de sortie enregistrable (par exemple une place disponible dans le tampon de sortie associé à un pilote de port série)
- un délai atteint (il a fallu temporiser)
- une ressource partagée disponible

2.2.1 Symboles utilisés dans WindView

états

en attente de délai atteint (delay)
prête (ready)
en cours (running)
en attente d'e/s (pend)
suspendue

primitives / événements

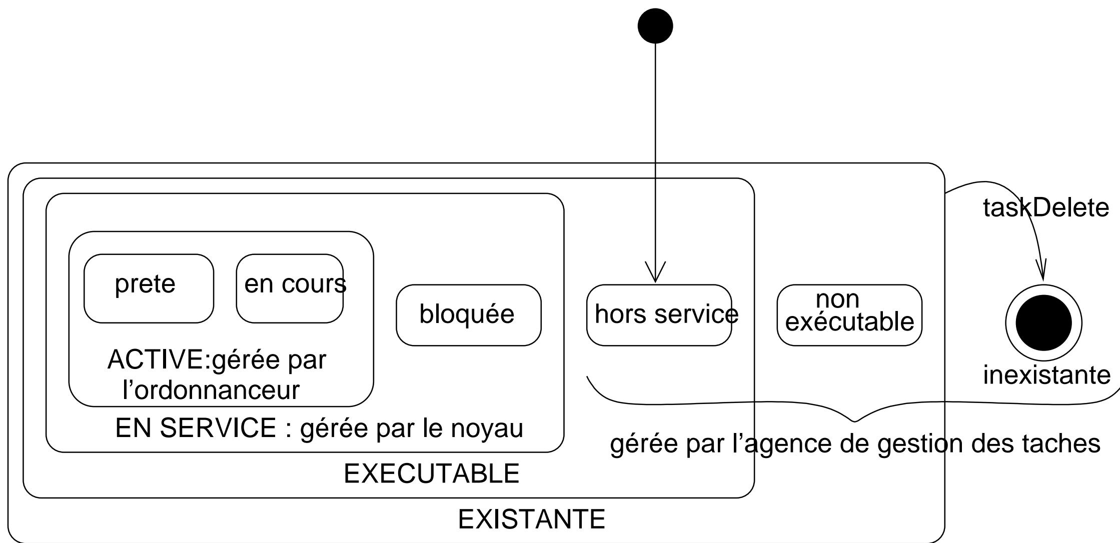
taskSpawn
tick_timeslice
taskDelete
taskDelay
timeOut
msgQReceive
msgQSend
semTake
semGive
hors délai

2.2.2 Etats, et groupements d'états SCEPTRE

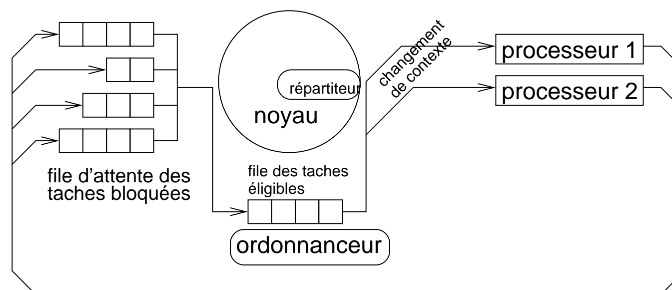
Le noyau assure la commutation des tâches, de l'état bloqué, à l'état actif

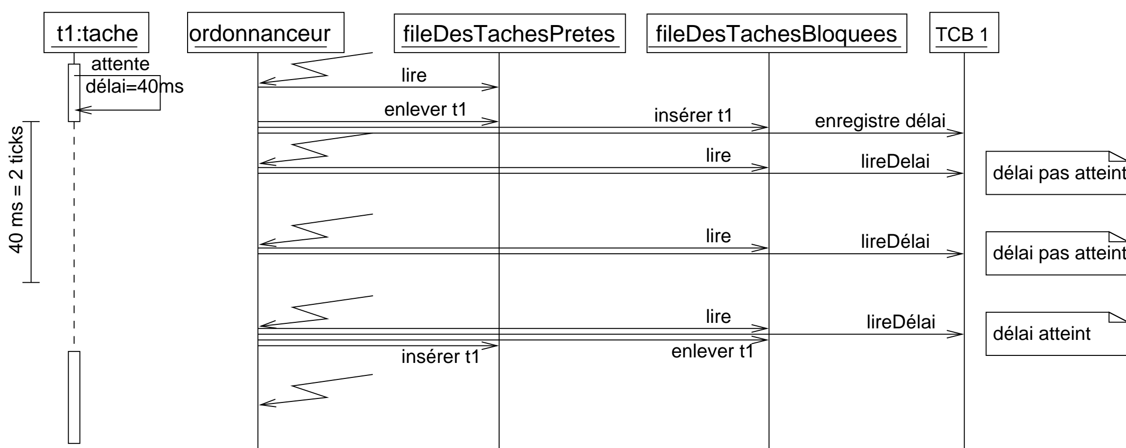
L'ordonnanceur , parmi l'ensemble des tâches , la tâche qui disposera du processeur.

Une file d'attente particulière est affectée à chaque type de blocage.



2.3 Ordonnancement (scheduling)





2.3.1 Systèmes préemptifs (preemptive scheduling), coopératifs (no preemptive scheduling)

Dans un noyau préemptif, l'ordonnanceur (preemptive scheduler) a périodiquement un droit de préemption sur la tâche en cours d'exécution. Le processeur étant réquisitionné par le noyau, l'ordonnanceur peut donc à tout instant attribuer le processeur à une autre tâche demandeuse

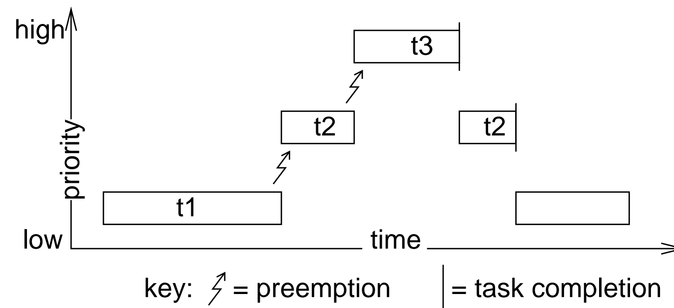


FIG. 2 – ordonnancement basé sur la priorité

de temps. La façon de choisir la tâche qui va passer en exécution est réglée par l'algorithme d'ordonnancement.

La tâche qui s'exécute actuellement ne peut pas toujours deviner qu'elle va perdre le processeur (elle peut aussi le demander explicitement).

La tâche peut perdre le processeur dans les cas suivants :

- La tâche va créer une autre tâche de
- La tâche va baisser sa propre priorité (tout en restant active)
- La tâche va exécuter une instruction qui risque de la (e/s, attente de sémaphore, ...)
- La tâche va exécuter une instruction qui risque d'activer une autre tâche de plus grande priorité (*V()*)
- La tâche veut perdre le processeur (délai non nul)
- La tâche va se terminer
- La tâche invoque explicitement l'ordonnanceur (`taskDelay(0)`)

La tâche ne peut pas deviner qu'elle va perdre le processeur dans les cas suivants :

- un événement se produit, qui provoque l'exécution d'une
- un événement logiciel se produit, qui provoque l'activation d'une tâche plus prioritaire (délai atteint)
- une tâche vient d'être créée, de plus haute priorité, et elle est

La tâche est presque sûre de ne pas perdre le processeur dans les cas suivants :

- La tâche est une tâche matérielle de haute priorité
- La tâche demande l' des IT matérielles
- La tâche demande l'arrêt de l' (`taskLock()`)

Dans un système coopératif (noyau réquisition du processeur), une tâche est exécutée jusqu'à ce qu'elle fasse appel à un service du noyau . Selon la situation, le noyau décide alors si la tâche doit continuer, ou libérer le processeur. Le temps d'attente d'une tâche prête pour passer en exécution est donc assez aléatoire.

2.3.2 Algorithme d'ordonnancement par priorité (algorithme par défaut dans VxWorks)

Il est souvent nécessaire, dans un environnement temps réel, de privilégier certaines tâches par rapport à d'autres. L'algorithme par priorité choisit, pour passer en exécution, la tâche prête de plus haute priorité.

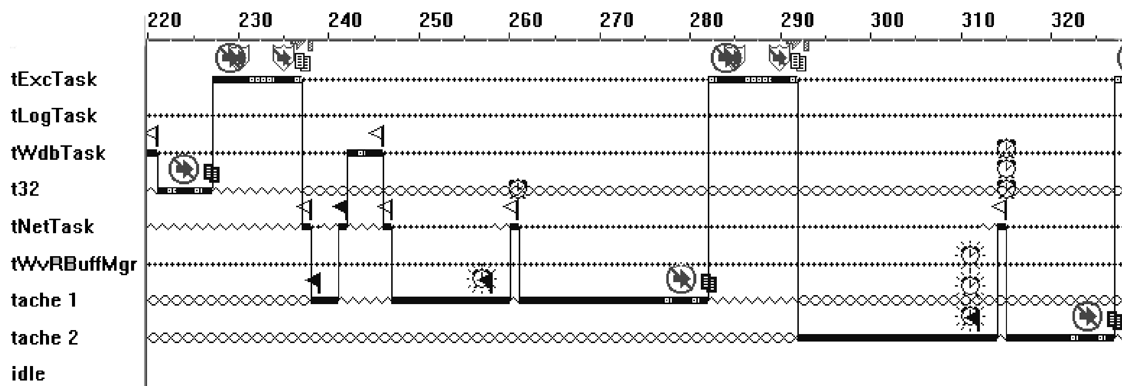
Ces priorités peuvent statiques ou dynamiques.

Dans Windows 2000, les fils d'exécution ont 32 niveaux de priorité, qui sont soit fixes, soit dynamiques. Dans le cas d'une priorité dynamique, la valeur de cette priorité varie entre deux bornes : elle augmente par exemple lors d'une attente d'e/s. La priorité d'un "thread" peut être modifiée par la fonction `SetThreadPriority()`.

2.3.3 Exercice : Exploitation d'un suivi WindView

```
#include <vxworks.h>
void leCode (void) {
int i=0 ;
  for (i=0 ; i<5000000 ; i++)
    ;
}
void demarrer() {
  kernelTimeSlice (0) ;
  taskSpawn ("tache 1",200,0,512,(FUNCPTR)leCode,0,0,0,0,0,0,0,0,0,0) ;
  taskSpawn ("tache 2",200,0,512,(FUNCPTR)leCode,0,0,0,0,0,0,0,0,0,0) ;
}
```

2.3.4 Etude du résultat obtenu



2.3.5 Algorithme d'ordonnancement en tourniquet (round robin)

Cet algorithme est l'un des plus utilisés dans les environnements multi-tâches du type multi-utilisateurs.

Chaque processus dispose d'un quantum de temps, ou (time slice) pendant lequel il s'exécute

Dans l'algorithme du tourniquet, les quanta de temps égaux rendent les différents processus égaux. Le temps CPU est réparti entre les tâches.

Dans OS9, une tâche prête voit sa priorité augmenter à chaque tranche de temps. Elle finit donc obligatoirement par s'exécuter pendant une tranche de temps, puis elle retrouve sa priorité initiale.

Les variables globales MINPRIORITY (défaut=0) et MAXAGE(défaut=65535) permettent de contrôler l'ordonnanceur, pour faire du temps réel.

MINPRIORITY impose une priorité minimum pour qu'une tâche puisse passer en exécution.

MAXAGE impose un butée de vieillissement, ce qui permet à une tâche très prioritaire de garder le processeur tant qu'elle est active.

2.3.6 Le tourniquet avec priorités (algorithme possible d'ordonnancement du noyau Wind)

On utilise généralement une combinaison des deux techniques précédentes. A chaque niveau de priorité correspond un tourniquet. L'ordonnanceur choisit le tourniquet non vide de priorité la plus élevée, et l'exécute.

Dans le noyau Wind, on peut modifier l'ordonnancement avec les appels système suivants:

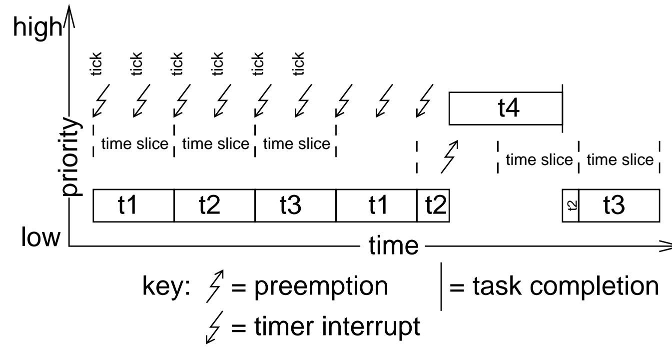


FIG. 3 – ordonnancement utilisant l'algorithme du tourniquet (round robin)

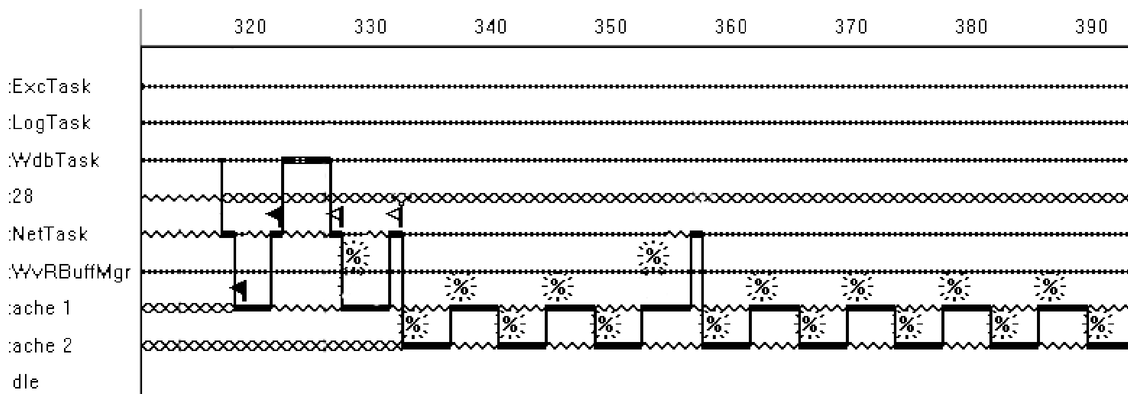
Appel système	Description
kernelTimeSlice()	
taskPrioritySet()	
taskLock()	
taskUnlock()	

2.3.7 Exercice : exploitation d'un suivi WindView

- Le code utilisé

```
#include <vxworks.h>
void leCode (void) {
int i=0;
  for (i=0; i<5000000; i++)
    ;
}
void demarrer() {
  kernelTimeSlice (2);
  taskSpawn ("tache 1",200,0,512,(FUNCPTR)leCode,0,0,0,0,0,0,0,0,0,0);
  taskSpawn ("tache 2",200,0,512,(FUNCPTR)leCode,0,0,0,0,0,0,0,0,0,0);
}
```

– Interprétation du résultat

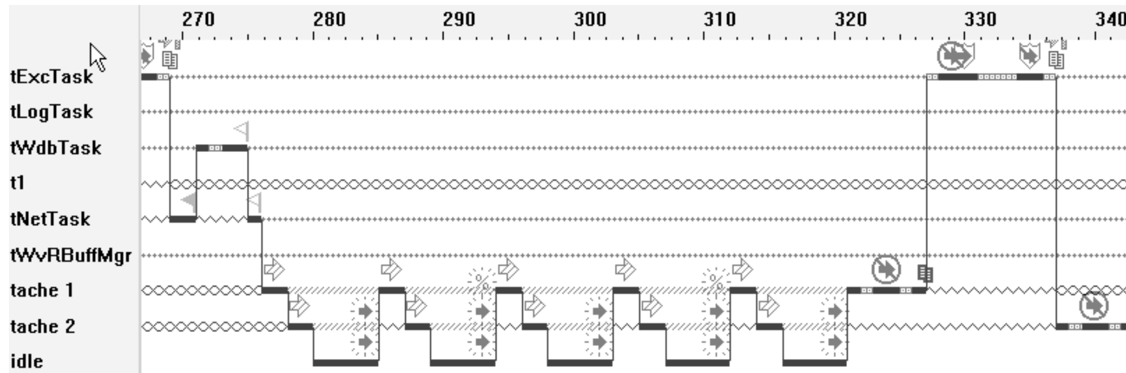


Exercice : Calculer approximativement les temps réels d'exécution des tâches tâche 1, et tâche 2.

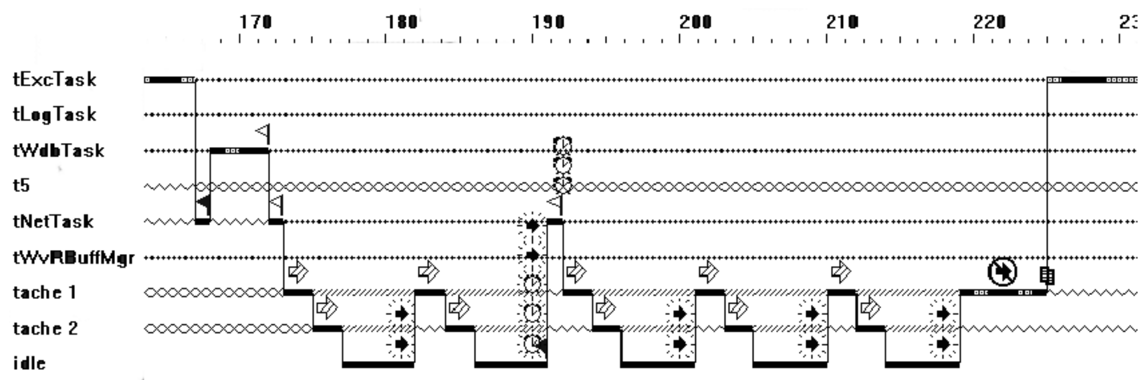
2.3.8 Influence du blocage d'une tâche sur la charge du processeur

Pour les suivis présentés, identifier l'ordonnancement, et trouver leCode.

Suivi N°1



Suivi N°2



2.3.9 La tranche de temps, et le rendement

Un paramètre important dans l'algorithme du tourniquet, est la valeur de la tranche de temps (time slice). La tranche de temps est toujours un multiple de la période de l'horloge (tick). La tranche de temps doit être aussi faible possible, pour le confort des utilisateurs, mais le temps consacré à la gestion du système doit être faible, par rapport au temps consacré aux applications.

exercice : calculer le rendement max d'un système qui a une tranche de temps de 5ms, et qui consacre 1ms au changement de contexte.

2.4 Portabilité : les primitives POSIX

2.5 Multi-tâches ou multi-threads ?

3 Exclusion entre processus

L'exclusion a pour but de limiter l'accès à une par un ou plusieurs processus. Cela concerne par exemple un fichier de données que plusieurs processus désirent mettre à jour. L'accès à ce fichier doit être réservé à un utilisateur pendant le moment où il le modifie, sinon, son contenu risque de ne plus être cohérent. La problématique est la même pour une imprimante, où un utilisateur doit se réserver son usage le temps d'une impression. On appelle ce domaine d'exclusivité une

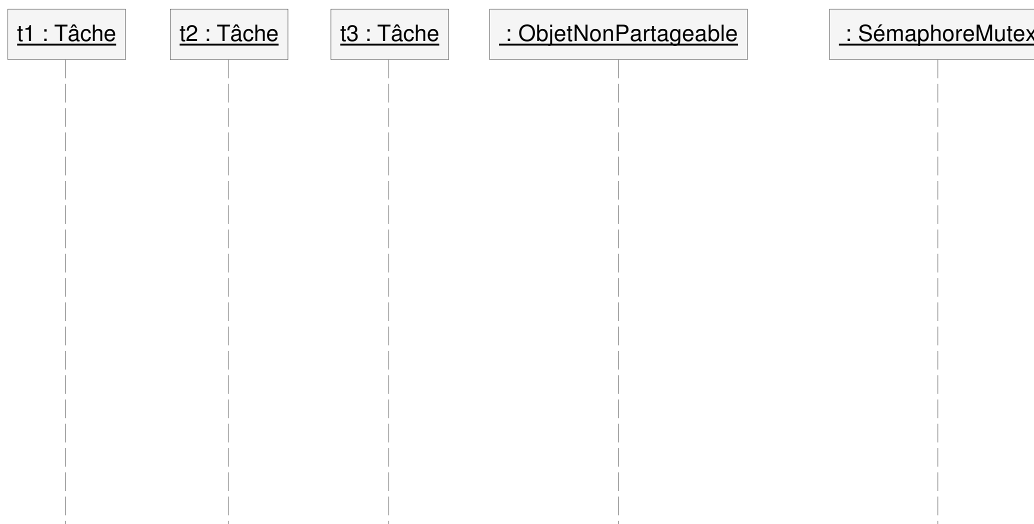
3.1 Les différentes formes d'implémentation de l'exclusion

3.1.1 Verrouillage des IT

3.1.2 Verrouillage de la préemption

taskLock(), taskUnlock()

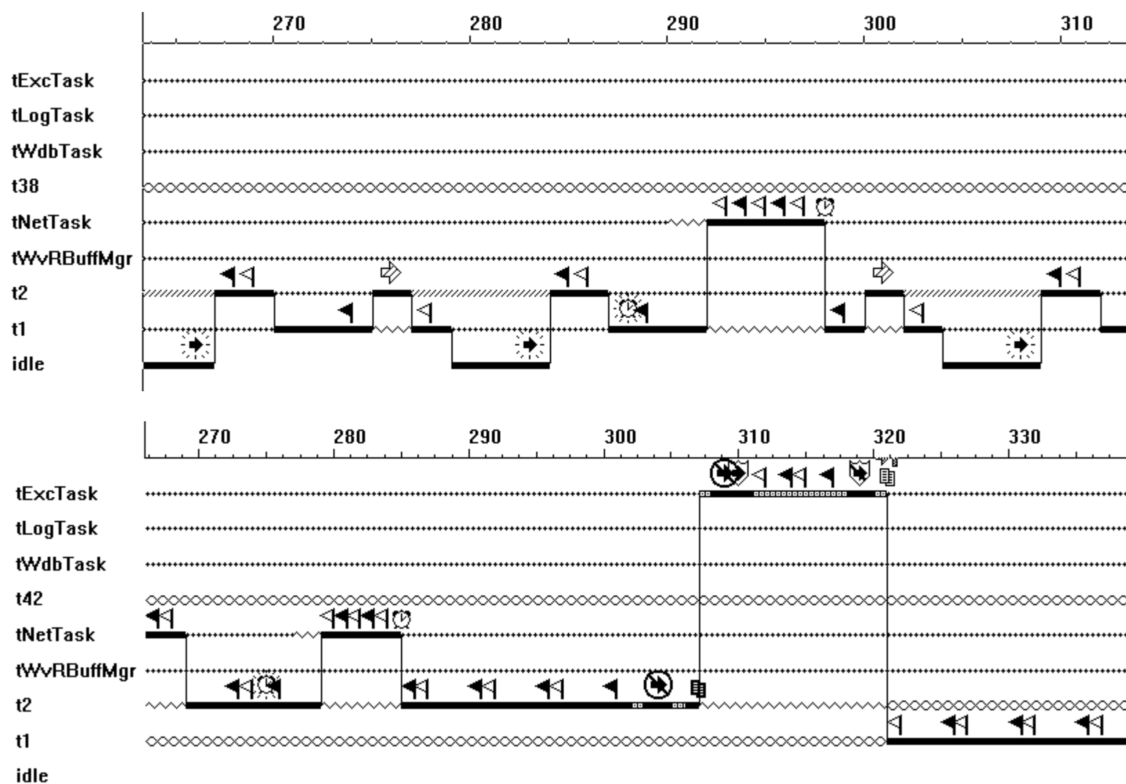
3.1.3 Sémaphores d'exclusion mutuelle (voir 4.3.6 page 33)



3.1.4 Traces typiques WindView

```
#include <vxworks.h>
#include <semLib.h>
SEM_ID sema ;
void tc1 (void) {
int i,j ;
 for (i=0 ; i<5 ; i++){
  semTake(sema,WAIT_FOREVER) ;
  for (j=0 ; j<500000 ; j++)
   ;
  semGive(sema) ;
 }
```

```
}
void tc2 (void) {
int i,j ;
 for (i=0 ; i<5 ; i++){
  semTake(sema,WAIT_FOREVER) ;
  for (j=0 ; j<500000 ; j++)
   ;
  /* taskDelay(2) :*/
  semGive(sema) ;
 }
}
void demarrer() {
 kernelTimeSlice (0) ;
 sema = semBCreate(SEM_Q_FIFO,SEM_FULL) ;
 taskSpawn ("t1",200,0,512,(FUNCPTR)tc1,0,0,0,0,0,0,0,0,0,0) ;
 taskSpawn ("t2",150,0,512,(FUNCPTR)tc2,0,0,0,0,0,0,0,0,0,0) ;
}
```



3.2 Pourquoi chercher à éviter les sections critiques ?

Pourquoi chercher à éviter les sections critiques ? Parce que cela implique un contrôle d'accès, et donc un des tâches qui ne doivent pas accéder à cette ressource. Globalement, à cause de la conception multi-tâches de l'application, (ce qui facilite la conception du code), on doit retourner à un fonctionnement plutôt de l'application. Une tâche doit se bloquer, alors qu'aucune autre tâche n'est prioritaire, et qu'elle est demandeuse de temps.

3.3 Comment éviter les sections critiques ?

Dans les systèmes multi-tâches du type multi-utilisateurs, il est difficile d'y échapper. L'exemple typique est l'imprimante, partagée par plusieurs applications.

Dans les systèmes multi-tâches du type applicatif (application multi-tâches), les ressources partagées sont parfois volontaires (mémoire partagée, dans un but de communication, tube de communication, ..), et parfois involontaires (plusieurs tâches accèdent à une fonction non réentrante, ou à un objet physique genre capteur, CAN, périphérique d'e/s TOR, carte réseau, ...).

D'une façon générale, pour éviter les sections critiques non volontaires, il est préférable de ne pas avoir d'objet partagé non partageable : chaque tâche qui accède à un objet non partageable doit être la seule à y accéder.

3.4 Comment identifier une ressource critique ?

4 Communication inter-processus (IPC)

Les IPC permettent à des tâches de coordonner leurs actions, afin d'atteindre l'objectif fixé par l'application

Les mécanismes de communication inter_tâches sont les suivants :

- les tubes de communication, et files de messages, pour communiquer entre tâches à l'intérieur d'une même unité centrale
- les sémaphores, pour la synchronisation, et l'exclusion mutuelle
- la mémoire partagée, pour manipuler des données communes
- les signaux, pour gérer les exceptions asynchrones
- les sockets et les rpc, pour une communication (utilisant le réseau de façon transparente) entre tâches s'exécutant sur les UC différentes

4.1 Le tube de communication (pipe) : un périphérique virtuel

Le tube non nommé est un moyen de communication souvent utilisé :

Le signe “|” de l’interprète de commandes Unix, relié à l’appel système pipe, permet à deux processus de communiquer tout en s’exécutant simultanément. Le premier processus fournit des données que le second exploite au fur et à mesure de leur production.

4.1.1 Caractéristiques universelles d’un tube de communication

–
– Synchronisation automatique : une tâche cherchant à lire un tube vide (ou à écrire dans un tube plein) se bloque (attente d’e/s)
– Les données sont dès qu’une tâche lit un tube (implémentation sous forme de tampon circulaire).
– périphérique d’entrée-sortie virtuel (pour les tubes nommés), donc utilisation des fonctions d’e/s standard :

`ioctl()`

4.1.2 Les tubes nommés dans VxWorks

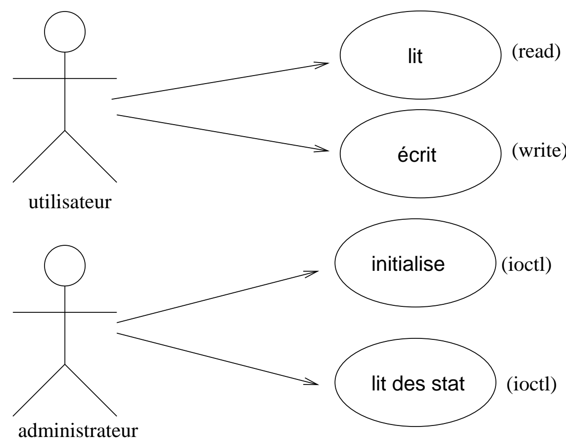
Les tubes VxWorks sont des périphérique d’entrée-sortie virtuels, gérés par le pilote `pipeDrv`. La fonction `pipeDevCreate()` crée un périphérique de type `pipe`

```
status = pipeDevCreate (‘‘/pipe/monTube’’, nbMsgsMax, tailleMaxMsg)
```

Comme tout périphérique d’e/s, les tubes peuvent être utilisés avec la fonctionnalité `select()`. Cette fonction permet à une tâche d’attendre qu’une donnée arrive sur l’un quelconque d’un ensemble de périphériques.

Ainsi, une tâche peut être en attente de donnée provenant d’un ensemble de tubes, de prises réseau (socket), de ports série ..

4.1.3 Cas d’utilisation des tubes `VxWorks`



Des fonctionnalités d'administration, à des fins de statistiques ou de tests, sont disponibles pour les tubes VxWorks, grâce à la fonction

```
ioctl (canal, numéro de la sous-fonction, paramètre de la sous-fonction)
```

fonctionnalité	sous-fonction de ioctl
nb de messages dans le tube ?	FIONMSGS
vider le tube en force	FIOFLUSH
nb d'octets non lus dans le premier message ?	FIONREAD

ioctl (canal, FIONMSGS, &nbMessages)

4.1.4 Les tubes nommés dans Unix

Les commandes `mknod` et `mkfifo` permettent de créer un fichier `spécial` du type `pipe`

```
mknod <nom du tube> type
```

avec : type = p (pipe)

```
mkfifo <nom du tube>
```

Le tube est alors utilisable en mode commandes par

```
ls > monTube
```

et en C par

```
canal=open ("monTube", O_RDWR, S_IRWXU) ;
```

4.2 Les files de messages

4.2.1 Files de messages Unix

Les files de messages Unix permettent la communication entre les processus (Inter Process Communication)

Principe

Utilise le principe des : on dépose dans la boîte un message que d'autres processus pourront lire.
Le mode de lecture/écriture se fait de manière groupée par une structure de taille donnée. Chaque instruction de lecture ou d'écriture se fait sur un message entier (toute la structure de message).
Pour que les lectures soient compatibles avec les écritures, les messages sont . On utilisera une structure dont le premier champ est un qui doit contenir le type du message.
Le type d'un message est un entier strictement .
Le type du message permet aux applications d'effectuer les bons ordres de lecture, mais aussi permet de un message dans la file.

Quelques macros permettant de paramétrer les appels :

`MSG_NOERROR`	l'extraction d'un message trop long n'entraine pas d'erreur (le message est tronqué).
`MSG_R`	autorisation de lire dans la file.
`MSG_W`	autorisation d'écrire dans la file.
`MSG_RWAIT`	indication qu'un processus est bloqué en lecture.
`MSG_WWAIT`	indication qu'un processus est bloqué en écriture.

La structure msqid_ds

```
struct msqid_ds {
    struct ipc_perm    msg_perm ;        /* opération permission struct */
    struct __msg       *msg_first ;      /* ptr to first message on q */
    struct __msg       *msg_last ;       /* ptr to last message on q */
    unsigned short int msg_qnum ;        /* # of messages on q */
    unsigned short int msg_qbytes ;      /* max # of bytes on q */
    pid_t              msg_lspid ;       /* pid of last msgsnd */
    pid_t              msg_lrpid ;       /* pid of last msgrcv */
    time_t             msg_stime ;       /* last msgsnd time */
    time_t             msg_rtime ;       /* last msgrcv time */
    time_t             msg_ctime ;       /* last change time */
    unsigned short int msg_cbytes ;      /* current # bytes on q */
    char               msg_pad[22] ;     /* room for future expansion */
   } ;
```

La structure générique d'un message

La structure suivante est un modèle pour les messages :

```
struct msgbuf {
    long mtype ;    /* type du message */
    char mtext[1] ; /* texte du message */
} ;
```

par exemple :

```
struct msg_buffer {
    long typeDuMsg ;
    float a ;
    char m[8] ;
} ;
```

Attention on ne peut pas échanger des adresses, en effet les adresses virtuelles utilisées par les différents programmes qui échangent des messages sont à priori , de plus les zones de mémoire manipulables par deux processus sont .

Important : le premier champ doit être un entier long qui contiendra le type du message.

Utilisation des files de messages

La primitive

```
#include <sys/msg.h>
int msgget (key_t  clef, int options) ;
```

est une fonction proche de la fonction open. Elle renvoie un descripteur d'IPC de file de messages de clef d'accès `clef`. Avec création ou non de la file de messages (cf le paramètre `options`).
La valeur du paramètre `options` doit être construite avec une conjonction du mode d'accès et des constantes `IPC_CREAT` et `IPC_EXCL`.

```
Si cle == IPC_PRIVATE, alors
  une nouvelle file de messages privée est créée.
Sinon
  Si la cle correspond à une file inexistante :
    SI IPC_CREAT est positionné (dans options), une nouvelle file est créée
    associée à cette clé, avec les droits définis dans options. Le créateur
    et le propriétaire sont positionnés aux valeurs de l'euid et du egid
    du processus réalisant l'appel, le dipc interne de la file est retourné.

    Sinon erreur retour -1
  Sinon la clé correspond à une file déjà existante :
    Si les 2 indicateurs IPC_CREAT et IPC_EXCL sont positionnés dans options
      une erreur est détectée, retour -1, errno = EEXIST.
    Sinon l'identification de la file est retourné.
```

En bref, IPC_EXCL nous permet de vérifier que la file n'existait pas déjà.

L'envoi de message

```
#include <sys/msg.h>
int msgsnd (int dipc, const void *p_msg, int lg, int options) ;
```

Envoie dans la file `dipc` le message pointé par `p_msg`.
`lg` : taille du message égale à `sizeof(struct msgbuf)-sizeof(long)`, le n'étant pas compté dans cette longueur.
Valeur de retour (0) succès (-1) échec.

Valeur de errno en cas d'échec (man msgsnd)	
EINVAL	file inexistante
EPERM	
EINVAL	type de message incorrect

Si `IPC_NOWAIT` est positionné, l'envoi de messages sur une file pleine n'est plus bloquant, alors dans le cas d'une file pleine, la fonction retourne -1 et errno est positionné à `EAGAIN`.

Un appel de msgsnd bloqué peut être interrompu par un signal ou par la destruction de la file de message. Dans ce cas, elle renvoie (-1) et errno est positionné à `EINTR` ou `EIDRM`.

La primitive de lecture (extraction)

```
#include <sys/msg.h>
int msgrcv(int dipc, void *p_msg, int taille, long type, int options) ;
```

est une demande de lecture dans la file `dipc` d'un message de longueur inférieure ou égale à taille, qui sera copié dans la zone pointée par
`options` est une combinaison (OU) des constantes :

`IPC_NOWAIT`	même si la file est vide, la lecture est non-bloquante
`MSG_NOERROR`	si le texte du message à extraire est de longueur supérieure à taille, alors le message est extrait tronqué sans signaler d'erreur
`MSG_EXCEPT`	associé avec type>0, permet de les lire le premier msg, qui n'est PAS de ce type

Le paramètre `type` permet de spécifier le type du message à extraire :

1. si type > 0, le plus ancien message de ce type est extrait (si l'option MSG_EXCEPT est présente, le premier message qui n'est PAS de ce type)
2. si type $== 0$, le plus ancien message (celui qui est en tête de la file) est extrait ;
3. si type < 0, le message le plus ancien du type le plus petit, mais inférieur ou égal à ,
   est extrait. Ceci permet de définir des priorités entre les messages (voir partie 4.2.1)

L'appel est si il n'y a pas de message du type voulu en attente. (sauf si l'option IPC_NOWAIT est choisie)

cause d'échec	*signification*
`EINVAL`	file inexistante
`EINVAL`	taille négative
`E2BIG`	taille message > taille, et pas de `MSG_NOERROR`
`ENOMSG`	pas de message et `IPC_NOWAIT`

et les mêmes codes d'interruptions que msgsnd.

– Implantation des messages urgents

La file de message Unix, grâce au champ type, permet l'implantation de la notion de message urgent, mais cette notion demande la des tâches productrices, et des tâches consommatrices (implantation non)

Même si une tâche produit des messages très urgents (type=1), ces messages restent placés dans la file dans l'ordre chronologique d'arrivée. La tâche consommatrice pourra les consommer en premier, .

La notion d'urgence n'est donc implantée que de façon potentielle. Le traitement effectif des messages urgents est implanté au niveau .

type du message spécifié par la tâche productrice	type du message spécifié par la tâche consommatrice	résultat
quelconque	0	file FIFO (pas d'urgence)
n	n	file FIFO : le plus ancien message de type n est extrait
n	-n	le plus ancien des messages les plus urgents(de type <=n) (s'ils existent) est extrait

Il est important de remarquer que les messages sont rangés dans la file par ordre (file FIFO), quels que soient leurs types : en aucun cas, les messages ne peuvent se .

Exemple de file (les messages de droites sont les plus anciens)

20	40	50	40	60	80	100	40	100	50	100

opération lireFile avec type =	résultat obtenu	aperçu de la file après l'opération
-5	blocage (si pas IPC_NOWAIT)	20 40 50 40 60 80 100 40 100 50 100
-60		40 50 40 60 80 100 40 100 50 100
-60	extrait le message le plus "urgent" (ici le plus ancien des messages de type 40)	40 50 40 60 80 100 100 50 100
-60	extrait le message le plus "urgent" (ici le plus ancien des messages de type 40)	40 50 60 80 100 100 50 100
100	extrait le message le plus ancien des messages de type 100	40 50 60 80 100 100 50
0		40 50 60 80 100 100

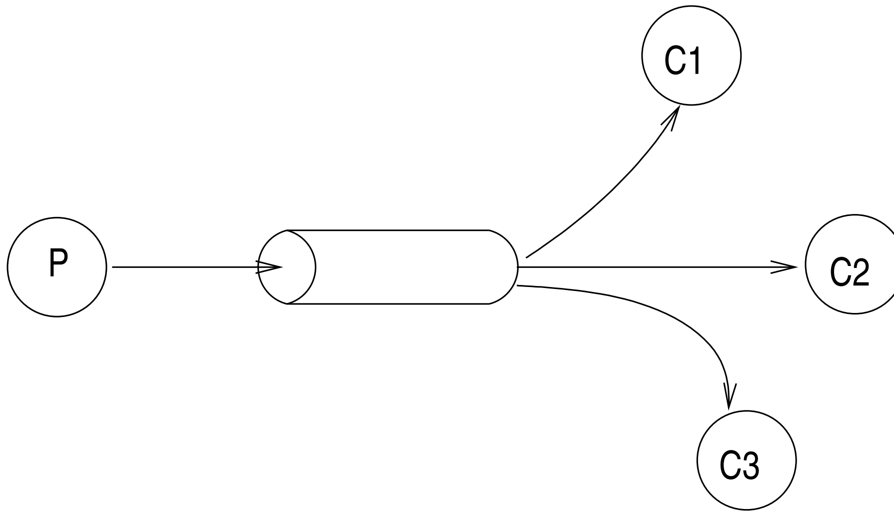
La primitive de contrôle

```
#include <sys/msg.h>
int msgctl (int dipc, int options, struct msqid *pmsqid);
```

permet de travailler sur la structure msqid pointée par pmsqid de la file de message dipc.

Valeur de options	
IPC_STAT	lecture de la structure
IPC_SET	positionnement, seuls les champs uid, gid et perm sont modifiables
IPC_RMID	permet de détruire la file de messages (super-utilisateur, ou créateur de la file de messages)

Exercice : implanter la notion de destinataire par défaut



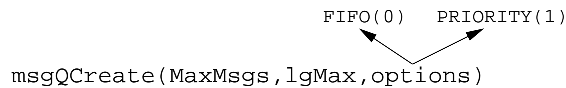
4.2.2 Les files de messages VxWorks

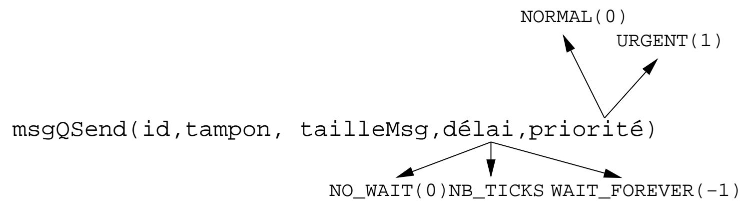
msgQ
– id
+ Create()
+ Send()
+ Receive()
+ NumMsgs()
+ Init()
+ InfoGet()

pipe

+ DevCreate()
+ read()
+ write()
+ ioctl()
+ open()

Les files VxWorks ont des spécificités temps réel





Les tubes de communication sont basés sur les   . Cela explique que les tubes VxWorks manipulent des messages.

C'est par `ioctl()` qu'on atteint des fonctions `msgQ` spéciales.

fonction tube	fonction msgQ	
DevCreate()	Create()	
open()	Create()	
read()	Receive()	
write()	Send()	
ioctl (canal,FIONREAD)	InfoGet()	réponse à éplucher
ioctl (canal,FIONMSGS)	NumMsgs()	
ioctl (canal,FIOFLUSH)	Init()	

La gestion des tubes impliquant une interface , certaines fonctionnalités, présentes dans les files de messages, ont disparu.

Il s'agit principalement de caractéristiques relatives à la gestion du (fonctionnalités temps réel) :

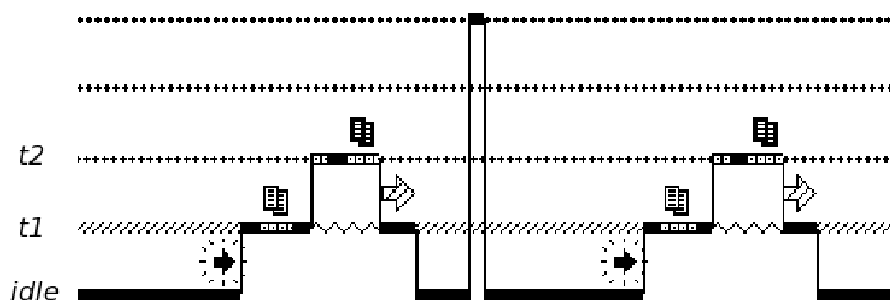
fonction pipe	fonction msgQ	commentaire
DevCreate(nom, nbMaxMsgs, lgMax)	Create(nbMaxMsgs, lgMax, options)	le paramètre options permet de choisir le mode de rangement des tâches (FIFO, ou par priorités croissantes)
write(canal, tampon, nbOctets)	Send(id, tampon, délaiMax, priorité)	délaiMax permet de bloquer la tâche avec une butée de temps. priorité permet d'implanter la notion de message urgent
read(canal, tampon, nbOctets)	Receive(id, tampon, délaiMax)	délaiMax permet de bloquer la tâche avec une butée de temps

La création d'une file de messages permet de spécifier de quelle façon seront rangées les tâches bloquées par cette file :

- MSG_Q_FIFO : les tâches bloquées par cette file sont rangées dans l'ordre
- MSG_Q_PRIORITY : les tâches bloquées par cette file sont rangées par

- La lecture d'une file de message peut être , si la file est vide. Le paramètre délaiMax permet de rendre l'attente limitée dans le temps, dans le cas où l'application estime qu'une donnée qui arrive n'a plus de valeur (cf définition du temps réel)

4.2.3 Symbolisation WindView des événements associés aux files de messages

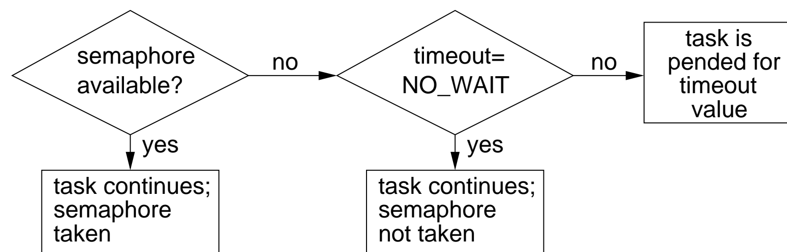


4.3 Les sémaphores

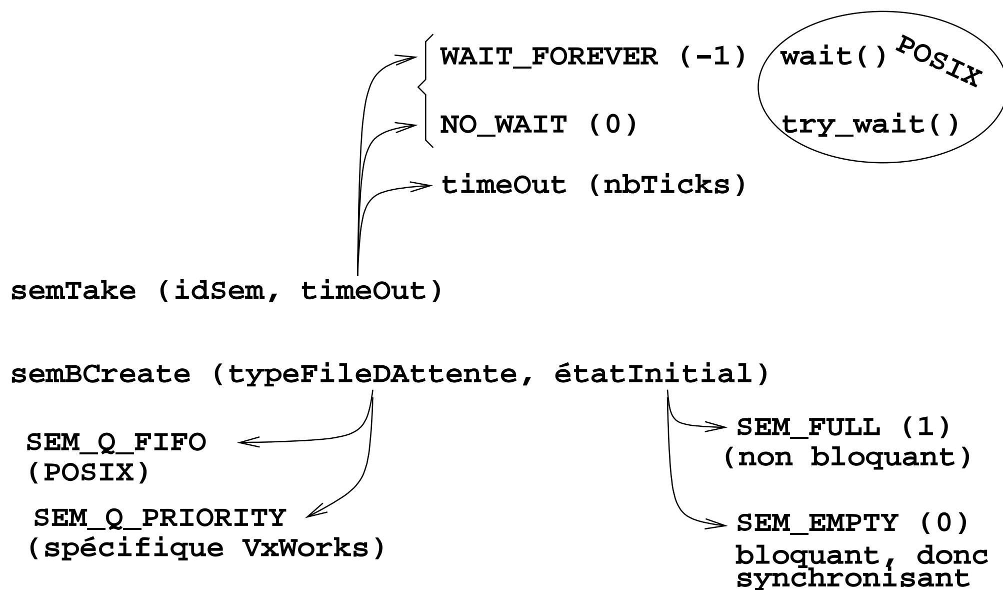
4.3.1 Sémaphore booléen d'usage général

Ce sémaphore est capable de régler les deux formes de la coordination inter-tâches : la synchronisation, et l'exclusion mutuelle. Il est léger et efficace. Dans le noyau WIND, un sémaphore d'exclusion mutuelle peut être implanté soit sous forme d'un sémaphore booléen d'usage général (avec une valeur initiale = EMPTY), soit sous la forme d'un sémaphore booléen disposant d'options spéciales permettant de régler les problèmes spécifiques liés à l'exclusion mutuelle (voir la partie 4.3.6 page 33) ;

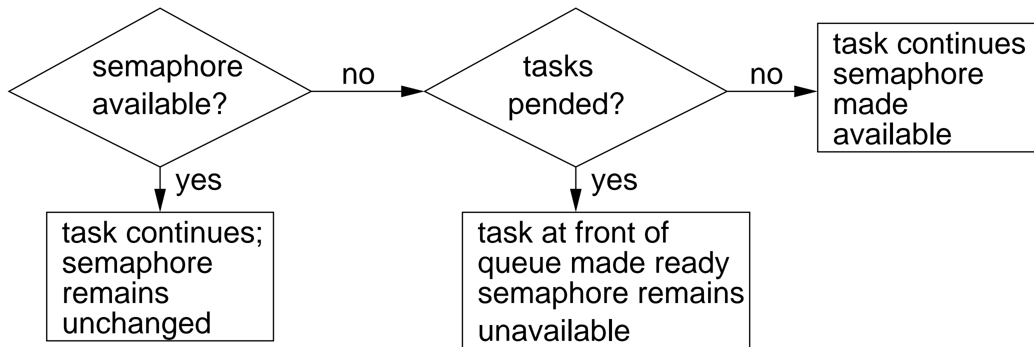
Un sémaphore booléen peut être vu comme un indicateur (drapeau) qui est levé (plein) ou baissé (vide). Lorsqu'une tâche cherche à prendre un sémaphore booléen (avec `P()`), le résultat dépend de l'état qu'avait le sémaphore (levé ou baissé) au moment de l'appel à la fonction `P()`. cf figure.



Si le sémaphore est levé, il est baissé, et la tâche qui a fait l'appel à la requête P() continue son exécution. Si le sémaphore est déjà baissé (par une autre tâche), la tâche qui fait appel à la requête P() est placée dans une file des tâches bloquées, et elle passe dans un état “attente d'un sémaphore levé” : elle attend que le sémaphore soit levé (par une autre tâche)



When a task gives a binary semaphore, using `semGive()`, the outcome also depends on whether the semaphore is available (`full`) or unavailable (`empty`) at the time of the call ; cf figure.

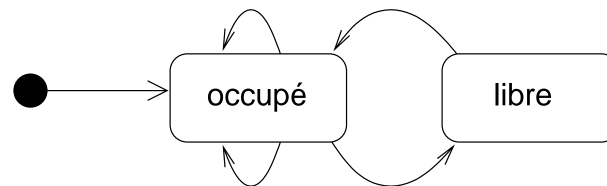


If the semaphore is already available (`full`), giving the semaphore has no effect at all. If the semaphore is unavailable (`empty`) and no task is waiting to take it, then the semaphore becomes available (`full`). If the semaphore is unavailable (`empty`) and one or more tasks are pending on its availability, then the first task in the queue of blocked tasks is unblocked, and the semaphore is left unavailable (`empty`).

4.3.2 Sémaphores de synchronisation

Un sémaphore de synchronisation permet à une tâche de se synchroniser avec des événements extérieurs.

Lorsqu'il est utilisé pour synchroniser deux tâches, un sémaphore peut représenter une condition, ou un événement qu'une tâche attend. Initialement, le sémaphore est non disponible (vide), et donc probablement bloquant. Une tâche, ou une routine d'IT signale l'occurrence de l'événement espéré, en libérant le sémaphore (see Interrupt Service Code for a complete discussion of ISRs). Une autre tâche attend le sémaphore (`P()`, ou `sem_wait()`, ou `semTake()`. Cette tâche en attente est bloquée jusqu'à ce que l'événement espéré se produise, et donc que le sémaphore soit libéré. Lorsqu'il est utilisé en synchronisation, le sémaphore a une valeur initiale 0 (EMPTY) : une tâche attend, pour prendre le sémaphore, que l'autre le libère.

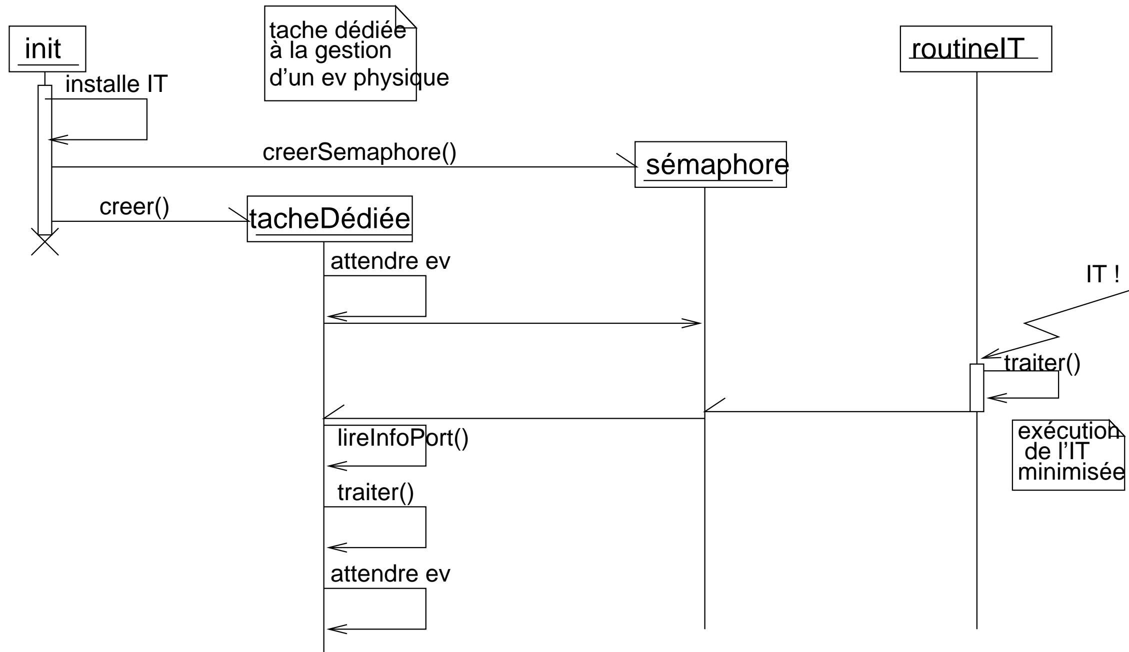


In Example, the init( ) routine creates the binary semaphore, attaches an ISR to an event, and spawns a task to process the event.
La fonction `task1()` s'exécute jusqu'à ce qu'elle appelle `semTake()`. Elle reste bloquée jusqu'à ce qu'un événement active la routine d'IT, qui elle-même appelle `semGive()`. Lorsque la routine a fini de s'exécuter, `task1()` passe en exécution, pour mener les actions consécutives à l'événement détecté.
There is an advantage of handling event processing within the context of a dedicated task : less processing takes place at interrupt level, thereby reducing interrupt latency. This model of event processing is recommended for real-time applications.

Exemple : Utilisation des sémaphore en synchronisation entre tâches

```
/* This example shows the use of semaphores for task synchronization. */
/* includes */
#include "vxWorks.h"
#include "semLib.h"
#include "arch/arch/ivarch.h" /* replace arch with architecture type */
SEM_ID syncSem ;        /* ID of sync semaphore */
init ( int someIntNum ) {
/* connect interrupt service routine */
intConnect (INUM_TO_IVEC (someIntNum), eventInterruptSvcRout, 0) ;
/* create semaphore */
syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY) ;
/* spawn task used for synchronization. */
taskSpawn ("tacheDediee", 100, 0, 20000, task1, 0,0,0,0,0,0,0,0,0,0) ;
}

task1 (void) {
         ...
         semTake (syncSem, WAIT_FOREVER) ; /* wait for event to occur */
         printf ("task 1 got the semaphore\n") ;
         ... /* process event */
```

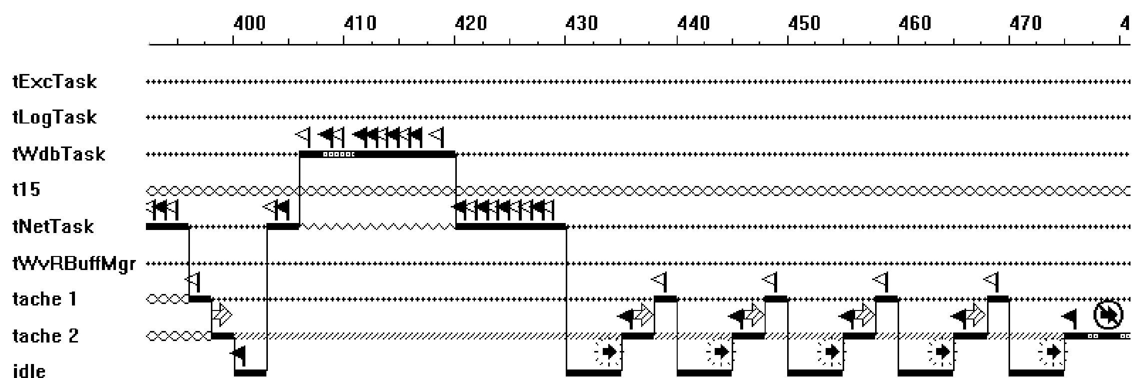
```
}

eventInterruptSvcRout (void) {
        ...
        semGive (syncSem);   /* let task 1 process event */
        ...
}
```

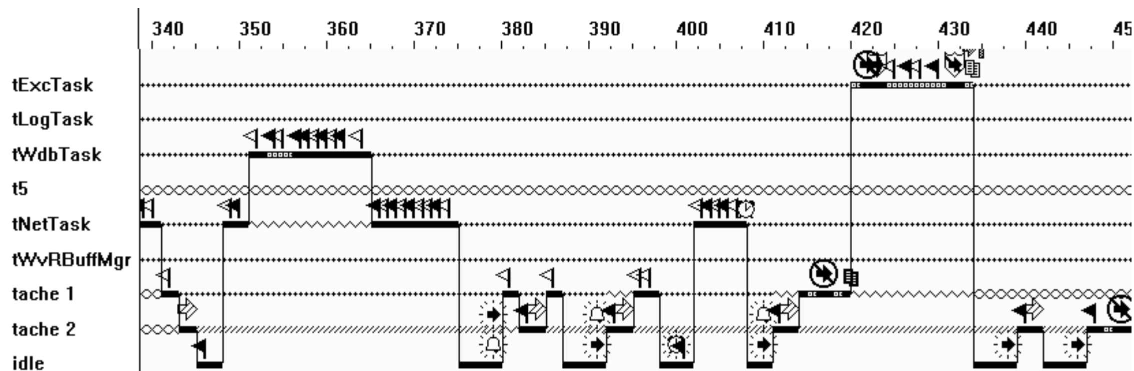


4.3.3 Suivi WindView

Fichier support :

```
#include <vxworks.h>
#include <semLib.h>
SEM_ID sema ;
void t1 (void) {
int i=0 ;
 for (i=0 ; i<5 ; i++){
  semTake(sema,WAIT_FOREVER) ;
 }
}
void t2 (void) {
int i=0 ;
 for (i=0 ; i<5 ; i++){
  taskDelay(2) ;
  semGive(sema) ;
 }
}
void demarrer() {
 /* kernelTimeSlice (0) ; */
 sema = semBCreate(SEM_Q_FIFO,SEM_EMPTY) ;
 taskSpawn ("tache 1",200,0,512,(FUNCPTR)t1,0,0,0,0,0,0,0,0,0,0) ;
 taskSpawn ("tache 2",200,0,512,(FUNCPTR)t2,0,0,0,0,0,0,0,0,0,0) ;
}
```





4.3.4 Exemple de sémaphore de synchronisation POSIX (linux pthread)

```
#include <stdio.h>
#include "pthread.h"
#include "semaphore.h"
#define VALINIT 5
sem_t monSemaphore ;
void init(void) {
 sem_init(&monSemaphore, 0, VALINIT) ;
}
void * tacheEsclave(void * data) {
int val ;
int i ;
 for (i=0 ; i<10 ; i++){
  sleep(1) ;
  sem_post(&monSemaphore) ;
  sem_getvalue(&monSemaphore, &val) ;
  printf (" valeur du sémaphore : %d\n", val) ;
 }
}
void * tacheMaitresse(void * data) {
int i ;
 for (i=0 ; i<10 ;i++) {
  sem_wait(&monSemaphore) ;
  printf ("tâche maîtresse -> sémaphore OK\n") ;
 }
 return NULL ;
}
int main(void) {
pthread_t th_a, th_b ;
void * retval ;
 init() ;
 /* Création des threads */
 pthread_create(&th_a, NULL, tacheEsclave, 0) ;
 pthread_create(&th_b, NULL, tacheMaitresse, 0) ;
 /* Attente de la fin des tâches enfant */
 pthread_join(th_a, &retval) ;
 pthread_join(th_b, &retval) ;
 return 0 ;
}
```

4.3.5 Exemple de synchronisation sur la mort du premier enfant

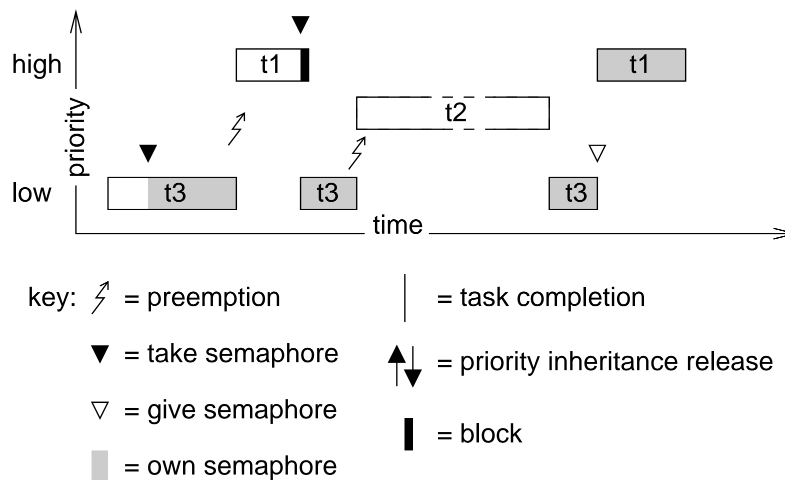
```
#include <stdio.h>
#include "pthread.h"
#include "semaphore.h"
#define VALINIT 0
sem_t semaphore_1;
sem_t semaphore_2;
void init(void) {
 sem_init(&semaphore_1, 0, VALINIT);
 sem_init(&semaphore_2, 0, VALINIT);
}
void * tache_1(void * data) {
 sem_wait(&semaphore_1);
 printf ("tâche 1 -> je commence\n");
 sleep(4);
 sem_post(&semaphore_2);
}
void * tache_2(void * data) {
 sem_wait(&semaphore_1);
 printf ("tâche 2 -> je commence\n");
 sleep(10);
 sem_post(&semaphore_2);
 return NULL;
}
int main(void) {
pthread_t th_a, th_b;
void * retval;
 init();
 /* Création des threads */
 pthread_create(&th_a, NULL, tache_1, 0);
 pthread_create(&th_b, NULL, tache_2, 0);
 sem_post(&semaphore_1);
 sem_post(&semaphore_1);
 /* Attente de la fin d'une des tâches enfant */
 sem_wait(&semaphore_2);
 printf("tâche parent -> tache fille 1 terminée\n");
/*
 pthread_kill(th_a,10);
 pthread_kill(th_b,10);
*/
 pthread_cancel(th_a);
 pthread_cancel(th_b);
 /* pthread_join(th_b, &retval); */
 return 0;
}
```

4.3.6 Sémaphores d'exclusion mutuelle dans VxWorks

Le sémaphore d'exclusion mutuelle est un sémaphore booléen spécial, réglant les problèmes inhérents à l'exclusion mutuelle, c'est-à-dire l'inversion de priorité, la protection contre la destruction, et l'accès récursif aux ressources.

Le comportement d'un tel sémaphore est donc identique au sémaphore booléen, avec les exceptions suivantes :
- il ne peut être utilisé que pour l'exclusion mutuelle
- il ne peut être rendu que la tâche qui l'a pris
- il ne peut pas être rendu par une routine d'IT
- l'opération `semFlush()` est interdite.

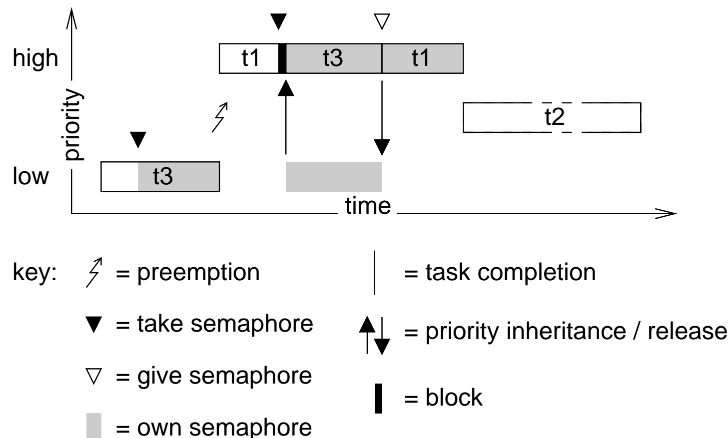
Le problème de l'inversion de priorité



Elle se produit lorsqu'une tâche de priorité élevée est obligée d'attendre pendant une durée indéfinie qu'une tâche de priorité plus faible se termine. Dans ce scénario, t1, t2, et t3 sont des tâches de priorité haute, moyenne, et basse. t3 a pris possessiond'une ressource en prenant le sémaphore de protection associé. Lorsque t1 prend le processeur (par préemption) à t3, et cherche à utiliser la ressource (en prenant le même sémaphore), elle se bloque.

Si on était assuré que t1 ne restera pas bloquée plus longtemps que le temps nécessaire à t3 pour terminer l'utilisation de la ressource, il n'y aurait pas de problème, car la ressource ne peut pas être préemptée. Cependant, la tâche de faible priorité est préemptible par une tâche de priorité moyenne (comme t2), ce qui peut empêcher t3 de libérer la ressource. Cette condition peut durer, ce qui bloque t1 pendant un temps indéterminé.

L'héritage de la priorité



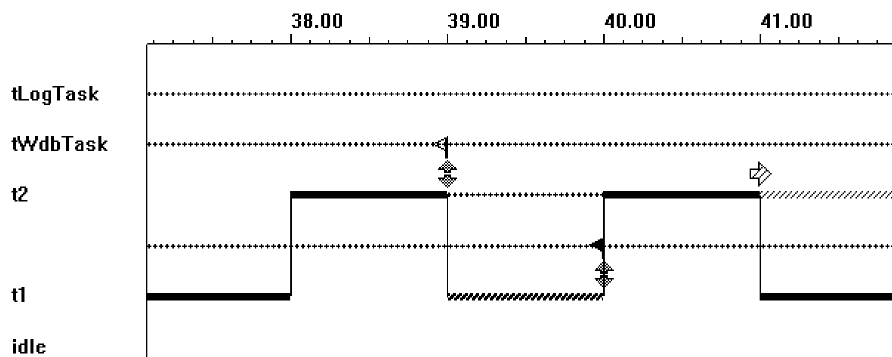
Dans le noyau Wind, le sémaphore d'exclusion mutuelle dispose de l'option `SEM_INVERSION_SAFE`, qui autorise l'algorithme d'héritage de la priorité. Ce protocole d'héritage de la priorité garantit qu'une tâche qui possède une ressource s'exécute avec une priorité égale à celle de la tâche la plus prioritaire qui est bloquée sur cette ressource. La priorité de la tâche est donc augmentée, et elle reste à ce niveau de priorité élevé jusqu'à ce que tous les sémaphores d'exclusion mutuelle que possède cette tâche soient relâchés : la tâche retourne alors à sa priorité normale. De cette façon, la tâche héritante est protégée contre toute préemption par des tâches de priorité intermédiaire. Cette option doit être utilisée en conjonction avec l'option de rangement des tâches dans la file d'attente par priorités (`SEM_Q_PRIORITY`).

Dans la figure, l'héritage de la priorité résoud le problème de l'inversion de la priorité, en élevant la priorité de t3 à la priorité de t1, pendant le temps que t1 est bloqué sur le sémaphore. Ceci protège t3, et indirectement t1 de la préemption par t2.

L'exemple suivant crée un sémaphore d'exclusion mutuelle qui utilise l'algorithme d'héritage de la priorité :

```
semId=semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE) ;
```

Observation avec WindView



Protection contre la destruction

La destruction des tâches est un autre problème lié à l'exclusion mutuelle. A l'intérieur d'une région critique protégée par un sémaphore, il est souvent utile de protéger la tâche qui s'exécute contre une destruction imprévue, qui pourrait être catastrophique : la ressource pourrait se retrouver dans un état inutilisable, et le sémaphore protégeant la ressource pourrait se retrouver indisponible, empêchant tout accès à la ressource.

Dans le noyau WIND, les primitives `taskSafe()` et `taskUnsafe()` fournissent une solution contre la destruction d'une tâche. Cependant, le sémaphore d'exclusion mutuelle offre l'option `SEM_DELETE_SAFE`, qui inclut, lors d'un appel `semTake()`, un appel implicite à la primitive `taskSafe()`, et lors d'un appel à `semGive()`, un appel implicite à la primitive `semUnsafe()`. De cette façon, une tâche peut être protégée contre la destruction quand elle possède le sémaphore. Cette option est plus efficace que l'utilisation explicite des primitives `taskSafe()` et `taskUnsafe()`, car le code résultant fait moins d'appels au noyau.

```
semId = semMCreate (SEM_Q_FIFO | SEM_DELETE_SAFE);
```

Accès récursif à une ressource

4.3.7 Sémaphores à compte

Le sémaphore à compte se comporte comme un sémaphore booléen, sauf qu'il garde trace du nombre de fois que le sémaphore est libéré. A chaque fois que le sémaphore est libéré, sa valeur est décrémentée. Lorsque sa valeur atteint 0, une tâche qui cherche à le prendre se bloque. De la même façon qu'un sémaphore booléen, s'il est libéré, et qu'une tâche est bloquée, cette dernière redevient prête. Cependant, contrairement au sémaphore booléen, si un sémaphore est libéré, alors qu'aucune tâche n'est bloquée, sa valeur est incrémentée. Cela a pour conséquence qu'un sémaphore à compte libéré deux fois, peut être pris deux fois sans blocage.

Exemple :

appel système	valeur après l'appel	résultat
semCCreate()	3	sémaphore initialisé à 3
semTake()		
semTake()		
semTake()		
semTake()		
semGive()		
semGive()		

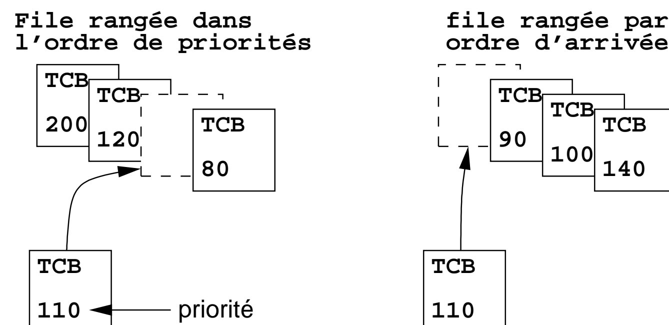
Les sémaphores à compte sont utiles pour protéger des copies multiples d'une ressource. Par exemple, dans un système multi-utilisateurs, le système s'écroule à partir de 5 compilations simultanées. Ou un tampon circulaire de 256 entrées peut être contrôlé par un sémaphore à compte de valeur initiale de 256.

4.3.8 Options spéciales des sémaphores dans le noyau WIND

Ces options permettent de prendre en compte les contraintes temps réel d'une application multi-tâches

Elles ne sont pas disponibles pour les sémaphores POSIX.

- Butée de temps (time out)
- file d'attente rangée dans l'ordre des priorités



4.3.9 Comparaison entre les sémaphores POSIX, et WIND

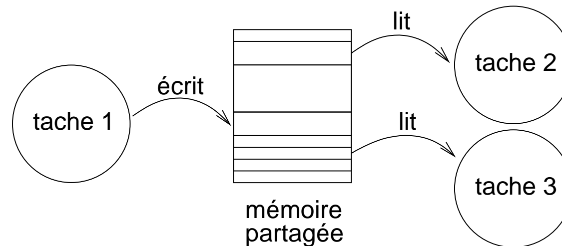
Les sémaphores POSIX sont des sémaphores à compte.

Les sémaphores WIND sont très similaires aux sémaphores POSIX, mais ils offrent les fonctionnalités suivantes :

- héritage de la priorité
- protection contre la destruction d'une tâche possédant un sémaphore mutex
- possibilité, pour une tâche de prendre plusieurs fois le même sémaphore
- prise de possession d'un sémaphore mutex
- butées de temps
- choix du mécanisme de rangement de la file d'attente

Quand une de ces propriétés est importante, les sémaphores WIND sont préférables .

4.4 La mémoire partagée



Un accès concurrent à une mémoire partagée peut demander un accès contrôlé (par sémaphore MUTEX)

4.4.1 Mémoire partagée dans un environnement de processus lourds (shared memory)

```
int shmget(key_t clé, int size, int shmflg) ;
```

`shmget()` renvoie l'identificateur du segment de mémoire partagée associé à la valeur de l'argument clé. Un nouveau segment mémoire, de taille size arrondie au multiple supérieur de `PAGE_SIZE`, est créé si clé a la valeur `IPC_PRIVATE` ou si aucun segment de mémoire partagée n'est associé à clé, et `IPC_CREAT` est présent dans `shmflg`.

Exemple de création et d'accès à une mémoire partagée :

```
#define CLEF 1234
struct MesDonnees {
 int a ;
 float b ;
 char texte[256] ;
} ;
struct MesDonnees * mem ;
int shmid=shmget ((key_t)CLEF, 64, 0750+IPC_CREAT) ;
mem = (struct MesDonnees * ) shmat (shmid, NULL,0) ;
printf ("prod : adr mem = %x\n", mem) ;
sprintf (mem->texte, "%ld",nb) ;
exit (0) ; // détachement
```

4.4.2 Mémoire partagée dans un environnement de processus légers (thread)

4.5 Les signaux

4.5.1 Fonctionnement général des signaux

Les signaux modifient le fil d'exécution d'une tâche. Toute tâche (ou routine d'IT) peut émettre un signal vers une autre tâche . La tâche qui reçoit le signal arrête immédiatement son exécution et exécute la routine d'interception du signal la prochaine fois qu'elle passe en exécution. La routine d'interception s'exécute dans le contexte de la tâche réceptrice du signal et utilise la pile de cette tâche. Cette routine d'interception s'exécute même si la tâche destinataire est bloquée.

Les signaux sont le moyen le plus approprié pour le traitement des erreurs et des exceptions. En général, les routines d'nterception des signaux sont considérées comme des routines d'IT; Une telle routine ne doit donc faire appel à aucune fonction qui risquerait de bloquer. Comme les signaux arrivent de façon asynchrone, il est difficile de prévoir quelles sont les ressources qui sont indisponibles, quand un signal arrive. Pour programmer en toute sécurité, il ne faut appeler que les routines qui peuvent êtres appelées dans une routine d'IT. En tout cas, il faut s'assurer que la routine ne provoquera une situation d'interblocage (deadlock).

Basic Signal Calls (BSD and POSIX 1003.1b)

POSIX 1003.1b Compatible	Call UNIX BSD Compatible Call	Description
signal( )	signal( )	Specify the handler associated with a signal.
kill( )	kill( )	Send a signal to a task.
raise( )	N/A	Send a signal to yourself.
sigaction( )	sigvec( )	Examine or set the signal handler for a signal
sigsuspend( )	pause( )	Suspend a task until a signal is delivered.
sigpending( )	N/A	Retrieve a set of pending signals blocked from delivery.
sigemptyset( ) sigfillset( ) sigaddset( ) sigdelset( ) sigismember( )	sigsetmask( )	Manipulate a signal mask.
sigprocmask( )	sigsetmask( )	Set the mask of blocked signals.
sigprocmask( )	sigblock( )	Add to a set of blocked signals.

The colorful name `kill()` harks back to the origin of these interfaces in UNIX BSD. Although the interfaces vary, the functionality of BSD-style signals and basic POSIX signals is similar.

```
kill [-signum] pid
```

Par de nombreux aspects, les signaux ressemblent aux interruptions matérielles. Les signaux de base sont au nombre de 31. Une routine d'interception de signal est associée à un signal particulier grâce à la fonction sigvec( ) ou sigaction( ), de la même façon qu'une routine d'IT est associée à un vecteur d'IT, avec intConnect( ). On peut émettre un signal avec la fonction kill( ). cela peut être comparé à l'occurrence d'une demande d'IT. Les fonctions sigsetmask( ) et sigblock( ) ou sigprocmask( ) permet d'ignorer certains signaux spécifiés.

4.5.2 Les signaux prédéfinis (`kill -l`)

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
```

```
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      32) SIGRTMIN    33) SIGRTMIN+1
34) SIGRTMIN+2  35) SIGRTMIN+3  36) SIGRTMIN+4  37) SIGRTMIN+5
38) SIGRTMIN+6  39) SIGRTMIN+7  40) SIGRTMIN+8  41) SIGRTMIN+9
42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14
50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10
54) SIGRTMAX-9  55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6
58) SIGRTMAX-5  59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2
62) SIGRTMAX-1  63) SIGRTMAX
```

Certains signaux sont associés aux exceptions matérielles. Par exemple, les erreurs de bus, instructions illégales, ou les exceptions mathématiques entraînent l'émission de signaux prédéterminés.

4.5.3 Un cas particulier : les alarmes

La fonction *alarm(delai)* permet de demander à recevoir le signal dans secondes

```
void routine (int sig);

int main(void) {
char s[20];
//
signal (SIGALRM, SIG_IGN);
signal (SIGQUIT, routine);
signal (SIGINT, routine);
signal (SIGTERM, routine);
  for (;;) {
    alarm (10); //
    scanf("%s",s);
    alarm(0); //
  }
}

// traitement de l'alarme
void routine (int sig) {
  printf (‘                                  ‘);
  if (sig==SIGTERM)
    exit (0);
  alarm (2);
}
```

4.5.4 Timers et Chiens de garde dans VxWorks

VxWorks inclut un système de gestion des timers chien de garde, qui permet à n'importe quelle fonction C de s'exécuter après un délai spécifié. Les timers chien de garde sont gérés par le système d'interruption de l'horloge système. Normalement, les fonctions appelées par les timers chien de garde, sont exécutées comme faisant partie d'une routine d'IT s'exécutant au niveau d'IT de l'horloge système. Si le noyau est incapable d'exécuter la fonction immédiatement (il est déjà en IT par exemple), la fonction est placée dans la file des activités à mener par la tâche `tExcTask`. et donc bénéficient de sa priorité (usually 0). Les restrictions d'utilisation des routines d'IT s'appliquent également aux fonctions enclenchées par les timers chien de garde. Le tableau suivant montre les fonctions proposées par la bibliothèque `wdLib` .

call	description
wdCreate()	Allocate and initialize a watchdog timer.
wdDelete()	Terminate and deallocate a watchdog timer.
wdStart()	Start a watchdog timer.
wdCancel()	Cancel a currently counting watchdog timer.

A watchdog timer is first created by calling `wdCreate()`. Then the timer can be started by calling `wdStart()`, which takes as arguments the number of ticks to delay, the C function to call, and an argument to be passed to that function. After the specified number of ticks have elapsed, the function is called with the specified argument. The watchdog timer can be canceled any time before the delay has elapsed by calling `wdCancel()`.

Exemple de timer Chien de garde

```
/* This example creates a watchdog timer and sets it to go off in 3 seconds. */
/* includes */
#include "vxWorks.h"
#include "logLib.h"
#include "wdLib.h"
/* defines */
#define SECONDS (3)
WDOG_ID myWatchDogId;
task (void) {
  /* Create watchdog */
  if ((myWatchDogId = wdCreate( )) == NULL)
    return (ERROR);
  /* Set timer to go off in 3 SECONDS - printing a message to stdout */
  if (wdStart (myWatchDogId, sysClkRateGet( ) * SECONDS, logMsg,
  "Watchdog timer just expired\n") == ERROR)
    return (ERROR);
  /* ... */
}
```

4.5.5 L'horloge auxiliaire dans VxWorks

Cette horloge dispose des même fonctionnalités que les timers chiens de garde:

fonction	rôle	exemple
sysAuxClkConnect()	installer une routine périodique	sysAuxClkConnect(routine,0)
sysAuxClkRateSet()	régler la fréquence	sysAuxClkRateSet(500)
sysAuxClkEnable()	activer l'horloge	
sysAuxClkDisable()	arrêter l'horloge	

4.5.6 POSIX Clocks and Timers

A clock is a software construct (`struct timespec`, defined in `time.h`) that keeps time in seconds and nanoseconds. The software clock is updated by system-clock ticks. `VxWorks` provides a `POSIX 1003.1b` standard clock and timer interface. The `POSIX` standard provides for identifying multiple virtual clocks, but only one clock is required–the system-wide real-time clock, identified in the clock and timer routines as `CLOCK_REALTIME` (also defined in `time.h`). `VxWorks` provides routines to access the system-wide real-time clock; see the reference entry for `clockLib`. (No virtual clocks are supported in `VxWorks`.)

The `POSIX` timer facility provides routines for tasks to signal themselves at some time in the future. Routines are provided to create, set, and delete a timer; see the reference entry for `timerLib`. When a timer goes off, the default signal (`SIGALRM`) is sent to the task. Use `sigaction()`to install a signal handler that executes when the timer expires (see Signals).

Exemple : POSIX Timers

```
/* This example creates a new timer and stores it in timerid. */

/* includes */
#include "vxWorks.h"
#include "time.h"

int createTimer (void) {
    timer_t timerid;

    /* create timer */
    if (timer_create (CLOCK_REALTIME, NULL, &timerid) == ERROR)
        {
        printf ("create FAILED\n");
        return (ERROR);
        }

    return (OK);
    }
```

An additional `POSIX` function, `nanosleep()`, allows specification of sleep or delay time in units of seconds and nanoseconds, as opposed to the ticks used by the Wind `taskDelay()` function. Only the units are different, however, not the precision: both delay routines have the same precision, determined by the system clock rate.