Manipulation n°1

1ère partie.

Objectifs :

- Configurer les paramètres de la BOOTROM afin de télécharger VxWorks.
- Configurer le **target server**.
- Reconfigurer VxWorks et tester la modification.

Précisions :

Les cibles seront nommées **Target*n*** avec n = numéro du poste.
Les adresses IP des cibles seront **172.16.40.20x**, ou x est le numéro du poste.
Pour la connexion FTP vous donnerez le nom de la cible "bab40" comme nom d'utilisateur avec le mot de passe **serveurFTP**

Tester le téléchargement et le démarrage de VxWorks. Si celui-ci s'est correctement déroulé la fenêtre de l'hyperterminal doit vous afficher quelques informations, référence CPU, version de VxWorks, date de création et surtout l'information **WDB ready**, spécifiant que l'agent de développement croisé résidant sur la cible est en attente de commandes.
Vous pouvez utiliser le **launcher** pour avoir une fenêtre plus complète d'informations sur la cible.

Fier de ce succès vous pouvez à présent envisager une reconfiguration du noyau en lui ajoutant les fonctionnalités permettant la réalisation d'un «ping » sur le PC.
Redémarrer la cible (sans pour cela effectuer de reset !) et tester cette nouvelle fonctionnalité.

Chacune des précédentes manipulations une fois réalisée avec succès devra faire l'objet d'une constatation par le professeur présent.

2nd partie.

Objectifs :

- Utilisation des commandes du Shell.
- Compiler et télécharger un programme.
- Lancer une tâche, au moins ! Calculer son temps d'exécution.

Evaluation d'expressions numériques.

1. Tester les commandes suivantes, *noter les résultats obtenus et les justifier*.

→ **$28**
→ **var1 = 0x14/(2+3)-1** (allocation dynamique)
→ **var1 = 65** (pas de création de symbole)
→ **var1**
→ ***Adresse de var1**

2. Vérifier l'historique des commandes par la commande d'édition h.

Par défaut les variables entières sont sur 32 bits, utiliser le cast (short) pour affecter une variable 16 bits.

→ **var2 = (short) 0x40**

→ **var2** (interpréter l'affichage retourné)
Quelle est la commande qui permet d'avoir l'affichage de la valeur 0x40 ?
→ **var3 = (float) 2.5**
→ **var3** (interpréter l'affichage retourné)
→ **d** ***Adresse de var3***
Justifier que la valeur retournée précédemment est correcte mais qu'elle peut aussi correspondre à 2.5.
Quelle est la commande qui permet d'avoir l'affichage de la valeur 2.5 ?

Utilisation du Shell pour exécuter des fonctions.

3. Tester la commande suivante :

→ **printf ″la valeur de var1 est %d\n″,var1**
Remarquez que les parenthèses sont optionnelles lorsque la fonction est en début de ligne.
La sortie par défaut se fait sur la liaison série (console ou hyperterminal).

Utilisation du Shell pour télécharger et exécuter un programme.

5. Compiler le fichier **demo.c** et télécharger le module correspondant.

6. Vérifier que le module est chargé à l'aide du Shell et du Browser.

7. Vérifier le fonctionnement de la fonction **demo()** selon les 2 méthodes invoquées dans le cours.

8. Vérifier les tâches en cours à l'aide de la commande i, interpréter.

9. Utiliser la fonction **period()** pour exécuter **demo()** toutes les 3 secondes.

10. Utiliser à nouveau la commande i ; on constate que **period()** « spawn » une tâche avec comme point d'entrée la fonction VxWorks **_periodHost()**.

11. Utiliser la commande **repeat** pour exécuter 3 fois la fonction **demo()**.

12. Mesurer le temps d'exécution de la fonction **demo()**.

13. Utiliser l'outil **spy** pour visualiser l'activité du microprocesseur.

L'ensemble des fonctions testées dans cette dernière partie devra faire l'objet d'un fichier script commenté.

Pour aller plus loin.

Un peu plus difficile, utiliser les commandes du Shell pour lire le fichier **demo.c** sur le disque dur du PC et afficher son contenu sur la console.

Le programme correspondant devra être sauvegardé dans un fichier script commenté.

Manipulation n°2

1ère partie.

Objectif :

- Apprendre à utiliser les commandes de base du débuggeur **CrossWind.**

Compiler et invoquer le débuggeur.

1. Éditer et compiler le fichier test.c.
2. Télécharger le module à "débuggeur", test.o
3. Créer une variable **NOM** et initialiser avec votre nom.
4. Exécuter **Ajout** en lui passant le paramètre **NOM** puis exécuter **Salutation**. Logiquement, vous devez vous faire insulter. Il faut donc "débuggeur" ces fonctions.
5. Lancer le "débuggeur", une fenêtre doit apparaître avec le prompt (gdb). Charger le module test.o. Vous pouvez vérifier les modules trouvés par celui-ci en visualisant la fenêtre du Browser.
6. Placer un point d'arrêt à l'entrée de la fonction Ajout et passez lui le paramètre **NOM,** "spawner" cette tâche. Dérouler la fonction pas à pas et corriger la (ou les) erreur(s).

Remarque : Tout en utilisant le "débuggeur" il est toujours possible d'utiliser les commandes du Shell, par exemple les commandes **i** et **ti** pour obtenir des informations sur les tâches.

Listing du programme test.c :

```
#include "vxWorks.h"
#include "taskLib.h"
#include "stdio.h"

char* Tableau[4] = {"Arthur","Martin","Elise"};

void Salutation(void);
void Ajout(char* nom);

void Salutation(void)
{
int i=1;
     for (i=0;i<10;i++) printf("\n");
     for (i=0;i<=4;i++)
     {
          printf("\tBonjour %s de la part de %s",
                                   Tableau[i],Tableau[4-i-1]);
          printf("\tpassage %x\n",i+1);
     }
}

void Ajout(nom)
char* nom;
{
     Tableau[5] = nom;
}
```

2nd partie.

Objectifs :

- reprendre le programme script de lecture distante du TP1 en langage C dans un projet,
- utiliser "**WindView**" et son "Trigger".

Reconfigurer le noyau et utiliser WindView.

1. Reconfigurer le noyau pour ajouter les fonctionnalités nécessaires à WindView.
2. Télécharger le module **demo.o** déjà utilisé lors du TP n°1.
3. Spawner cette tâche périodiquement.
4. Lancer WindView, l'utiliser pour vérifier l'exécution périodique de la tâche.
5. Créer un nouveau projet nommé "WindProjet" sur votre compte et lui ajouter un fichier C dans lequel vous aller coder :
   - une fonction "**STATUS etatDesTaches( )**" réalisant l'affichage des informations suivantes pour les tâches en cours dans la limite de 10 tâches :
     - ✓ le tid de la tâche en hexadécimal,
     - ✓ le nom de la tâche,
     - ✓ la priorité de la tâche,
     - ✓ l'état de la tâche (suspendue, prête, etc.).
   - cette fonction retournera OK si tout c'est bien passé et ERROR autrement.

Tester et valider cette partie.

6. Pour se "faire la main", tester WindView et son trigger en réalisant l'exemple présenté dans le cours.
7. On veut de la même façon visualiser le déroulement de la fonction **etatDesTaches()**, pour cela ajouter à votre projet une variable globale que vous affectez à 1 au début de la fonction et à 2 avant de sortir. Configurer le trigger en conséquence et relever le cycle avec WindView sur fond blanc (menu WindView->Options). L'imprimer ou le sauvegarder pour une impression future.

Compte-rendu à remettre en début de TP suivant :

- le listing commenté du programme en langage C qui doit gérer les valeurs retournées par les fonctions système,
- les graphiques WindView commentés en relation avec le code du programme.

Manipulation n°3

1ère partie.

Objectifs :

- Manipuler les commandes de création, blocage, mise en état d'attente, mise en mode retardé des tâches.
- Activer deux tâches concurrentes et modifier le mode d'ordonnancement.
- Spécifier le "**time slice**" du noyau.

Activation d'une fonction récursive.

1. Compiler le fichier **recur.c** et télécharger le module correspondant. On peut au passage, vérifier que les variables globales non initialisées sont par défaut à zéro.

```
#include "vxWorks.h"

int count;

void recur (int level)
     {
     if (level > 0)
          {
          count++;
          recur (level - 1);
          }
     }
```

2. Activer une tâche pour la fonction **recur()** et lui passer le paramètre 6, vérifier à l'aide de la variable de comptage **count** que la fonction s'est bien exécutée 6 fois.
3. Créer une tâche qui active indéfiniment la fonction **recur**, toujours avec le paramètre 6. Prendre soin de sauvegarder l'identificateur de cette tâche.
4. Vérifier que la variable de comptage s'incrémente continuellement.
5. Examiner l'état de la tâche à l'aide de la commande **ti**, puis à plusieurs reprises afficher l'état de la pile à l'aide de la commande **tt** et commenter.
6. Vérifier l'usage fait de la pile, noter la taille réservée, la taille utilisée lors de votre appel et l'usage maximum qui en a été fait jusqu'alors.
7. Si vous désirez à tout prix planter le système vous pouvez toujours relancer la fonction **recur** indéfiniment en spécifiant une taille de pile insuffisante, par exemple :
   tid = taskSpawn("tbadStack" , 75 , 0 , 100 , repeatHost , 0 , recur ,100)
   Vérifier alors l'état de la pile, la marge doit être à zéro et le nom de la tâche doit avoir disparu. Il est alors conseillé de "rebooter" la cible !

Activation de 2 tâches concurrentes.

8. Compiler le fichier **recur2.c** et télécharger le module correspondant.

```
#include "vxWorks.h"
#include "taskLib.h"
int count1;
int count2;

void recur1 (int level)
     {
     if (level > 0)
          {
          count1++;
          recur1 (level - 1);
          }
     }

void recur2 (int level)
     {
     if (level > 0)
          {
          count2++;
          recur2 (level - 1);
          }
     }
```

9. Tout comme précédemment, créer une tâche qui active indéfiniment la fonction **recur1()** en sauvegardant son **TID**. Faites de même pour la fonction **recur2()**.
10. Vérifier l'évolution des tâches à l'aide des variables de comptage **count1** et **count2**. Commenter.
11. Augmenter la priorité de la tâche qui ne s'exécute pas afin qu'elle "prenne la main".
12. Repositionner les 2 tâches avec le même niveau de priorité et faire en sorte qu'elles s'exécutent toutes deux.

2nd partie.

Problème.

Supposons qu'une tâche en cours d'exécution soit retardée pour n1 ticks, puis dès le début du délai suspendue pour n2 ticks avec n2 < n1.

13. Quelles sont alors les deux éventualités de déroulement de la tâche ? Proposer une méthode pour déterminer le déroulement obtenu avec VxWorks et la tester.
14. Supposons à présent que l'on dispose de 2 tâches d'un niveau de priorité p1 et de deux autres tâches d'un niveau de priorité p2. On désire que seules les tâches de priorités p1 s'exécutent "simultanément" ; l'utilisation du mode round robin n'est donc pas envisageable. Proposer une solution et la tester en modifiant, par exemple, le fichier **recur2.c**.

Manipulation n°4

1ère partie.

Objectifs:

- Utiliser un sémaphore binaire.
- Mettre en œuvre le mécanisme d'exclusion mutuelle.

Mise en œuvre d'un sémaphore binaire.

1. Créer un sémaphore binaire avec file d'attente des tâches par ordre de priorité et état initial vide. Sauvegarder son identificateur, **semId**.
2. Prendre le sémaphore avec un «time-out» de 2 secondes. Que se passe t-il, expliquer ? Il est possible d'utiliser la fonction **printErrno** pour plus d'informations.
3. Rendre le sémaphore puis, toujours avec le même "time-out" le prendre 2 fois de suite. Il est conseillé d'utiliser le **browser**, menu **Object Information** avec le paramètre **semId** pour obtenir l'état du sémaphore.
4. Compiler le fichier **semB.c** puis télécharger le module correspondant.

```
#include "vxWorks.h"
#include "semLib.h"
#include "stdio.h"
#include "taskLib.h"
void semB (SEM_ID semId)
     {
     FOREVER
          {
          if (semTake (semId, WAIT_FOREVER) == ERROR)
               {
               printErr ("semTake erreur\n");
               return;
               }
     printf("Prise du sémaphore par la tache %s\n",taskName (0));
          }
     }
```

5. Activer une tâche d'identificateur **tid** qui exécute la fonction **semB()** puis rendre le sémaphore **semId**. Expliquer le résultat, quel est l'état de la tâche ?
6. Rendre à nouveau le sémaphore et commenter.
7. Activer une tâche de faible priorité **tidlow** qui exécute également **semB()**. Lorsque vous donnez le sémaphore quelle est la tâche qui s'exécute ? Expliquer.
8. Rendre à nouveau le sémaphore. Quel est à présent l'état du sémaphore ? Vous pouvez pour cela utiliser la fonction **show()**. Libérer les tâches de la file du sémaphore.
9. On suppose que l'on active une tâche avec la priorité 100 initialisant un sémaphore binaire avec l'état initial 0 puis exécutant le code suivant :

```
FOREVER
        {
        semTake ( semId , WAIT_FOREVER ) ;
        printf ( ″ Prise du semaphore \n″) ;
        }
```

Définir le nombre de fois où la phrase "Prise du semaphore" sera affichée pour les 3 scénarios suivants :

→ **repeat (1 , semGive , semId)**
→ **repeat (2 , semGive , semId)**
→ **repeat (3 , semGive , semId)**

Mise en œuvre d'un sémaphore d'exclusion mutuelle MUTEX.

10. Compiler le fichier **semM.c** et télécharger le module correspondant.

```
#include "vxWorks.h"
#include "semLib.h"
#include "stdio.h"
#include "taskLib.h"

void semM (SEM_ID semId)
     {
     if (semTake (semId, WAIT_FOREVER) == ERROR)
          {
          printErr ("Erreur semTake\n");
          return;
          }
     taskSuspend (0);
     semGive(semId);
     }
```

11. Créer un sémaphore MUTEX d'identificateur **semId** avec queue de priorités, sécurité d'inversion et sécurité de suppression.
12. Activer une tâche d'identificateur **tid1** qui exécute la fonction **semM()** en lui passant l'identificateur du sémaphore précédemment créé. Vérifier que celle-ci est suspendue et qu'elle est propriétaire du sémaphore.
13. Activer une seconde tâche d'identificateur **tid2** sur la même fonction et avec le même sémaphore. Justifier l'état de cette tâche.
14. Faire en sorte que la tâche **tid1** se termine et vérifier que la tâche **tid2** prend le sémaphore. Terminer cette seconde tâche et vérifier que le sémaphore n'a plus de propriétaire.

Protection d'une zone critique par sémaphore MUTEX.

15. Compiler le fichier **mutex.c** et télécharger le module correspondant.

```
#include "vxWorks.h"
#include "stdio.h"
#include "taskLib.h"
#include "semLib.h"
int v1, v2, count;
```

```
void mutex1 (void)
     {
     FOREVER
          {
               v1 = count; v2 = count; count++;
          }
     }

void mutex2 (void)
     {
     int same;
     FOREVER
          {
          same = (v1 == v2);
          if (!same)     printf("v1 = %d, v2 = %d\n", v1, v2);
          taskDelay (1);
          }
     }
```

16. Activer une tâche d'identificateur **tid1** qui exécute la fonction **mutex1()**. Vérifier que les variables s'incrémentent.
17. Activer une tâche d'identificateur **tid2** exécutant la fonction **mutex2()**, susceptible d'interrompre régulièrement la fonction **mutex1()**. Exposer le problème rencontré.
18. Supprimer les deux tâches précédentes et modifier le code du fichier **mutex.c** afin de protéger la section critique par sémaphore. Tester votre solution.

2nd partie.

Objectif :

- Comparer les temps d'exécution des sémaphores.

1. En utilisant les fonctions **timexFunc** et **timexN** déterminer les temps d'exécution du couple de fonctions **semTake/semGive** pour un sémaphore MUTEX puis pour un sémaphore binaire sans options.
2. Ecrire un programme permettant de visualiser avec **WindView** les chronogrammes présentés dans le cours mettant en évidence le phénomène d'inversion de priorité lié à l'usage d'un sémaphore MUTEX. On synchronisera l'acquisition à l'aide d'une variable globale.
3. Ecrire un programme permettant de visualiser avec **WindView** les chronogrammes présentés dans le cours mettant en évidence le phénomène de protection contre la suppression sur un sémaphore MUTEX. On synchronisera l'acquisition à l'aide d'une variable globale.

Manipulation n°5

Première partie. Mise en œuvre des outils de communication inter tâches.

Objectifs:

- utiliser une queue de messages,
- utiliser un tube de communication ou "pipe".

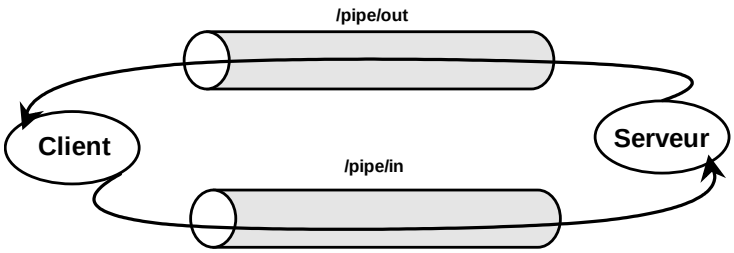
Gestion d'une queue de messages.

1. Créer une queue de messages d'identificateur **msgId** acceptant jusqu'à 20 messages de 100 octets avec gestion des priorités.
2. Ecrire un message dans la queue, de priorité normale et sans attente.
3. Allouer un "buffer" et y stocker le message lu. L'afficher sur le terminal.
4. Envoyer à nouveau le précédent message puis un message différent avec l'option message urgent ; vérifier l'ordre de rangement des messages.
5. Vider la queue de message en vérifiant l'ordre de lecture puis supprimer la queue.
6. Editer et tester un fichier script rassemblant l'ensemble des commandes précédentes.

Gestion d'un tube de communication ou "pipe".

7. Créer un tube de communication acceptant jusqu'à 10 messages de 100 octets. Pour visualiser les circuits ou "devices" du système utiliser la commande **devs**.
8. Ouvrir le tube de communication en mode lecture/écriture puis y écrire un message.
9. Allouer un "buffer" et y stocker le message lu. L'afficher sur le terminal.
10. Activer une tâche qui lit à nouveau le tube, commenter l'état de cette tâche.
11. Ecrire un nouveau message dans le tube et afficher le message lu.
12. Editer et tester un fichier script rassemblant l'ensemble des commandes précédentes.

Relation client/serveur utilisant un tube de communication.



La tâche serveur consiste simplement à répéter indéfiniment la lecture du tube "**/pipe/in**" et à recopier le message lu dans le tube "**/pipe/out**". Cette tâche a pour point d'entrée le fichier "**copyMsg.c**" décrit ci-dessous.

13. Compiler le fichier "**copyMsg.c**" et télécharger le module correspondant.

```
#include "vxWorks.h"
#include "ioLib.h"

void copyMsg (int inFd, int outFd)
     {
     char buf[100];
```

```
int nBytes;
while ((nBytes = read (inFd, buf, sizeof (buf))) > 0)
     write (outFd, buf, nBytes);
}
```

14. Créer deux tubes de communication de 10 messages chacun, un message pouvant accepter au plus 100 octets.
15. Ouvrir les tubes en mode lecture/écriture. Le premier utilisé par le serveur en lecture initialisera un descripteur de fichier **fdin** et le second utilisé pour écrire initialisera un descripteur de fichier **fdout**.
16. Activer une tâche exécutant la fonction "**copyMsg()**" puis envoyer un message au serveur.
17. Lire le message retourné par le serveur et l'afficher. Supprimer la tâche.
18. Editer et tester un fichier script rassemblant l'ensemble des commandes précédentes.

Seconde partie. Résolution d'un problème, faire exécuter à une tâche A une fonction transmise par une tâche B.

Le problème consiste à créer une tâche de faible priorité qui s'exécute lorsqu'il n'y a rien d'autre à faire. Cette tâche doit lire un message dans un tube, le message étant constitué d'une structure contenant deux éléments :

- l'adresse d'une fonction à exécuter,
- un argument pour la fonction à exécuter.

La structure à déclarer nommée "**MSG_REQUEST**" est la suivante :

```
typedef struct
        {
                VOIDFUNCPTR routine ;
                int  argument ;
        }MSG_REQUEST ;
```

1. Ecrire les trois fonctions suivantes dans un fichier prob_pipe.c :

- **int serveurStart(char* tube)** : cette fonction doit créer le tube de communication, ouvrir ce tube et activer une tâche exécutant la fonction serveur, elle retournera le tID de la tâche serveur créée.
- **serveur(int fdtube)** : cette fonction doit lire le tube dans une boucle afin d'en extraire les éléments de la structure et d'exécuter la fonction passée avec son argument.
- **serveurSend(char* tube, VOIDFUNCPTR f, int a)** : cette fonction doit initialiser une structure **MSG_REQUEST** et recopier celle-ci dans le tube.

2. Tester vos fonctions avec les commandes suivantes par exemple :

→ serverStart("/monTube")
→ serveurSend("/monTube", printf , ″ message\n″)
→ serveurSend("/monTube",@reboot , 0)

3. Faire en sorte que l'application serveur s'arrête si l'on lui transmet un message contenant NULL comme adresse de fonction, on prendra soin dans ce cas de fermer le descripteur sur le tube, de détruire le tube puis de faire en sorte que la tâche serveur se termine. Vous pouvez à cette fin modifier les prototypes des fonctions si nécessaire.

Manipulation n°6

Objectif :

- Utiliser les fonctionnalités de la librairie **selectLib** pour bloquer une tâche sur plusieurs descripteurs de fichiers, avec ou sans "time-out".

Précisions :

La librairie **selectLIb** fournit quelques "macro-fonctions" utiles pour la manipulation des bits de la structure **fd_set** du descripteur de fichiers, ces macro-fonctions sont rappelées ci-dessous :

FD_SET (fd , &fdSet) : force à 1 le bit qui correspond au descripteur **fd**,
FD_CLR (fd , &fdSet) : force à 0 le bit qui correspond au descripteur **fd**,
FD_ISSET (fd , &fdSet) : retourne une valeur positive si le bit correspondant au descripteur **fd** est à 1 sinon retourne 0,
FD_ZERO (&fdSet) : force à 0 tous les bits de la structure **fdSet**. (utilise la fonction **bzero**).

Première partie. Gestion d'une attente multiple sans time-out.

Ecrire une fonction **test_select()** en langage C réalisant :

- ✓ la création de 2 tubes **/pipe/1** et **/pipe/2,**
- ✓ l'ouverture des tubes et sauvegardant les descripteurs correspondant sous les noms fd1 et fd2,
- ✓ l'initialisation de deux pointeurs sur des chaînes de caractères initialisées selon votre intuition,
- ✓ l'allocation mémoire nécessaire à une structure de type **fd_set** pointée par **pReadFd** et positionnant à zéro tous les bits de cette structure,
- ✓ le positionnement à 1 les bits de la structure **fd_set** correspondant aux descripteurs fd1 et fd2,
- ✓ l'attente dans une boucle infinie sur la fonction **select()** en lecture des deux tubes avec un time-out infini,
- ✓ l'affichage du contenu du ou des tubes prêts.

Ecrire une fonction **Ecritube()** en langage C à laquelle on passe 3 arguments :
un premier argument prenant la valeur 1 si l'on veut écrire dans le tube 1, deux si l'on veut écrire dans le tube 2 et 3 pour écrire dans les deux tubes,
un second argument pointant sur la chaîne de caractères destinée au tube 1,
un troisième argument pointant sur la chaîne de caractères destinée au tube 2.
Ecrire une fonction **deletetube()** de destruction des précédents tubes.

Tester et faire valider l'ensemble de cette première partie.

Seconde partie. Gestion d'une attente multiple avec time-out.

Modifier le programme précédent (fonction **test_select()**) afin de se bloquer sur **select()** avec un "time_out" de 15 secondes. Lors du déblocage il faudra faire en sorte de :

- ✓ afficher le contenu du ou des "pipe(s)" prêt(s) si la libération est due à l'écriture dans l'un et/ou l'autre d'entre eux,
- ✓ afficher un message "sortie par time_out" si c'est le cas.

Tester et faire valider cette seconde partie.

Troisième partie. Gestion d'une attente multiple avec time-out et gestion du clavier.

Modifier le programme précédent (fonction **test_select()**) afin de lire également les données issues du tube connecté au clavier de la cible.

Tester et faire valider cette troisième partie.

Manipulation n°8

1ère partie.

- Installer une capture de signal permettant à une tâche de sortir élégamment suite à une exception.

Activation d'un signal d'erreur.

Pour cela on va utiliser le fichier **prob_pipe.c** développé lors de la manipulation n°8.

1. Télécharger le module **prob_pipe.o** et exécuter **serveurStart().**
2. Exécuter **serveurSend()** en passant comme premier élément une adresse impaire, ce qui génère une exception erreur d'adresse avec les processeurs de la famille **68XXX**. Vérifier que la tâche est hors service !

Résolution du problème mis à jour.

1. Inclure au fichier **prob_pipe.c** le fichier d'en-tête signal.h.
2. Ecrire une fonction de traitement du signal capturé, par exemple comme ci-dessous, et la prototyper juste à la suite des inclusions.

```
void monHandler ( int signal )
        {
        logMsg (″ Signal %d recu. Redemarrage du serveur \n″,signal,0,0,0,0,0 ) ;
        taskRestart(0) ;
        }
```

3. Installer les fonctions de capture des signaux suivants :
   - **SIGBUS** : Erreur d'adresse
   - **SIGILL** : Instruction illégale
   - **SIGSEGV** : Erreur de bus
4. Tester votre programme ainsi modifié. Vérifier que la tâche « serveur » est toujours présente et fonctionnelle.

2nd partie.

- Utiliser un « watchdog timer ».
- Utiliser l'horloge auxiliaire pour exécuter périodiquement une fonction sous interruption.

Installation d'un « watchdog ».

1. Créer un watchdog d'identificateur **wdId**. Démarrer le watchdog de façon à ce qu'il exécute la fonction logMsg après 5 secondes.
2. Compiler le fichier **wdTache.c** et télécharger le module correspondant.
3. Créer un sémaphore binaire d'identificateur semId pour une synchronisation.
4. Activer une tâche exécutant la fonction wdTache() avec comme argument le sémaphore précédent.
5. Démarrer un watchdog qui exécute **semGive()** après 5 secondes. Une fois le temps écoulé redémarrer ce watchdog avec un délai très long puis avec un délai très court. Commenter.
6. Supprimer le watchdog.

Exécution d'une fonction périodiquement.

Créer un fichier « **periodicAux.c** » dans lequel vous écrirez les deux fonctions suivantes :

- Une fonction **periodicAux()** à laquelle on passe le paramètre fréquence, et qui :
  - Connecte une fonction **compteur()** à l'horloge auxiliaire.
  - Initialise la fréquence de l'horloge auxiliaire à fréquence.
  - Démarre l'horloge auxiliaire.
- Une fonction **compteur()** qui incrémente une variable globale et qui affiche un message sur la console toutes les secondes.

Manipulation n°7

Gestion d'une ressource partagée

Objectif : mettre en œuvre les sémaphores pour gérer l'accès à un "buffer" de caractères partagé entre un processus lecteur et un processus rédacteur puis entre plusieurs processus lecteur et un processus rédacteur.

Dans un premier temps et afin de mettre en évidence le problème lié au partage d'une ressource dans les applications multitâches les protections ne seront pas gérées. L'application consistera pour l'essentiel à développer trois fonctions :

une fonction d'initialisation **init()** réalisant les initialisations dont le chargement en mémoire RAM, dans un buffer global, d'un fichier "d:\init.txt",

une fonction **redacteur()** réalisant la lecture d'un fichier "d:\init.txt", analysant celui-ci et recopiant son contenu ou une image du contenu dans un buffer global partagé.

une fonction **lecteur()** réalisant la lecture du buffer global partagé et affichant son contenu sur le terminal.

L'ensemble devra être codé dans un projet nommé "**projetZDC**" faisant partie d'un nouvel espace de travail "**WorkspaceZDC**".

Première partie : écriture et test des fonctions sans protection de la ressource partagée.

Cette partie est à traiter dans un unique fichier nommé "AppZDC1.c".

Ecriture de la fonction d'initialisation, **STATUS Init (int ticks, int ts).**

Entrées :

ticks = le nombre de ticks par secondes pour l'initialisation de la base de temps du noyau,
ts = le time slice de l'ordonnanceur en nombre de ticks,

Sortie :

Une variable d'état signifiant si tout s'est déroulé correctement.

Coder la fonction **Init()** réalisant l'initialisation de la base de temps et du time slice de l'ordonnanceur. Tester cette fonction.

Ecriture de la fonction rédacteur, **STATUS redacteur (char* pData, char* nom)**.

Entrées :

pData = pointeur vers le buffer partagé dans lequel la fonction écrit des données,
nom = le nom complet d'un fichier à recopier dans le buffer global.

Sortie :

une variable d'état signifiant si tout s'est bien déroulé.

Coder la fonction **redacteur()** réalisant la lecture du fichier et sa recopie dans le buffer global caractère par caractère. On fera en sorte qu'à chaque activation de la fonction rédacteur une variable booléenne soit complémentée, puis testée ; si elle est vraie le fichier sera copié dans le buffer sans modification et si elle est fausse il sera copié en remplaçant les caractères '*' par des caractères '$'.

Remarque : pour les accès au fichier il est imposé d'utiliser les fonctions ANSI de la librairie ANSIstdio.
Tester cette fonction.

Ecriture de la fonction lecteur, **STATUS lecteur (char* pData).**

Entrée :

pData = pointeur vers le buffer partagé dans lequel la fonction lit des données,

Sortie :

une variable d'état signifiant si tout s'est bien déroulé.

Coder la fonction **lecteur()** effaçant l'écran, positionnant le curseur du terminal sur la cinquième ligne et première colonne puis réalisant la lecture du fichier partagé et sa recopie sur le terminal caractère par caractère.
Remarque : pour les accès au terminal il est imposé d'utiliser les fonctions de la librairie ioLib.
Tester cette fonction.

Ecrire une fonction void demarrer (void) qui :

active une tache de priorité 110 sans option et de taille de pile 5koctets pour chacune des fonctions lecteur() et redacteur().

Tester l'application avec un tick de 10ms et un time slice de 1 tick en activant périodiquement la fonction démarrer. Commenter le problème rencontré. Le mettre en évidence avec WindView en synchronisant de début d'acquisition avant le démarrage de la tâche "rédacteur" et la fin en fin de tâche "lecteur" à l'aide d'une variable globale. On prendra soin de filtrer les événements et les tâches afin de ne garder que ce qui est utile à l'interprétation.

Deuxième partie : écriture et test des fonctions avec protection de la ressource partagée par sémaphore.

Cette partie est à traiter dans un unique fichier nommé "AppZDC2.c".

Copier le fichier "AppZDC1.c" dans "AppZDC2.c" et modifier celui-ci de façon à corriger le problème mis en évidence.

Tester l'application et vérifier que l'accès à la ressource partagée se passe sans problème. Mettre en évidence la protection de l'accès à la ressource partagée avec WindView en synchronisant de début d'acquisition avant le démarrage de la tâche "rédacteur" et la fin en fin de tâche "lecteur" à l'aide d'une variable globale. On prendra soin de filtrer les événements et les tâches afin de ne garder que ce qui est utile à l'interprétation et de faire apparaître les prises et restitution de votre sémaphore.

Troisième partie : Accès simple rédacteur et multiples lecteurs.

Cette partie est à traiter dans un unique fichier nommé "AppZDC3.c".

L'accès à la ressource partagée "d:\partage.txt" est toujours protégée entre tâches "rédacteur" et "lecteur" mais on souhaite qu'il soit possible que plusieurs lecteurs y accèdent quasi-simultanément.

Copier le fichier "AppZDC2.c" dans "AppZDC3.c" et modifier celui-ci de façon à prendre en compte l'accès multiple des lecteurs. La fonction démarrer sera modifiée de façon à activer une tâche "rédacteur" et quatre tâches "lecteur".

Tester l'application et vérifier que l'accès à la ressource partagée se passe sans problème. Mettre en évidence la protection de l'accès à la ressource partagée avec WindView en synchronisant le début d'acquisition avant le démarrage de la tâche "rédacteur" et la fin en fin de tâche "lecteur" à l'aide d'une variable globale. On prendra soin de filtrer les événements et les tâches afin de ne garder que ce qui est utile à l'interprétation et de faire apparaître les prises et restitutions de vos sémaphores.

Manipulation n°7

Problème.

Le problème consiste à créer une tâche de faible priorité qui s'exécute lorsqu'il n'y a rien d'autre à faire. Cette tâche doit lire les éléments d'une structure dans un buffer circulaire simulant le stockage, par l'ensemble des caisses d'un grand magasin, d'informations relatives à chaque transaction.
La structure contient les quatre éléments suivants :

- Le numéro de la caisse.
- Le montant de la transaction.
- La date de la transaction au format jj:mm:aaaa.
- L'heure de la transaction au format hh:mm.

La structure à déclarer est la suivante :

```
typedef struct
        {
        int             caisse ;
        float           montant ;
        char            date[11] ;
        char            heure[6] ;
        }TICKET ;
```

Première partie : Gestion du buffer circulaire.

Le buffer circulaire fait appel aux fonctionnalités de la librairie **rngLib**.

1. Ecrire une fonction **Init_buffer()** qui crée un buffer circulaire à l'état initial vide. Celui-ci doit pouvoir contenir au moins une centaine de tickets.

- Paramètre d'entrée : nombre de tickets enregistrables.
- Paramètre de sortie : identificateur du buffer circulaire.

2. Ecrire une fonction **Ecrit_buffer()** permettant d'écrire le contenu d'une structure de type **TICKET** dans le buffer circulaire à condition que celui-ci dispose de l'espace nécessaire.

- Paramètres d'entrées : Identificateur du buffer circulaire.
  Adresse de la structure à écrire dans le buffer.
- Paramètre de sortie : Le nombre d'octets écrits dans le buffer.

3. Ecrire une fonction **Lit_buffer()** permettant de lire le contenu d'une structure de type **TICKET** dans le buffer circulaire à condition que celui-ci ne soit pas vide !

- Paramètres d'entrées : Identificateur du buffer circulaire.
  Adresse de la structure à remplir.
- Paramètre de sortie : Le nombre d'octets lus dans le buffer circulaire ou –1 s'il n'y a rien à lire.

Deuxième partie : Fonction Client.

1. Ecrire une fonction **Client()** à laquelle on passe les éléments de la structure **TICKET**, qui remplit une structure de ce type et la recopie dans le buffer circulaire.

- Paramètres d'entrées : Identificateur du buffer circulaire.
  1° élément de la structure **TICKET.**
  2° élément de la structure **TICKET**.
  3° élément de la structure **TICKET**.
  4° élément de la structure **TICKET**.
- Paramètre de sortie : Aucun.

Troisième partie : Fonction « CaisseCentrale ».

2. Ecrire une fonction **CaisseCentrale()** qui a pour objectif de lire les éléments de la structure **TICKET** dans le buffer circulaire et affiche les éléments de cette structure sur le terminal.

- Paramètre d'entrée : Identificateur du buffer circulaire.
- Paramètre de sortie : Aucun.

  Cette fonction devra afficher ainsi tous les tickets disponibles dans le buffer circulaire et ne se terminera que lorsque celui-ci sera vide.

Quatrième partie : Test d'ensemble.

1. Ecrire un programme de test nommé **TestCaisse()** transférant un certain nombre de tickets différents dans la structure, puis activant la fonction **CaisseCentrale()**. Faire constater le fonctionnement.
2. Modifier le programme de test précédent afin qu'il stocke continuellement des tickets dans le buffer circulaire avec un délai d'attente entre chaque stockage. Le programme **CaisseCentrale()** sera quant à lui activé par une tâche concurrente à celle exécutant la fonction **TestCaisse()**. Faire constater le fonctionnement.
3. Les fonctions **TestCaisse()** et **CaisseCentrale()** étant activées par deux tâches concurrentes, un problème ne risque t-il pas de se poser ? Ce problème, s'il existe, nécessite-t-il d'ajouter certaines protections ? Expliquer.

Manipulation n°8

Objectifs:

- Ecrire, compiler et tester une classe avec CrossWind, sous TORNADO.

1. Test d'une classe minimale « CuveRect ».

Soit les fichiers **Cuve.h** et **Cuve.cpp** suivants, qui correspondent à l'écriture d'une classe de gestion d'une cuve rectangulaire.

```
/***** DEFINITION DE CLASSE*****/
class CuveRect
{
  double fluidlevel;
  double length, width, height;
  char fluidname [20];
  const double MAX = 10.0;
  const double MIN = 0.0;
  public:
  CuveRect(double length1, double width1, double height1, char fluid[20],
 double fluidlevel1);
  void inclevel(double value);      // augmente le niveau de value unites
  void declevel(double value);      // diminue le niveau de value unites
  void setlevel(double newflulevel);
  void showstatus();
};
```

```
#include "vxWorks.h"
#include "stdio.h"
#include "string.h"
#include "Cuve.h"

CuveRect::CuveRect(double length1, double width1, double height1, char fluid[20],
double fluidlevel)
{
  fluidlevel = fluidlevel1;
  length = length1;
  width = width1;    // cuve rectangulaire
  height = height1;
  strcpy(fluidname, fluid);
  printf("Cuve construite\n");
  printf("Liquide  : %s\n",fluidname); printf("Longueur : %4.1f\n", length);
  printf("Largeur  : %4.1f\n",width);
  printf("Hauteur  : %4.1f\n",height);
  printf("Niveau   : %4.1f\n\n",fluidlevel);
}

//  INCLEVEL : augmente le niveau d'une cuve de value unites
void CuveRect::inclevel(double value) {  }

// DECLEVEL : diminue le niveau d'une cuve de value unites
void CuveRect::declevel(double value) { }

// SETLEVEL : positionne le niveau d'une cuve
```

```
void CuveRect::setlevel(double newflulevel)
{
      fluidlevel = newflulevel;
      printf("Niveau positionne   : %4.1f\n\n", fluidlevel);
}

// SHOWSTATUS : affiche l'etat d'une cuve
void CuveRect::showstatus()
{
  printf("Liquide  : %s\n",fluidname);
  printf("Niveau   : %4.1f\n", fluidlevel);
  printf("Quantite : %4.1f\n\n", length*width*fluidlevel);
}
```

1. Ajouter les fonctions membres permettant d'augmenter et de diminuer le niveau de la cuve d'une valeur passée en argument, le niveau ne devra évidemment pas sortir des limites [MIN,MAX]. Compléter également le code de la méthode **setlevel**.
2. Ecrire un programme permettant d'instancier un objet du type **CuveRect** et de tester ses fonctions membres à l'aide du débuggeur.
3. Modifier le constructeur de l'objet **CuveRect** pour qu'il soit initialisé par défaut avec les valeurs suivantes :
   longueur = 2
   largeur = 2
   hauteur = 8
   liquide = eau
   niveau = 0
   Tester la prise en compte des valeurs par défaut en totalité ou partiellement.
4. Surcharger le constructeur avec un autre modèle de votre choix, puis tester ces deux constructeurs.
5. Ajouter un destructeur à la classe **CuveRect** affichant un message précisant la destruction de l'objet.
6. Ajouter une variable membre statique initialisée à zéro que l'on incrémentera avec le nombre d'octets réservés par chaque création d'objet et que l'on décrémentera d'autant lors de la destruction des objets. Instancier plusieurs objets du type **CuveRect** et vérifier que cette variable statique est commune à tous ces objets.

   Exemple de déclaration (Cuve.h) :

   Static int memory

   Exemple d'affectation (Cuve.cpp) :

   Int CuveRect : :memory=0 ;

7. Vérifier Que la variable **memory** représente bien l'espace mémoire alloué par les objets en instanciant au moins un objet par valeur, par pointeur et par copie. Exposer le problème rencontré avec la création d'un objet par copie, proposer et tester une solution pour corriger le problème.
8. Ecrire une classe nommée **CuveCyl** équivalente à la précédente pour des cuves cylindriques.
9. Modifier les classes **CuveRect** et **CuveCyl** afin qu'elles deviennent des classes dérivées d'une même classe de base nommée **Cuve**. Bien entendu, les variables et fonctions membres communes aux deux classes dérivées devront être exportées dans la classe de base.
10. Si votre analyse objet est correcte vous devez vous retrouver avec une méthode **showstatus** dans chaque classe dérivée, faire en sorte que l'existence de cette méthode soit imposée par la classe de base. Proposer et tester votre solution.

Manipulation n°10 – Mise en œuvre des protocoles réseaux standards de connexion distante

Objectifs :

- Utiliser le protocole **rlogin** pour se connecter à une cible à partir d'un poste **Linux**,
- utiliser le protocole **telnet** pour se connecter à une cible à partir d'un poste **Windows**.

Utilisation du protocole "rlogin".
Dans cette partie vous allez modifier le noyau VxWorks afin de lui ajouter le shell résidant, ceci permettra d'utiliser les commandes du shell, soit directement à partir du clavier du terminal connecté à la cible, soit à partir d'un poste distant connecté à la cible par **rlogin** ou **telnet**.

1. Modifier le projet noyau de façon à lui ajouter le shell résidant ou target shell, inclusion "**INCLUDE_SHELL**". Il sera peut-être nécessaire d'ôter d'autres fonctionnalités du noyau (bibliothèques C++ par exemple) afin que la taille de celui-ci ne dépasse pas 512 koctets.

Pour utiliser les commandes du shell il est également nécessaire que le noyau dispose de la table des symboles du shell, inclusion "**INCLUDE_SYM_TBL**", correspondance entre les noms et les adresses des fonctions de celui-ci. Pour ajouter cette table deux solutions sont envisageables nécessitant en plus de la précédente l'une des inclusions suivantes :

- ✓ soit la table des symboles est ajoutée comme partie intégrante du noyau, inclusion "**INCLUDE_STANDALONE_SYM_TBL**",
- ✓ soit la table des symboles est téléchargée dans un fichier de symboles "**vxworks.sym**" indépendamment du noyau, inclusion "**INCLUDE_NET_SYM_TBL**".

Si l'on désire que le shell résidant dispose également des fonctions permettant de charger/décharger des modules il faut également ajouter les inclusions "**INCLUDE_LOADER**" et "**INCLUDE_UNLOADER**".

2. Modifier le projet noyau de façon à lui ajouter la table des symboles du shell dans un fichier indépendant.
3. Ajouter au noyau le module de gestion du protocole **rlogin**, inclusion "**INCLUDE_RLOGIN**", cette inclusion permettant une connexion distante à la cible non sécurisée, i.e. sans authentification ; puis charger le nouveau noyau.
4. Se connecter à la cible via **rlogin** à partir d'un poste **Linux** et vérifier le fonctionnement du shell résidant à partir de quelques commandes comme **i**, **period**, etc.

Le shell résidant n'est pas multi utilisateurs, il n'est donc pas possible d'obtenir plusieurs sessions **rlogin** concurrentes sur une même cible. De même si l'on utilise le shell à partir du terminal il est nécessaire d'interdire une session **rlogin** afin que celle-ci ne prenne pas la main, bloquant ainsi l'usage du shell depuis le terminal. Pour bloquer le shell à partir du terminal on dispose de la fonction **shellLock** (paramètre =1 pour le verrouillage et 0 pour le déverrouillage).

5. Fermer la session **rlogin** à l'aide de la commande "**logout**" ou "**~.**", puis à partir du terminal verrouiller l'accès au shell et tester son usage.

Il est également possible de sécuriser les connexions **rlogin** à partir d'un nom d'utilisateur et d'un mot de passe encrypté.

6. Ajouter au noyau le module d'authentification, inclusion "**INCLUDE_SECURITY**".

Une fois le nouveau noyau chargé, il est possible de :

- ✓ ajouter un utilisateur avec mot de passe, fonction **loginUserAdd("utilisateur"," mot de passe encrypté")**,
- ✓ visualiser les utilisateurs autorisés, fonction **loginUserShow()**,
- ✓ supprimer un utilisateur, fonction **loginUserDelete("utilisateur"," mot de passe encrypté")**.

Pour obtenir le mot de passe encrypté à utiliser lors de l'ajout d'un utilisateur à partir du mot de passe souhaité, on dispose d'un utilitaire fourni avec tornado2, **vxencrypt.exe**.

7. Ajouter un utilisateur à la cible, le visualiser, et tester une connexion **rlogin** sécurisée.

Utilisation du protocole "telnet".
La prise en compte du protocole telnet nécessite l'inclusion "**INCLUDE_TELNET**" et ne gère pas d'authentification.

8. Modifier le noyau VxWorks afin de supprimer le module d'authentification et le module rlogin puis d'ajouter le module telnet.
9. Se connecter à la cible via **telnet** à partir d'un poste **Windows** et vérifier le fonctionnement du shell résidant à partir de quelques commandes comme **i**, **period**, etc. Faire en sorte de tester si le shell résidant est multi-utilisateurs.
10. Reconstruire le noyau initial après avoir supprimé toutes les inclusions ajoutées au noyau au cours de ce TP.

Manipulation n°11 – Système de fichiers et protocoles réseaux standards de transfert de fichiers

Objectifs :

- créer un RAM disque et y installer un système de fichiers DOS,
- utiliser le protocole FTP pour lire un fichier DOS sur une machine distante et le stocker en RAM disque,
- créer un RAM disque et y installer un système de fichiers RAW (UNIX),
- utiliser le protocole NFS pour lire un fichier UNIX sur une machine distante et le stocker en RAM disque,
- créer un fichier exécutable sur une machine distante via NFS.

Première partie, DOS-FTP.

Création d'un RAM disque avec système de fichiers DOS à l'aide du shell.

Cette partie nécessite de vérifier que le noyau dispose des inclusions nécessaires à la gestion du RAM disque et des systèmes de fichiers DOS.

1. Créer dynamiquement un RAM disque de 1024 octets par blocs, 1 piste de 64 blocs et sans offset.
2. Installer un système de fichiers DOS sur le RAM disque que l'on nommera "**dos:**".
3. Créer un fichier sur ce système que vous nommerez "**ficDOS**", accessible en lecture et en écriture.
4. Vérifier que ce fichier est créé, le tester en y écrivant un message que l'on relira pour l'afficher sur la console. Créer des répertoires avec le shell du PC (commandes **ioctl**) puis avec le shell résidant à la cible (commande **mkdir**) puis les supprimer (commande **rmdir**).

Utilisation du protocole FTP à l'aide du shell.

Cette partie nécessite de modifier le noyau afin de lui ajouter le module "**client FTP**".

5. Visualiser la table des adresses réseau connues de votre cible. Lui ajouter l'adresse d'un poste distant qui vous sera précisée à la demande.
6. Créer un circuit FTP avec authentification (fonction **iam**) puis ouvrir un canal FTP sur le fichier "**General.txt**" préalablement installé dans "**c:\VxWorks**" du poste distant.
7. Lire le contenu de ce fichier afin de le copier dans votre fichier local "**monFichier**" et de l'afficher sur la console. A partir de ce moment le pire est à craindre, veillez à bien protéger votre serveur FTP local qui, je le rappelle, est actif depuis votre première session Tornado !

Les nouvelles commandes utiles à cette partie de TP sont :

- **netDevCreate()**
- **hostShow()**
- **dosFsMkfs()**
- **hostAdd()**

Attention, elles ne sont pas dans l'ordre et en plus il en manque une !

Seconde partie, RAW (UNIX)-NFS.

Pour cette partie il est nécessaire de modifier le noyau VxWorks afin de lui ajouter le shell résidant, ceci permettra d'utiliser les commandes du shell à partir du terminal afin de pouvoir lister le contenu du RAM disque.

Création d'un RAM disque avec système de fichiers RAW à l'aide du shell.

Cette partie nécessite de modifier le noyau afin de lui ajouter la gestion du RAM disque et des systèmes de fichiers RAW.

1. Créer dynamiquement un RAM disque de 1024 octets par blocs, 1 piste de 64 blocs et sans offset.
2. Installer un système de fichiers RAW sur le RAM disque que l'on nommera "**raw**:".

Utilisation du protocole NFS à l'aide du shell.

Cette partie nécessite de modifier le noyau afin de lui ajouter le module "**client NFS**".

3. Visualiser la table des adresses réseau connues de votre cible. Lui ajouter l'adresse d'un poste distant exécutant le service "**serveur NFS**", à savoir "**serveurLinux**", qui vous sera précisée à la demande.
4. Visualiser les répertoires exportés par le poste distant afin de vérifier que votre poste est autorisé à monter un système de fichier NFS.
5. Monter un système de fichier NFS nommé "**/nfs**" sur votre "**homedirectory**" du serveur NFS et vous authentifier auprès de celui-ci à l'aide de vos identificateurs d'utilisateur et de groupe (uid et gid).
6. Lire le contenu d'un fichier de votre choix sur votre compte Linux afin de le copier dans le système de fichier raw local et de l'afficher sur la console.
7. Toujours à partir de votre session NFS, créer un répertoire nommé "**TPvxworks**" à l'aide d'une commande du shell de la cible dans votre "**homedirectory**", puis au sein de ce répertoire :
   - créer un fichier exécutable nommé "**scriptLS**" avec les droits de lecture/écriture/exécution pour le propriétaire,
   - lui ajouter les deux lignes nécessaires pour préciser le programme à appeler en tant qu'interpréteur et que ce script exécute simplement la commande **ls** avec les options **l** et **i**.
8. Tester le fonctionnement de ce script à partir d'une session Linux.

Manipulation n°12 – Mise en œuvre des protocoles réseaux standards DNS et SNTP

Objectifs:

- utiliser les fonctionnalités d'un client DNS pour résoudre un nom et une adresse IP en questionnant un serveur DNS "**serveurii**",
- utiliser les fonctionnalités d'un client SNTP pour obtenir l'heure à partir d'un serveur SNTP "**serveurii**".

Résolution de nom et d'adresse IP à l'aide d'un client DNS (Domain Name System).

Pour traiter cette partie il est nécessaire de reconfigurer le noyau VxWorks en lui ajoutant le module client DNS, inclusion "**INCLUDE_DNS_RESOLVER**". Il faut également configurer les paramètres permettant de spécifier le serveur DNS à utiliser par défaut, à savoir le paramètre "**RESOLVER_DOMAIN_SERVER**" avec une des adresses IP du serveur DNS du réseau, et le paramètre "**RESOLVER_DOMAIN**" avec le nom de domaine du serveur DNS "RAMSES.II".

1. Au sein d'un nouveau projet créer une fonction nommée **printConfigServeurDns()** qui affiche dans le terminal la configuration courante du client DNS relativement au serveur DNS à utiliser.
2. Créer une fonction de test qui utilise la fonction **printConfigServeurDns()** et vérifier que la configuration est correcte.
3. Ajouter à votre projet une fonction nommée **clientDnsHostversIp** qui reçoit en paramètre le nom d'hôte d'un poste et retourne son adresse IP dans sa notation à point. Celle-ci devra respecter le prototype suivant :

   STATUS clientDnsHostversIp(char* host, char* ip)

   Pour coder cette fonction vous aurez besoin d'analyser le contenu de la structure "**hostent**" dont un des champs contient un tableau de pointeurs vers les adresses IP résolues. Ces pointeurs étant du type char*, un changement de type sera nécessaire pour récupérer l'adresse complète stockée sous la forme d'un entier long, celle-ci devant ensuite être convertie dans sa notation à point.
4. Compléter la fonction de test pour qu'elle résolve un nom d'hôte de votre choix et qu'elle affiche celui-ci dans le terminal avec son adresse IP.
5. Ajouter à votre projet une fonction nommée **clientDnsIpversHost** qui reçoit en paramètre l'adresse IP d'un poste dans sa notation à point et retourne son nom d'hôte. Celle-ci devra respecter le prototype suivant :

   STATUS clientDnsIpversHost(char* ip, char* host)
6. Compléter la fonction de test pour qu'elle résolve une adresse IP de votre choix et qu'elle affiche celle-ci dans le terminal avec son nom d'hôte.

Lecture de l'heure par interrogation d'un serveur SNTP (Simple Time Network Protocol).

Pour traiter cette partie il est nécessaire de reconfigurer le noyau VxWorks en lui ajoutant le module client SNTP, inclusion "**INCLUDE_SNTPC**".

7. Au sein du précédent projet ajouter un nouveau fichier de code source et créer une fonction nommée **getTimeSNTP()** qui retourne l'heure courante au format "HH:MM" dans une chaîne de caractères après interrogation d'un serveur SNTP avec un "time-out" donné. Celle-ci devra respecter le prototype suivant :

   STATUS getTimeSNTP(char* serveurSNTP, char* time, int timeout).

   La date et l'heure peuvent être formatées à l'aide de la fonction **ansi strftime()** qui attend entre autres paramètres l'adresse d'une structure **tm**, structure contenant divers champs avec les secondes, l'heure, etc (voir la librairie **ansiTime**).

   L'heure et la date peuvent également être représentées au sein d'une structure **timespec** contenant le temps écoulé depuis la date de référence du 01 janvier 1970 à 0heure 0mn GMT. La conversion des données de temps d'une structure **timespec** vers une structure **tm** se fait grâce à la précieuse fonction **ansi gmtime().**
8. Créer une fonction de test qui utilise la fonction précédente et affiche l'heure GMT du serveur sur le terminal.
9. Ajouter une fonction nommée **getDateSNTP()** qui retourne la date courante au format "jj/mm/aaaa" (par exemple "01/04/2003") dans une chaîne de caractères après interrogation d'un serveur SNTP avec un "time-out" donné. Celle-ci devra respecter le prototype suivant :

   STATUS getDateSNTP(char* serveurSNTP, char* date, int timeout).
10. Compléter la fonction de test pour également afficher la date.

Manipulation n°14

Objectifs:

- Programmation client/serveur avec les « sockets » UDP.
- Programmation client/serveur avec les « sockets » TCP.
- Développer un serveur TCP concurrent.

1. Ecrire un client VxWorks avec le protocole UDP.

Une application serveur UDP est installée sur une cible VxWorks à une adresse fournie lors de la manipulation. Le fichier correspondant, **vxServeurUDP.c**, est présenté ci-dessous :

```
/* Serveur UDP
Affiche l'adresse IP, le numero de port et le contenu du message
Retourne un accuse de reception */

#include "vxWorks.h"
#include "sockLib.h"
#include "sys/socket.h"
#include "netinet/in.h"
#include "inetLib.h"
#include "ioLib.h"
#include "string.h"
#include "stdio.h"
#include "taskLib.h"

char  buff[100];
typedef int SOCK_FD;

LOCAL void error (char * str);

void vxServeur (u_short port)
     {
     int clientAddrLength;
     SOCK_FD sockFd;
     struct sockaddr_in clientAddr;
     struct sockaddr_in srvAddr;
     char inetAddr[INET_ADDR_LEN];
     u_short clientPort;
     char *reply = "Accuse de reception du serveur\n";

     routeAdd("10.0.2.0","10.0.1.200") ;
     clientAddrLength = sizeof (clientAddr);

     /* Creation de la socket */
     if ( (sockFd = socket (PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
          error ("Probleme Socket");

     bzero ((char *)&srvAddr, sizeof(srvAddr));
```

```
     srvAddr.sin_family = AF_INET;
     srvAddr.sin_port = htons(port);
     srvAddr.sin_addr.s_addr = INADDR_ANY;
     if (bind (sockFd, (struct sockaddr *) &srvAddr, sizeof(srvAddr))
< 0)
          {
          close (sockFd);
          error ("Probleme Bind");
          }
     FOREVER
          {
          if (recvfrom (sockFd, buff, 100, 0,(struct sockaddr *)
&clientAddr, &clientAddrLength) < 0)
               {
               close (sockFd);
               error ("Probleme Recvfrom");
               }
          inet_ntoa_b (clientAddr.sin_addr, inetAddr);
          clientPort = ntohl (clientAddr.sin_port);
          printf ("Message recu du client "(port=%d,inet=
%s):\n",clientPort, inetAddr);
          printf (buff);
          if (sendto (sockFd, reply, strlen(reply) + 1, 0,(struct
sockaddr *) &clientAddr, clientAddrLength) < 0)
               {
               close (sockFd);
               error ("Probleme sendto");
               }
          }
     }

void error (char * str)
     {
     perror (str);
     exit (1);
     }
```

1. Pour accéder au poste distant il est tout d'abord nécessaire d'initialiser une structure **sockaddr_in** avec les paramètres de la liaison. Initialiser cette structure et vérifier que son initialisation est correcte. Vous disposez pour cela de deux fonctions dans le fichier **initSockAddr.c**. Remarquez cependant que vous n'êtes en rien obligé de les utiliser !

```
#include "vxWorks.h"
#include "fioLib.h"
#include "hostLib.h"
#include "inetLib.h"
#include "netinet/in.h"
#include "stdio.h"
#include "string.h"
#include "sys/socket.h"
#include "sys/types.h"
```

```
void sockAddrShow (struct sockaddr_in * pSockAddr)
     {
     printf ("sin_family  = %d\n", pSockAddr->sin_family);
     printf ("sin_addr = %s\n",
             inet_ntoa(pSockAddr- >sin_addr.s_addr )
     printf ("sin_port    = %d\n", ntohs (pSockAddr->sin_port));
     }

STATUS inetAddrInit (struct sockaddr_in * pAddr,char * pHost,int port)
    {
u_long inet;
    bzero ((char *) pAddr, sizeof (struct sockaddr_in));
    if ((inet = hostGetByName (pHost)) == ERROR)
     {
      if ((inet = inet_addr (pHost)) == ERROR) return (ERROR);
     }
    pAddr->sin_family         = AF_INET;
    pAddr->sin_port           = htons (port);
    pAddr->sin_addr.s_addr    = inet;
    return (OK);
    }
```

2. Envoyer un message au serveur et lire sa réponse à partir de WindShell en utilisant les commandes :
   - **Socket.**
   - **Sendto.**
   - **Read.**

Il est recommandé d'écrire un fichier de « script » que l'on exécutera à partir du Shell.

3. Ecrire le programme C du client correspondant à la séquence précédente que l'on nommera **vxClientUDP.c**.

2. Ecrire un client VxWorks avec le protocole TCP.

Une application serveur TCP est installée sur une cible VxWorks à une adresse fournie lors de la manipulation. Le fichier correspondant, **vxServeurTCP.c**, est présenté ci-dessous :

```
#include "vxWorks.h"
#include "sockLib.h"
#include "sys/socket.h"
#include "netinet/in.h"
#include "inetLib.h"
#include "ioLib.h"
#include "string.h"
#include "stdio.h"
#include "taskLib.h"
#include "routeLib.h"

#define MAX_MSG_SIZE 80
#define LOCAL  static
```

```
typedef int SOCK_FD;

LOCAL void doRequest ();
LOCAL void error ();

void vxServeur (u_long port)
     {
     int  clientAddrLength;
     SOCK_FD   sockFd;
     SOCK_FD   newSockFd;
     struct sockaddr_in  clientAddr;
     struct sockaddr_in  srvAddr;
     clientAddrLength = sizeof (clientAddr);
     routeAdd("10.0.2.0","10.0.1.200");

     if ( (sockFd = socket (PF_INET, SOCK_STREAM,0)) < 0 )
          error ("Probleme socket");
     bzero ((char *)&srvAddr, sizeof (srvAddr));
     srvAddr.sin_family= AF_INET;
     srvAddr.sin_port= htons (port);
     srvAddr.sin_addr.s_addr = INADDR_ANY;

     if (bind (sockFd, (struct sockaddr*)&srvAddr, sizeof(srvAddr)) <
0)
          {
          close (sockFd);
          error ("probleme Bind");
          }

     /* Queue pour une demande uniquement */
     if (listen (sockFd, 1) < 0)
          {
          close (sockFd);
          error ("probleme Listen");
          }

     FOREVER
          {
          newSockFd = accept (sockFd,(struct sockaddr *) &clientAddr,
&clientAddrLength);
          if (newSockFd < 0)
               {
               close (sockFd);
               error ("Probleme Accept");
               }
          doRequest (newSockFd, &clientAddr);
          printf ("Fermeture de la connexion par le client\n\n")
          close (sockFd);
          exit (0);
          }
     }

LOCAL void doRequest (sock, pClientAddr)
     SOCK_FD sock;
```

```
     struct sockaddr_in * pClientAddr;
     {
     char* reply="Accuse de reception du serveur\n";
     char      buf [MAX_MSG_SIZE];
     int  msgSize;
     char *    pClientInet;
     u_short   clientPort;
     pClientInet = inet_ntoa(pClientAddr->sin_addr);
     clientPort     = ntohs(pClientAddr->sin_port);
     printf  ("Client  connecte  depuis  l'adresse  IP  %s,  et  le  port
%d\n",
               pClientInet, clientPort );

     FOREVER
          {
          msgSize = read (sock, buf, MAX_MSG_SIZE - 1);
          if (msgSize < 0)
               {
               close (sock);
               error ("Probleme Read");
               }
          else if (msgSize == 0)
               {
               close (sock);
               break;
               }
          else
               {
               printf ("Ceci est votre message de %d octets: \n%s\n",
msgSize, buf);
               if (write (sock, reply, strlen(reply)) < 0)
                    {
                    close (sock);
                    error ("Probleme Write");
                    }
               }
          }
     }
LOCAL void error (pStr)
     char * pStr;
     {
     perror (pStr);
     exit (1);
     }
```

1. Envoyer un message au serveur et lire sa réponse à partir de WindShell en utilisant les commandes :

- **socket.**
- **connect.**
- **write.**
- **read.**

Il est recommandé d'écrire un fichier de « script » que l'on exécutera à partir du Shell.

2. Ecrire le programme C du client correspondant à la séquence précédente que l'on nommera **vxClientTCP.c**.

3. Ecrire un serveur concurrent.

La question est simple, la réponse l'est peut-être un peu moins ; il s'agit de faire muter le serveur séquentiel **vxServeur_TCP.c** vers un serveur concurrent que l'on nommera **conServeur.c**. Pour tester celui-ci vous pouvez soit écrire un script utilisant les fonctions **built-in** du Shell soit écrire un programme c à votre convenance.

4. Portage du serveur TCP concurrent sur plateforme Windows.

Pour ceci nous allons utiliser la librairie **Winsock** (librairie **wsock32.dll**) qui est la librairie de plus bas niveau permettant la programmation TCP/IP dans les applications windows.

L'initialisation de Winsock

Pour pouvoir faire des appels à **Winsock**, l'application doit initialiser cette bibliothèque dans sa fonction membre **InitInstance()**. Cette initialisation est réalisée, pour une librairie version n°2, par les lignes de code suivantes :

```
WSADATA wsd;
WORD wVersion;
wVersion=MAKEWORD(2,2);
int error = WSAStartup(wVersion, &wsd);
```

Installation d'un "Thread" de service

Pour permettre la gestion concurrente de plusieurs connexions "simultanées" à des clients nous allons devoir gérer celles-ci à l'aide de "Threads" de service activés à partir du "Thread" principal de l'application ou processus.

Un programme en cours d'exécution (processus) peut contenir plusieurs chemins d'exécution appelés "threads" ou unités d'exécution. Ils sont gérés par le système d'exploitation et possèdent chacun leur propre pile.
La bibliothèque MFC distingue deux sortes de "threads" ; les "threads" de service et les "threads" utilisateur.
Le "thread" de service (le plus utile) ne gère pas de fenêtre et n'a donc pas besoin de traiter les messages.
Le "thread" utilisateur dispose de fenêtres et a donc sa propre boucle de messages.

Nous allons traiter la connexion à chaque client par un "thread" de service. Ainsi, la prise en compte des connexions par l'application sera toujours possible. En effet, les différents "threads" s'exécutent indépendamment, en fonction de leurs priorités respectives.

Exemple :

Pour activer un "thread" de service exécutant une fonction nommée **ThreadServ()** on fait appel à la fonction **AfxBeginThread()** de la classe **CWinThread**. Il est également possible de le suspendre et de le restaurer à l'aide des fonctions **SuspendThread()** et **ResumeThread().**

CWinThread* pThread=AfxBeginThread (ThreadServ,
GetSafeHwnd(), THREAD_PRIORITY_NORMAL, 0, 0);

L'instruction précédente démarre un "thread" de service pour la fonction **ThreadServ()** avec comme paramètre passé le Handle de fenêtre du "thread" principal, une priorité normale, la taille de pile identique au "thread" principal, un démarrage immédiat du "thread" et des attributs de sécurité identiques au "thread" principal. La fonction **ThreadServ()** doit être prototypée comme suit :

UINT ComputeThreadServ(LPVOID pParam)

Le paramètre passé à la fonction est le Handle de la fenêtre du "thread" principal, ceci permettra au "thread" de service, par exemple, de poster des messages vers le "thread" principal.

1. Créer un projet du type simple document sans les options d'impression et avec l'option de gestion des sockets.
2. Ajouter au menu de votre application une rubrique "démarrage serveur" et lui associer une fonction membre de la classe vue.
3. Coder la précédente fonction de façon à ce qu'elle installe une interface logicielle "socket" d'écoute avec le numéro de port 6000 ; qu'elle s'attache à cette "socket" et qu'elle crée une liste d'attachement à la "socket maître" de taille égale à 5 et qu'elle active un "Thread" de service exécutant une fonction globale nommée **serverThreadProc()** en lui passant comme paramètre le "handle" sur la fenêtre courante.
4. Coder la fonction **serverThreadProc()** de façon à ce qu'elle :
    - ✓ se positionne en attente d'une connexion, et si une connexion est activée,
    - ✓ active un nouveau "Thread" de service pour traiter une éventuelle autre connexion,
    - ✓ lise la requête du client, l'affiche, par exemple à l'aide d'une "messageBox" et retourne au client une réponse.

Manipulation n°XXX

Objectifs:

- Reconfigurer VxWorks pour le faire démarrer sur une application.
- Démontrer que la formation a été utile et développer un driver en toute autonomie !

1. Démarrer VxWorks sur une application.

La question tient en une phrase, reprendre une des applications précédentes brillamment développées, par exemple le serveur concurrent, et faire en sorte qu'après téléchargement du noyau le système démarre sur cette application.
Il peut-être utile d'ajouter un petit message terminal au démarrage de l'application afin d'être sûr que celui-ci s'est correctement déroulé.

2. Développer un driver réactif, et utile !

Nous disposons actuellement, nos moyens étant très limités, de 4 pauvres cartes d'interface « cartes VMOD » et d'aucun driver pour les piloter…
Les 4 cartes sont les suivantes :

- Une carte d'entrées sorties 20 ports optocouplés avec trois compteurs 16 bits **WMOD-TTL/O**.
- Une carte CNA 2 voies 12 bits, **VMOD 12A2.**
- Une carte CAN 8/16 voies 12 bits, **VMOD 12E8**.
- Une carte de commande de moteurs à courant continu ou pas à pas, **VMOD MTC.**

Il est bien entendu que la documentation complète relative à chaque carte est à votre entière disposition.
Aussi, en cette période de fêtes, et donc de don de soi, il vous est imploré de vous répartir les modules afin de développer un driver pour celle-ci et bien sûr de le tester. Merci beaucoup pour cette moindre contribution à la survie du secteur Informatique Industrielle salle 230.

Manipulation n°XX

Résolution du problème des philosophes.

Objectif:

- Mettre en œuvre au sein d'un même problème le sémaphore MUTEX pour gérer une ressource partagée et le sémaphore binaire pour gérer des synchronisations entre tâches.

Rappel des données du problème.

Cinq philosophes se retouvent autour d'une table ronde et disposent chacun d'une assiette et d'une fourchette. Le repas servi nécessite deux fourchettes pour pouvoir manger proprement, il est donc impossible que tous mangent simultanément.
Devant cette situation le protocole commun adopté est le suivant :

- Un philosophe ne peut emprunter que la fourchette située à sa droite ou à sa gauche.
- Initialement tous les philosophes pensent.
- Un philosophe se trouve toujours dans l'une des trois situations suivantes :
  - ✓ Pense (avant et après avoir mangé).
  - ✓ Mange (dès lors qu'il a pu obtenir 2 fourchettes)
  - ✓ Affamé (s'il a cherché à obtenir deux fourchettes sans succès)

Solution proposée.

La solution proposée (mais pas obligatoire) attribue une variable d'état à chaque philosophe pouvant prendre les valeurs suivantes :

- Etat[i] = 0 si le philosophe i pense.
- Etat[i] = 1 si le philosophe i mange.
- Etat[i] = 2 si le philosophe i est affamé.

L'accès à la ressource partagée (fourchettes) nécessite l'emploi d'un unique sémaphore MUTEX.
La synchronisation entre philosophes, attente que l'un de ceux qui mangent pose ses fourchettes pour pouvoir manger à son tour, nécessite l'emploi d'un sémaphore binaire de synchonisation par philosophe.

L'algorithme proposé page suivante décrit la tache philosophe faisant évoluer l'état de celui-ci.

```
Philosophe ( i )
        {
        Tant que (rien)
                {
```

```
        penser
        prendre la ressource partagée ( fourchettes)
        si ( Etat[philosophe de gauche]≠mange) et ( Etat[philosophe de droite]≠mange)
                {
                Le philosophe (i) mange ( Etat[i] = mange )
                Donner le sémaphore de synchonisation du philosophe (i)
                }
        sinon   Le philosophe (i) est affamé ( Etat[i] = affamé )
        rendre la ressource partagée ( fourchettes )
        prendre le sémaphore de synchronisation du philosophe (i)
        penser
        prendre la ressource partagée ( fourchettes)
        si (Etat[philosophe de gauche]=affame) et (Etat[philosophe de gauche+1]≠mange)
                {
                Le philosophe de gauche  mange ( Etat[i+1] = mange )
                Donner le sémaphore de synchonisation du philosophe de gauche
                }
        si (Etat[philosophe de droite]=affame) et (Etat[philosophe de droite+1]≠mange)
                {
                Le philosophe de droite mange ( Etat[i-1] = mange )
                Donner le sémaphore de synchonisation du philosophe de droite
                }
        rendre la ressource partagée ( fourchettes )
        }
Fin de tant que
}
```

Analyse de la solution proposée.

1. Justifier le rôle du sémaphore MUTEX ainsi que les emplacements choisis pour les primitives prendre et rendre.
2. Justifier le rôle des sémaphores binaires ainsi que les emplacements choisis pour les primitives prendre et rendre.

Vérification de la solution proposée.

1. Ecrire une fonction Initialisation () qui initialise les sémaphores nécessaires à l'application et active une tâche Philo'i' par philosophe (celle-ci devra avoir pour point d'entrée la fonction philosophe précédemment décrite).
2. Ecrire la fonction Philosophe(i) correspondant à la solution proposée ou à une autre solution de votre choix.
3. On désire qu'à la création de chaque tâche philosophe, le message 'Passage à table du philosophe i ' soit envoyé sur la console. Proposer une solution en utilisant les primitives « Hook ».
4. Ajouter à la fonction philosophe(i), aux endroits appropriés, la fonction d'affichage de l'état des philosophes suivante :

   logMsg('Etat des philosophes %d %d %d %d %d\n',Etat[0], Etat[1], Etat[2], Etat[3], Etat[4]);
5. Tester et faire valider votre travail