Le système embarqué temps réel VxWorks

Frank Singhoff

Bureau C-203

Université de Brest, France

LISyC/EA 3883

singhoff@univ-brest.fr

Sommaire

Présentation

- **Caractéristiques des systèmes embarqués temps réel et objectifs :**

1. Comme tous systèmes temps réel : déterminisme logique, temporel et fiabilité.
2. Mais en plus :
   - Ressources limitées (mémoire [a], vitesse processeur, énergie).
   - Accessibilité réduite.
   - Autonomie élevée.
   - Interaction avec son environnement (capteurs).

$\Longrightarrow$ **Environnements d'exécution spécifiques.**

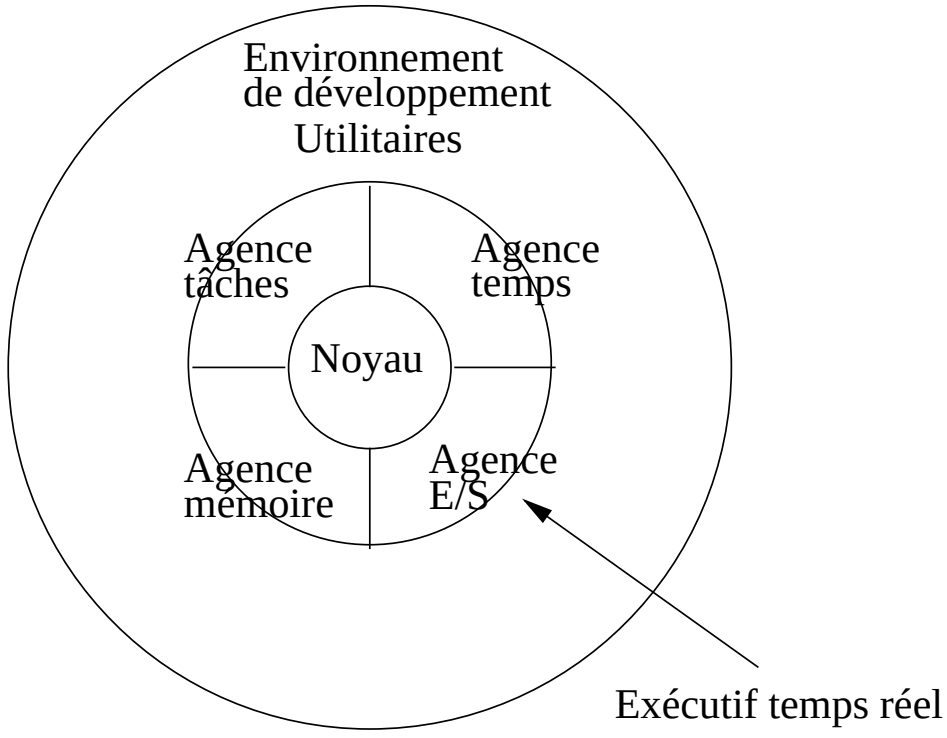
[a] *footprint* ou empreinte mémoire.

Exécutif : architecture et services (1)

- **Caractérisques de l'exécutif :**
  - Modulaire et de petite taille. Flexible vis-à-vis de l'application.
  - Accès aisé aux ressources physiques.
  - Abstractions adaptées (parallélisme, exception, interruption, tâches, ...)
  - Support de langages pour le temps réel (ex : C, Ada).
  - Livré avec ses performances temporelles.

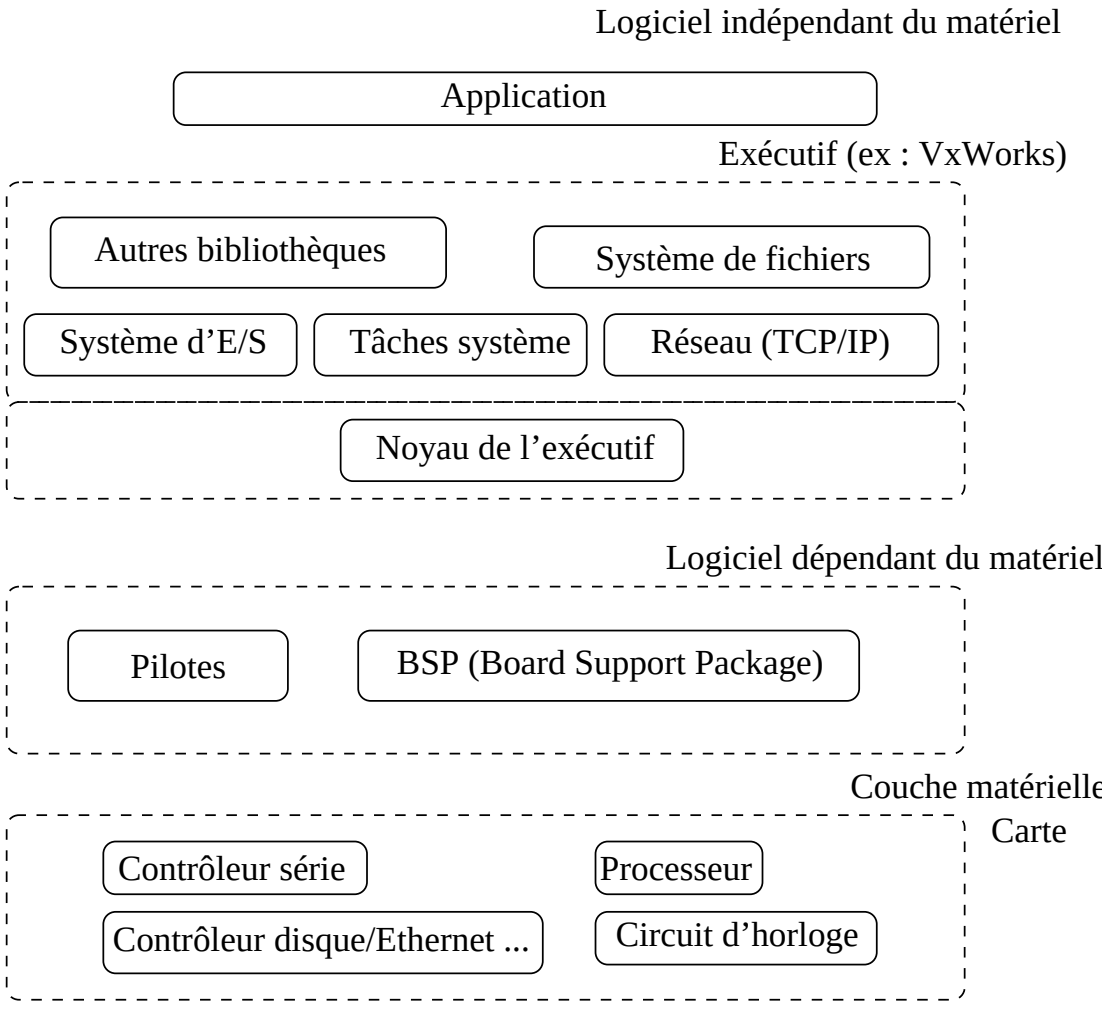
Exécutif : architecture et services (2)

- **Services offerts par un système à base d'exécutif :**



- Notion de noyau et d'agence $\Longrightarrow$ modularité.
- Deux niveaux principaux : exécutif et environnement de développement/mise au point.
- Outils : "cross-compilateur", chargeur de code, outils de mise au point, simulateur, etc.
- Deux phases : Phase de développement/test, phase de mise en exploitation.

Exécutif : architecture et services (3)



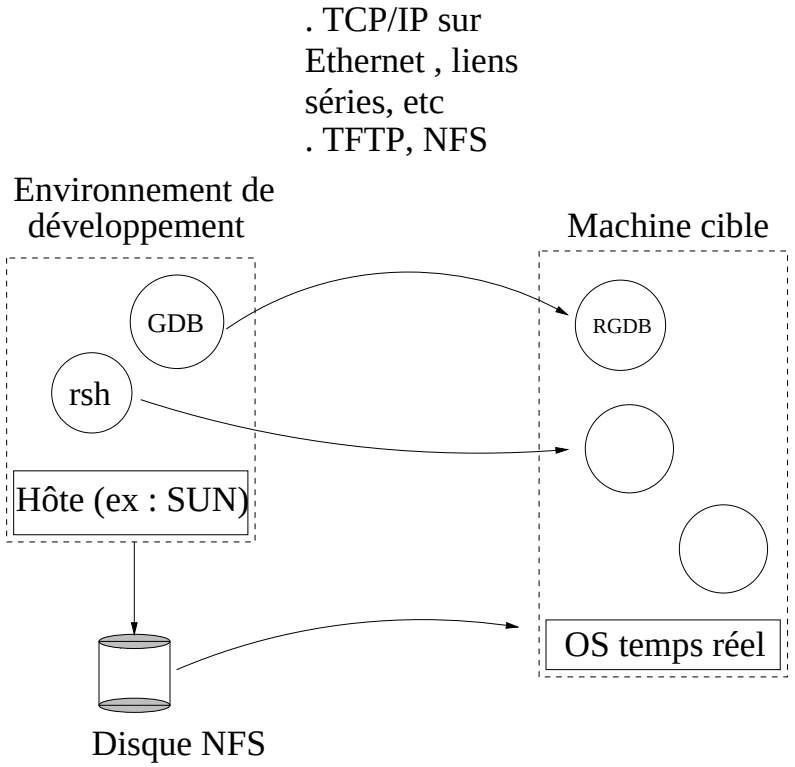
- Structuration en couches $\Longrightarrow$ notion de BSP.
- Portabilité du système et de l'applicatif.

Exécutif : architecture et services (4)

- **Performance connue et déterministe:**
  - Doit permettre l'évaluation de la capacité des tâches par exemple.
  - Utilisation de benchmarks (ex : $Rhealstone$, $Hartstone$, etc).

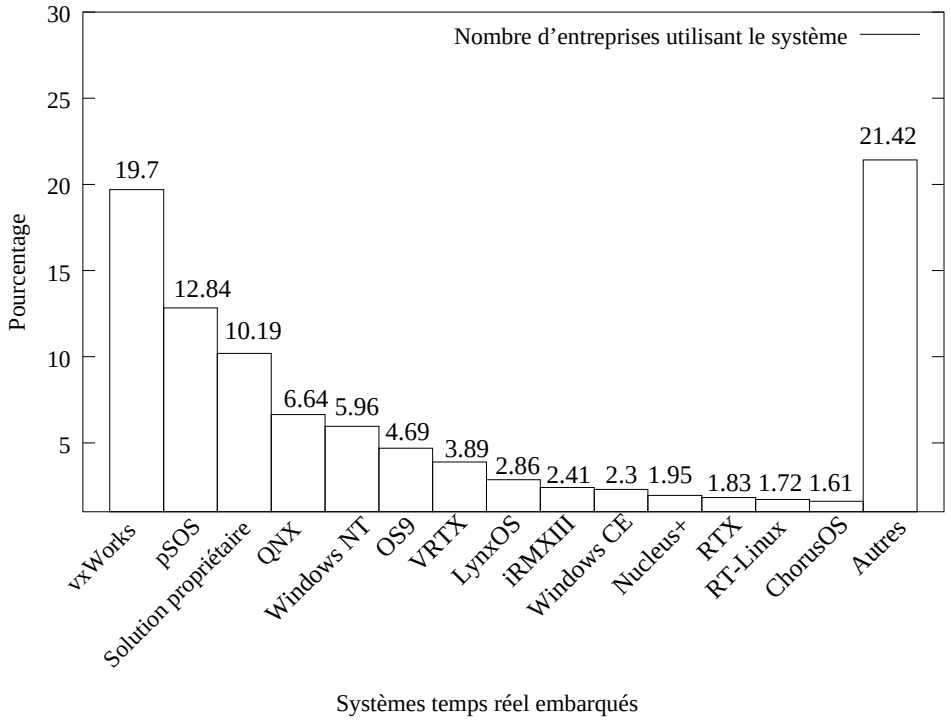
- **Critères de performances :**
  - Latence sur interruption.
  - Latence sur commutation de contexte/tâche.
  - Latence sur préemption.
  - Sémaphore "shuffle" (temps écoulé entre la libération d'un sémaphore et la réactivation d'une tâche bloquée sur celui-ci).
  - Temps de réponse pour chaque service (appel système, fonctions de bibliothèque).
  - etc.

Exécutif : architecture et services (5)



- **Phase de développement :** édition du source, compilation croisée, téléchargement, exécution et tests.
- **Phase d'exploitation :** construction d'une image minimale (exécutif + application) sans les services de développement. Stockage en EEPROM, Flash.

Etat du marché (1)



- **Caractéristiques du marché [TIM 00] :**
  - Diversité des produits présents $\Longrightarrow$ produits généralistes ou spécifiques à des applications types.
  - Présence importante de produits "maisons".

Etat du marché (2)

- **Quelques exemples de produits industriels :**
  - VxWorks : produit généraliste et largement répandu (PABX, terminal X de HP, Pathfinder, satellite CNES, etc).
  - pSOS édité par ISI (appli militaire, tél. portable).
  - VRTX édité par Microtec (appli militaire, tél. portable).
  - LynxOs (Unix temps réel).
  - Windows CE/Microsoft (systèmes embarqués **peu** temps réel).
- **Produits "open-source" :**
  - OSEK-VDX (appli. automobile).
  - RTEMS de Oar (appli. militaire).
  - eCos de cygnus.
  - RT-Linux.

La norme POSIX (1)

- **Objectif :** définir une interface standard des services offerts par UNIX[VAH 96, J. 93] afin d'offrir une certaine **portabilité** des applications.

- Norme étudiée/publiée conjointement par l'ISO et l'ANSI.

- **Problèmes :**
  - Portabilité difficile car il existe beaucoup de différences entre les UNIX.
  - Tout n'est pas (et ne peut pas) être normalisé.
  - Divergence dans l'implantation des services POSIX (ex : threads sur Linux).
  - Architecture de la norme.

- **Exemple de systèmes POSIX :** Lynx/OS, VxWorks, Solaris, Linux, QNX, etc .. (presque tous les systèmes temps réel).

La norme POSIX (2)

- **Architecture de la norme :** découpée en chapitres optionnels et obligatoires. Chaque chapitre contient des parties obligatoirement présentes, et d'autres optionnelles.

- Exemple de chapitres de la norme POSIX :

Chapitres	Signification
POSIX 1003.1	Services de base (ex : $fork$, $exec$, ect)
POSIX 1003.2	Commandes shell (ex : sh)
POSIX 1003.1b [GAL 95]	Temps réel
POSIX 1003.1c [RIF 95]	Threads
POSIX 1003.5	POSIX et Ada
etc	

La norme POSIX (3)

- Cas du chapitre POSIX 1003.1b : presque tout les composants sont optionnels !!

Nom	Signification
_POSIX_PRIORITY_SCHEDULING	Ordonnancement à priorité fixe
_POSIX_REALTIME_SIGNALS	Signaux temps réel
_POSIX_ASYNCHRONOUS_IO	E/S asynchrones
_POSIX_TIMERS	Chien de garde
_POSIX_SEMAPHORES	Sémaphores
etc ...	

- **Conséquence :** que veut dire "être conforme POSIX 1003.1b" ... pas grand chose puisque la partie obligatoire n'est pas suffisante pour construire des applications temps réel.

La norme POSIX (4)

- Les threads POSIX.
- Services d’ordonnancement.
- Outils de synchronisation.
- Les signaux temps réel.
- La manipulation du temps.
- Les entrées/sorties asynchrones.
- Les files de messages.
- La gestion mémoire.

Sommaire

Sommaire

1. Présentation.
2. Environnement de compilation croisée et environnement d’exécution.
3. Services offerts par l’exécutif VxWorks.

Présentation

- Edité par la société Wind River System[RIV 97]. Leader du marché.
- Exécutif multi-programmé $\Longrightarrow$ abstraction de tâches.
- Adressage virtuel mais pas de protection mémoire. Pas de swap. **Il n'existe qu'un seul espace d'adressage dans le système**.
- Exécutif modulaire.
- L'exécutif possède deux interfaces : POSIX 1003[GAL 95] et propriétaire.
- Disponible pour de très nombreuses architectures matérielles (BSP).

Abstractions de VxWorks (1)

- **Rappel :** un processus UNIX = contexte d'exécution + code partagé ou non + espace d'adressage privé $\Longrightarrow$ autant d'espace d'adressage que de processus.

```
#include <stdio.h>

int a=100;

int main(int argc, char* argv[]) {
        if(fork()==0)
                a+=100;
        else    {
                sleep(1);
                printf("a = %d\n",a);
                }
}
```

- Quelle est la valeur affichée après le fork() ?
- Idem sur VxWorks ?

Abstractions de VxWorks (2)

```
#include <stdio.h>

int a=100;

int main(int argc, char* argv[])
{
        if(fork()==0)
                a+=100;
        else    a+=200;

        sleep(1);
        printf("a = %d\n",a);
}
```

- Quelle est la valeur affichée après le fork() ?
- Idem sur VxWorks ?

Abstractions de VxWorks (3)

- **Un seul espace d'adressage unique implique :**
  - Corruption possible des données de l'exécutif par les applications $\Longrightarrow$ tout le monde voit tout. Pas de protection mémoire.
  - Cohérence des données partagées entre les tâches $\Longrightarrow$ nécessite donc des outils de synchronisation.
- **D'où les abstractions suivantes de l'exécutif :**
  - Tâches.
  - Sémaphores.
  - File de messages et pipes.

Abstractions de VxWorks (4)

- **Pourquoi un seul espace d'adressage :**
  - Contexte de tâche plus petit.
  - Pas d'appel système : les services sont invoqués par un simple appel de fonction $\Longrightarrow$ toutes les tâches applicatives sont des tâches "superviseurs/privilégiées".
  - Communication entre tâches sans traverser le système d'exploitation.

= gain de temps
= gain de "place" mémoire, de ressources
= gain de prédictiblité (cf. contraintes systèmes embarqués temps réel).

Architecture de l'exécutif (1)

- Constitué de tâches et de bibliothèques/agences :
- **Exemple de bibliothèques :**
  - $taskLib$. Gestion des tâches.
  - $sockLib$. Sockets BSD.
  - $ftpLib$. Client FTP.
- **Exemple de tâches systèmes :**
  - $tShell$. Shell "à la UNIX" pour la cible.
  - $tLogTask$. Prise de journal.
  - $tRlogind$. Connexion par la commande $rlogin$ sur la cible.

Architecture de l'exécutif (2)

- Exécutif modulaire $\Longrightarrow$ composition selon l'application ciblée ou l'utilisation de l'exécutif :

- **Exemple de composition pendant la :**
  - Phase de développement/test : bibliothèques d'entrées sorties, bibliothèques et pilotes réseaux. Tâches pour le développement (ex : $tShell$, $tRlogind$).
  - Phase d'exploitation : le noyau, $taskLib$, pas de pilote Ethernet, pas de pile TCP/IP, pas de tâche $tShell$, etc.

- **Le contenu de l'exécutif est choisi lors de la construction d'une image par le biais :**
  1. De bibliothèques ajoutées lors de l'édition des liens
  2. Des tâches à démarrer au "boot".

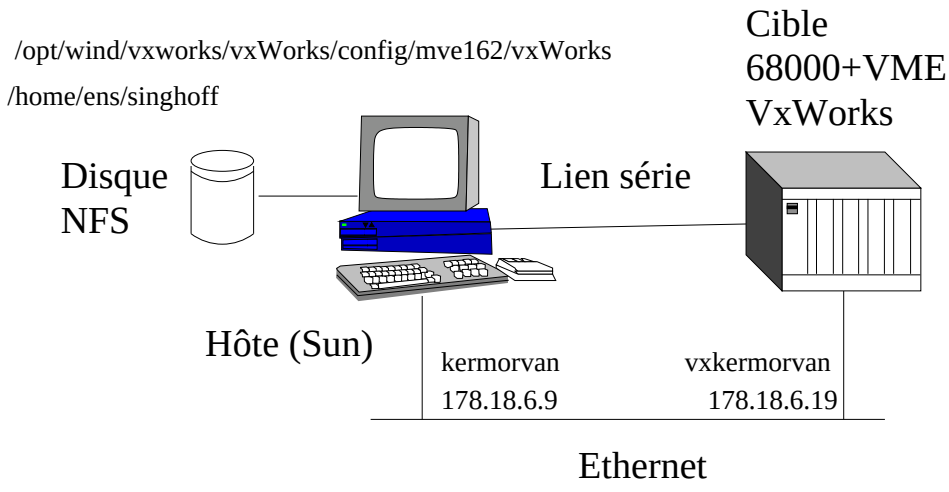
Sommaire

1. Présentation.
2. Environnement de compilation croisée et environnement d'exécution.
3. Services offerts par l'exécutif VxWorks.

Environnement

- Le chargeur.
- La chaîne de compilation croisée.
- Le shell VxWorks.
- La mise au point à distance.

Le chargeur (1)



- **Lancement exécutif et/ou application :**

1. Lancement du chargeur à la mise sous tension (EEPROM).
2. Modification éventuelle des paramètres de "boot".
3. Configuration interface Ethernet + chargement ftp/rsh de l’image.
4. Démarrage de l’exécutif et des tâches VxWorks.
5. Montage NFS du répertoire utilisateur.

Le chargeur (2)

- **Paramètres de "boot" :**
  - Adresses IP de l'hôte et de la cible.
  - Localisation de l'image constituant l'exécutif.
  - Nom de l'utilisateur.
  - Protocole à utiliser pour le chargement.

- **Commandes du chargeur :**
  - CTRL+X : redémarrage de la carte.
  - @ : chargement et lancement de l'exécutif.
  - c : modification des paramètres de chargement.
  - p : affichage des paramètres de chargement.
  - ? : aide en ligne.

Le chargeur (3)

- **Exemple :**

1. A partir de l'hôte, on se connecte à la cible par la voie série :

```
connect A
```

2. Affichage et modification éventuelle des paramètres :

```
[VxWorks Boot]: p
boot device          : ei
host name            : kermorvan
file name            : /opt/..../config/mv162/vxWorks
inet on ethernet (e) : 172.18.6.19
host inet (h)        : 172.18.6.9
user (u)             : singhoff
ftp password (pw) (blank = use rsh):
target name (tn)     : vxkermorvan
```

3. Chargement de l'exécutif :

```
[VxWorks Boot]: @
->
```

Compilation croisée (1)

- La chaîne de compilation sur l'hôte (ex : Sun) est constituée de :
  - $cpp68k$. Préprocesseur.
  - $cc68k$. Compilateur C.
  - $ld68k$. Editeur de liens (édition incrémentale uniquement).
  - $nm68k$. Liste les symboles d'un objet.
  - $as68k$. Assembleur 68000.
  - $ar68k$ et $ranlib68k$. Outils de construction de bibliothèques.
- Les binaires produits par ces commandes ne sont pas exécutables sur Sun mais sur la cible (Motorola 68000 + VME).
- Il n'y a pas d'édition de liens "finale". Le binaire obtenu reste un module objet (collection de données et de fonctions).
- Il n'y a pas de fonction "main".

Compilation croisée (2)

- Exemple de programme (fichier $hello.c$) :

```
#include <vxWorks.h>
#include <stdio.h>
#include <taskLib.h>

helloWorld() {
        int id=taskIdSelf();
        printf("Nom = %s ; Id = %x\n",taskName(id),id);
}
```

- Compilation sur l'hôte (pas d'édition de liens) :

```
kermorvan$cc68k -g -c hello.c -DCPU=MC68030
```

- Table de symboles générée :

```
kermorvan$nm68k hello.o
0000002e T _helloWorld
         U _printf
         U _taskIdSelf
         U _taskName
00000000 t gcc2_compiled.
```

Compilation croisée (3)

- **Un programme VxWorks se compose :**
  1. D'une collection de variables et de fonctions.
  2. D'une **tâche d'initialisation**, de priorité maximale, qui alloue les ressources nécessaires à l'application (opérations coûteuses et non prédictibles).
  3. D'un ensemble de **tâches activées par la tâche d'initialisation**.

- Objectif : permettre une préallocation des ressources et un démarrage synchronisé de toutes les tâches.

$\Longrightarrow$ **Patron de conception utilisé dans de nombreuses plates-formes => ARINC 653**

Le shell VxWorks (1)

- S'exécute sur la cible (tâche VxWorks $tShell$). Constitue l'interface entre l'exécutif et l'utilisateur.

- **Services offerts :**
  - Chargement de code et de données. Edition des liens.
  - Mise au point (niveau assembleur).
  - Consultation et modification des ressources VxWorks (ex : tâches).
  - Gestion de symbole des modules.
  - Services classiques offerts par un shell : variables, redirection E/S, appel de fonction, historique, etc.

Le shell VxWorks (2)

- **Chargement et exécution d'un module sur la cible :**

1. On se place dans le répertoire où se trouve le module :

```
->cd "mon_repertoire"
```

2. Chargement du module :

```
->ld < hello.o
```

3. Exécution par appel de fonction :

```
->helloWorld()
Hello world : nom tache = tShell ; tache id = 3a8460
```

4. Exécution par la création d'une tâche :

```
->sp helloWorld
task spawned: id = 0x3a5bc0, name = t1
Hello world : nom tache = t1 ; tache id = 3a5bc0
```

Le shell VxWorks (3)

- **Consultation d'informations sur les symboles et modules :**

```
-> lkup "hell"
_helloWorld                0x003fe5de text     (hello.o)
_shellTaskId               0x0007b188 bss      (vxWorks.sym)
_shellInit                 0x0004355e text     (vxWorks.sym)
_shell                     0x0004372c text     (vxWorks.sym)
_shellOrigStdSet           0x00043de8 text     (vxWorks.sym)
_shellPromptSet            0x00043dcc text     (vxWorks.sym)
...
-> moduleShow
MODULE NAME   MODULE ID  GROUP      TEXT START DATA START  BSS START
------------ ---------- ---------- ---------- ---------- ----------
hello.o       0x3fef78   2          0x3fef1c   0x3fef70   0x3fef70
```

Le shell VxWorks (4)

- **Consultation d'informations sur les tâches du système :**

```
-> i
  NAME        ENTRY       TID   PRI   STATUS    PC       SP   ERRNO DELAY
--------- ----------- ------ --- ------- ------- ------ ----- -----
tShell    _shell      3a8460   1  READY  2ff38   3a813c   0     0
t1        _helloWorld 3a5bc0 100  READY  3fef68  3a5b90   0     0
...
-> ti tShell
  NAME        ENTRY     TID   PRI   STATUS    PC       SP     ERRNO   DELAY
---------- --------- ------ --- ------- ------- ------- ------ -----
tShell     _shell    3a8460   1  READY  2ff38   3a813c       0    0

stack: base 0x3a8460  end 0x3a5d50  size 9000  high 4092  margin 4908

VX_SUPERVISOR_MODE  VX_DEALLOC_STACK    VX_FP_TASK    VX_STDIO
d0  =       30   d1  =     3004   d2  =        0   d3  =        0
d4  =        0   d5  =        0   d6  =        0   d7  =        0
a0  =    7708e   a1  =   3a5bc0   a2  =        0   a3  =        0
```

Le shell VxWorks (5)

- **Fonctions de type "shell" (utiles pour la mise au point) :**

```
#include <vxWorks.h>

int var = 100;
affiche()
{
        printf("var = %d\n",var);
}
```

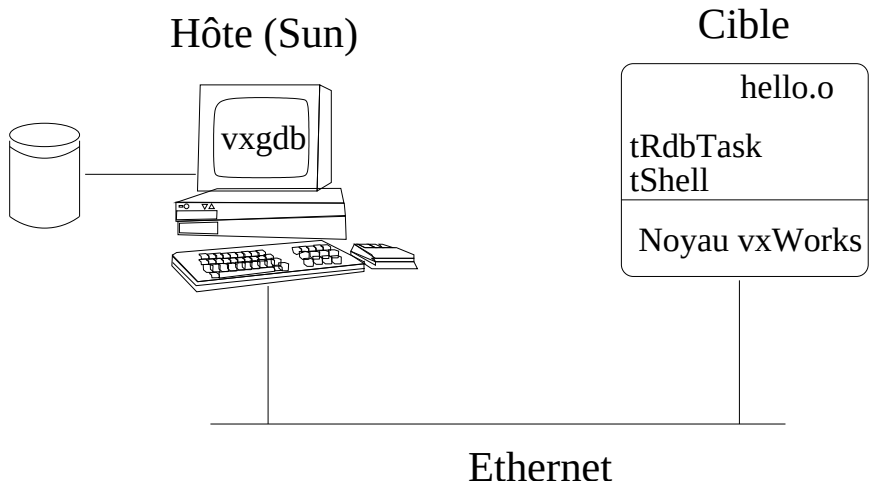
- **Scénario d'exécution :**

```
-> ld < var.o
-> affiche()
var = 100

-> ma_var = 200
new symbol "ma_var" added to symbol table.
-> var = var + ma_var

-> affiche()
var = 300
```

Le shell VxWorks (6)

- **Principales commandes du shell VxWorks :**
  - ls, ll, cd, pwd. Commandes de parcours du répertoire utilisateur.
  - ld. Chargement d'un module et édition des liens.
  - $moduleShow$. Affiche la liste des modules.
  - $lkup, lkAddr$. Consultation de la table des symboles.
  - $sp, td, ts, tr, repeat$. Opérations sur les tâches : création, suppression, etc.
  - $i, ti, checkStack, pc, printErrno$. Information sur les tâches.
  - Aide en ligne : $help$.

La mise au point à distance (1)



- **Principe : permettre la mise au point au niveau source d'une application s'exécutant sur une cible.**
  - gdb pour VxWorks : $vxgdb$. Commandes (presque) identiques.
  - $vxgdb$ est lancé sur l'hôte. L'application est exécutée sur la cible $\Longrightarrow$ coopération entre $vxgdb$ et la cible.

La mise au point à distance (2)

- **Sur la cible :**
  1. Chargement du module :

     ```
     ->ld < hello.o
     ```

- **Sur l'hôte :**
  1. Lancement de gdb :

     ```
     kermorvan$vxgdb&
     ```

  2. Connexion à la cible :

     ```
     (vxgdb)target vxworks vxkermorvan
     ```

  3. Lancement d'une tâche :

     ```
     run helloWorld
     ```

  4. Commandes $vxgdb$: break, list, step, next, display, printf, etc.

Sommaire

1. Présentation.
2. Environnement de compilation croisée et environnement d'exécution.
3. Services offerts par l'exécutif VxWorks.

Services de l’exécutif VxWorks

- **Services classiquement offerts par un exécutif temps réel :**

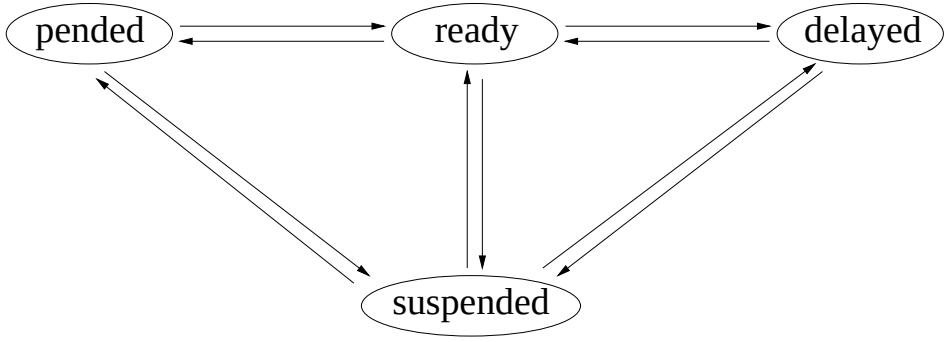
  - Ordonnancement et tâches.
  - Synchronisation : les sémaphores.
  - Communication : pipe et file de messages.
  - Manipulation du temps.
  - Gestion des interruptions.
  - Drivers et vxWorks.

Ordonnancement (1)

- **Caractéristiques principales :**
  - Ordonnancement à priorités fixes, préemptif, HPF [a]. Niveaux de priorité allant de 0 (la plus forte) à 255 (la plus faible).
  - Quantum identique pour toutes les tâches. Modifiable par l'utilisateur.
  - Politique de round-robin pour les tâches de même priorité : équivalent à $SCHED\_FIFO$ si le quantum est nul, équivalent à $SCHED\_RR$ sinon (cf. POSIX 1003[GAL 95]).

[a] Highest Priority First.

Ordonnancement (2)



- **La tâche dans VxWorks :**
  - Tâche = pile + TCB [a]. Le TCB stocke le contexte d'une tâche (ex : $errno$, PC, SP, état, etc). La pile est de taille fixe.
  - Pas d'espace d'adressage privé.
  - Pas de relation de parenté entre les tâches.
  - Identifiant unique numérique et alphanumérique.

[a] Task Control Block

Ordonnancement (3)

- **Services principaux :**

$taskSpawn$	Création et démarrage d'une tâche.
$taskDelay(t)$	Suspend durant t tics une tâche.
$taskPriorityGet()$, $taskPrioritySet()$	Lecture et changement de la priorité d'une tâche.
$kernelTimeSlice(t)$	Positionne le quantum de l'ordonnanceur.
$taskName, taskIdTcb,$ $taskIdSelf$	Renvoient le nom, l'identifiant et le TCB d'une tâche.
$exit$	Termine une tâche.

Ordonnancement (4)

- **Exemple de création d'une tâche (fichier $pere.c$) :**

```
coucou() {
        printf("Tache %s\n", taskName(taskIdSelf()));
}

init() {
        tid=taskSpawn("fils1",101,0,5000,
            (FUNCPTR)coucou,0,0,0,0,0,0,0,0,0,0);
        tid=taskSpawn("fils2",101,0,5000,
            (FUNCPTR)coucou,0,0,0,0,0,0,0,0,0,0);
}
```

- **Exécution :**

```
-> ld < pere.o
-> sp init
task spawned: id = 0x3a59a0, name = fils1
task spawned: id = 0x3a59a1, name = fils2
Tache fils1
Tache fils2
```

Ordonnancement (5)

- **Autres services importants :**

$taskLock()/UnLock()$	Désactive le caractère préemptif de l'ordonnanceur.
$taskSuspend(tid)$	Suspend une tâche $\Longrightarrow$ mise au point seulement.
$taskResume(tid)$	Réveille une tâche.
$taskDelete(tid)$	Destruction d'une tâche.
$taskInit/taskActivate$	Création et activation séparées d'une tâche.
etc.	

Synchronisation : les sémaphores (1)

- **Types de sémaphores offerts :**
  1. Les mutex.
  2. Les sémaphores à compteur.

- **Propriétés :** absence d'interblocage (cf. méthode d'allocation), famine (acceptée dans un système temps réel), garantie sur la synchronisation attendue.

- **Utilisation :**
  - Exclusion mutuelle, producteurs/consommateurs, lecteurs/rédacteurs ...
  - Gestion de ressources (moniteur) :
    - Méthodes d'allocation : classe ordonnée, allocation globale.
    - Patrons de conception : compteurs de ressources, sémaphores privés, collectifs.

Synchronisation : les sémaphores (2)

- **Interface pour la manipulation des sémaphores :**

$semGive(sem)$	Libération du sémaphore.
$semTake(sem, timer)$	Acquisition du sémaphore. Primitive éventuellement bloquante. $timer$ = timeout sur le temps d'attente. $WAIT\_FOREVER$ permet d'utiliser un timeout infini.
$semDelete(sem)$	Destruction du sémaphore.

Synchronisation : les sémaphores (3)

- **Les mutex :**
  - Uniquement utilisable pour la mise en œuvre de section critique : la libération du sémaphore doit **obligatoirement** être effectuée par la tâche qui l'a précédemment acquise.
  - Prise de section critique récursive non bloquante.
  - Création par la primitives $semMCreate(opt)$. Options :
    1. Gestion FIFO ou par priorité de la file d'attente (options SEM_Q_FIFO et $SEM\_Q\_PRIORITY$).
    2. Utilisation d'un protocole d'héritage de priorité (option $SEM\_INVERSION\_SAFE$, protocole PCP).

Synchronisation : les sémaphores (4)

- **Exemple : section critique**

```
#include <semLib.h>
#include <taskLib.h>

SEM_ID msm;   int tid;

void tache()  {
    while(1) {
          semTake(msm,WAIT_FOREVER);
          /* utiliser la section critique */
          semGive(msm);
            }
    }
void init()  {
     msm = semMCreate(SEM_INVERSION_SAFE | SEM_Q_PRIORITY);
     tid = taskSpawn("mut1",101,0,5000,(FUNCPTR)tache,
            0,0,0,0,0,0,0,0,0,0);
     tid = taskSpawn("mut2",101,0,5000,(FUNCPTR)tache,
            0,0,0,0,0,0,0,0,0,0);
    }
```

Synchronisation : les sémaphores (5)

- **Les sémaphores à compteur : sémaphore mémorisant le nombre d'opérations effectuées.**

  - Sémaphores pour la mise en œuvre de section critique, la synchronisation entre tâches, la gestion des ressources (allocation, libération), la communication entre tâches.

  - $semCCreate(opt, val)$. Création du sémaphore. opt correspond aux mêmes options que pour les mutex. val est la valeur initiale du sémaphore.

Synchronisation : les sémaphores (6)

- **Exemple 1 : synchronisation entre tâches**

```
SEM_ID msm;
void  tache() {
              printf("tache %d en attente\n",taskIdSelf());
              semTake(msm,WAIT_FOREVER);
              printf("tache %d debloquee\n",taskIdSelf());
}
void init() {
          msm = semCCreate(SEM_Q_FIFO ,0);
          tid = taskSpawn("sem",101,0,5000,(FUNCPTR)tache,
                        0,0,0,0,0,0,0,0,0,0);
          taskDelay(800);
          printf("init : liberation de la tache\n");
          semGive(msm);
}
```

- Scénario d'exécution :

```
-> sp init
tache 3772500 en attente
init: liberation de la tache
tache 3772500 debloquee
```

Synchronisation : les sémaphores (7)

- **Exemple 2 : moniteur (compteur de ressources)**

```
/* Sémaphores compteur d'imprimante */
SEM_ID imprimantes;

/* Nombre d'imprimantes disponibles initialement  */
#define NB_IMPRIMANTES 5

init() {
     imprimantes=semCCreate(0,NB_IMPRIMANTES);
}

/* Allouer une imprimante  */
void allouer(void) {
     semTake(imprimantes,WAIT_FOREVER);
}

/* Liberer une imprimante  */
void liberer(void) {
     semGive(imprimantes);
}
```

Synchronisation : les sémaphores (8)

- **Exemple 3 : moniteur (paradigme des sémaphores privés)**

```
SEM_ID sempriv[NB_TACHES];     /* Sémaphores privés, un par tache */

#define ATTENDRE 1    /* Etat du systeme */
#define FAIT_QQ_CHOSE 2
#define UTILISER 3
int taches[NB_TACHES];
int ressources=5;

SEM_ID section_critique;   /* Mutex */

init() {
     int i;
     section_critique=semMCreate(0);
     for(i=0;i<NB_TACHES;i++) {
           sempriv[i]=semCCreate(0,0);
           taches[i]=FAIT_QQ_CHOSE;
           taskSpawn(...);
           }
}
```

Synchronisation : les sémaphores (9)

```
/* Allouer les ressources pour la tache "id" */
void allouer(int id)
{
     semTake(section_critique,WAIT_FOREVER);

     if(ressources<0)
          {
          taches[id]=ATTENDRE;
          }
     else {
          taches[id]=UTILISER;
          ressources--;
          semGive(sempriv[id]);
          }

     semGive(section_critique);

     semTake(sempriv[id],WAIT_FOREVER);
}
```

Synchronisation : les sémaphores (10)

```
/* Liberer les ressources de la tache "id" */
void liberer(int id)
{
     int i;

     semTake(section_critique,WAIT_FOREVER);

     taches[id]=FAIRE_QQ_CHOSE;
     ressources++;

     for(i=0;i<NB_TACHES;i++)
          if( (ressources>0) && (taches[i]==ATTENDRE) )
               {
               ressources--;
               taches[i]=UTILISER;
               semGive(sempriv[i]);
               }

     semGive(section_critique);
}
```

Communication (1)

- **Outils de communication offerts :**

1. Mémoire partagée + sémaphores.
2. Files de messages : efficace, utilisation de priorités pour la gestion de la file, timeout possible.
3. Pipes nommés : utilisation des fonctions E/S standard (read/read/select) , descripteur de fichiers.

Communication (2)

- **Les files de messages :**
  - Déterminisme temporel $\Longrightarrow$ pré-allocation mémoire.
  - Message de taille variable.
  - Producteurs et consommateurs multiples.
  - File mono-directionnelle.
  - Gestion des files d'attente : FIFO ou par priorité.
  - Utilisation de timeout.
- Primitives : $msgQCreate()$, $msgQSend()$, $msgQReceive()$.

Communication (3)

- $msgQCreate(nb, taille, opt)$. Création d'une file contenant au plus nb messages de maximal $taille$ octets. Options : $MSG\_Q\_PRIORITY$ (tâches servies selon leur priorité), MSG_Q_FIFO (tâches servies de façon FIFO).

- $msgQSend(id, buff, taille, timeout, prio)$. Emission d'un message sur la file id de maximal $taille$ octets et stocké à l'adresse $buff$. $prio$ désigne la mode d'insertion dans la file (MSG_PRI_NORMAL = en queue ; MSG_PRI_URGENT = en tête). A expiration de $timeout$ tics, une erreur est levée.

- $msgQReceive(id, buff, taille, timeout)$. Primitive bloquante jusqu'à réception d'un message sur la file id de taille maximal $taille$ et stocké à l'adresse $buff$.

Communication (4)

- **Exemple : producteur/consommateur**

```
MSG_Q_ID msg;

void init()  {
        msg = msgQCreate(20,500,MSG_Q_FIFO);
        tid=taskSpawn("cons",101,0,10000,
             (FUNCPTR)consommateur, 0,0,0,0,0,0,0,0,0,0);
        tid=taskSpawn("prod",101,0,10000,
             (FUNCPTR)producteur, 0,0,0,0,0,0,0,0,0,0);
}

void producteur()  {
        char buff [100];

        while(1)  {
                sprintf(buff,"Il est %d tics\n",tickGet());
                msgQSend(msg,buff,sizeof(buff),
                        WAIT_FOREVER,MSG_PRI_NORMAL);
        }
}
```

Communication (5)

```
void consommateur()
{
        char buff [100];

        while(1)
        {
                strcpy(buff,"");
                msgQReceive(msg,(char *) buff,sizeof(buff),
                            WAIT_FOREVER);
                printf("message = %s\n",buff);
        }
}
```

- **Scénario d'exécution :**

```
-> sp init
message = Il est 32262 tics
message = Il est 33002 tics
...
```

Manipulation du temps (1)

- **Services liés au temps :**
  1. Quelle heure est il ?
  2. Bloquer un processus/tâche pendant une durée donnée.
  3. Réveiller un processus/tâche régulièrement ($timer$ ou chien de garde) $\Longrightarrow$ tâches périodiques.
- **Précision de ces services :**
  1. Liée au matériel présent (circuit d'horloge), à ses caractéristiques (période de l'interruption) et au logiciel qui l'utilise (handler d'interruption).
  2. Ex : Linux intel : circuit activé périodiquement (10 ms) + registre RDTSC du Pentium. Résultat = mesure en micro-seconde ($gettimeofday$) mais temps de réveil autour de 10/12 ms.

Manipulation du temps (2)

- **Services disponibles utilisant des "tics" (portabilité):**
  - Chiens de garde VxWorks.
  - $taskDelay(nb)$: bloque une tâche pendant nb tics.
  - $tickSet()/tickGet()$. Positionne ou consulte le nombre de top d'horloge depuis la mise sous tension.
  - $sysClkRateSet()/sysClkRateGet()$. Modification ou consultation de la fréquence de l'horloge système.
  - Outils de profiling ($timexLib$).

Manipulation du temps (3)

- **Chien de garde :** génération d'une interruption sur expiration d'un timeout.

- **Interface pour la gestion des chiens de garde :**

  - $wdCreate()$. Initialisation d'un chien de garde.
  - $wdStart(wd, delai, func, parm)$. Activation du chien de garde : après $delai$ tics, la fonction $func$ est exécutée avec le paramètre $parm$.
  - $wdCancel(wd)$. Annulation d'un chien de garde en cours de décrémentation.
  - $wdDelete(wd)$. Destruction d'un chien de garde.

Manipulation du temps (4)

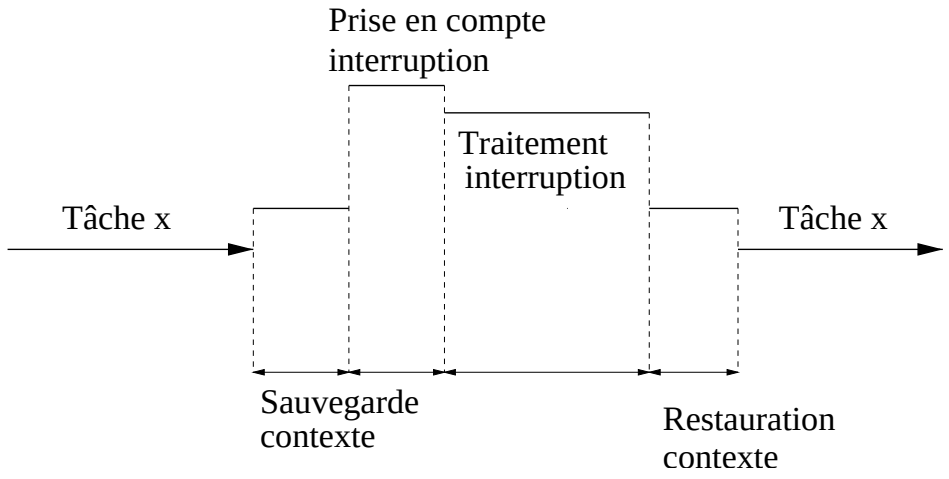
- **Exemple : contrôle d'une échéance :**

```
WDOG_ID wd;
int go=1;

trop_tard()  {
      go=-1;
}
...
wd=wdCreate();
wdStart(wd,echeance,(FUNCPTR)trop_tard,0);
while(go>0) {
     printf("Je Bosse .....\n");
     taskDelay(10);
     }
printf("Debloquee par interruption : echeance ratee\n");
```

- **Scénario d'exécution :**

```
-> sp echeance
Je Bosse .....
Je Bosse .....
Debloquee par interruption : echeance ratee
```

Gestion des interruptions (1)



- Prise en compte d'événements matériels ou logiciels.
- **Caractéristiques :**
  - Modification possible du vecteur d'interruption.
  - Une seule pile pour toutes les interruptions (attention aux interruptions imbriquées).
  - Pas de co-gestion tâches/interruptions : contexte interruption différent $\Longrightarrow$ services de l'exécutif pas tous disponibles dans une interruption (ex : opérations bloquantes, E/S, etc).
  - Partie dépendante du BSP.

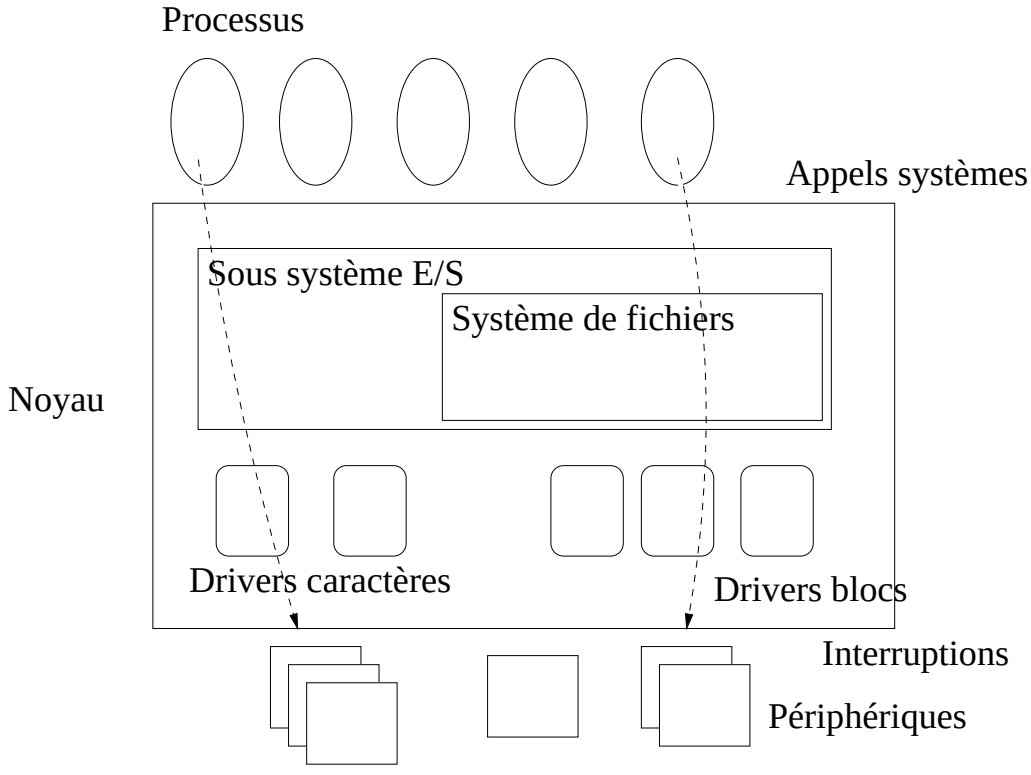
Gestion des interruptions (2)

- **Principales primitives :**

$intConnect()$	Connection d'un handler à une interruption donnée (ex : horloge).
$intLock()$	Autorise les interruptions.
$intUnLock()$	Interdit les interruptions.
$intCount()$	Consultation du niveau d'imbrication de traitement des interruptions.
$intContext()$	Détermine si le contexte d'exécution est celui d'une interruption ou non.

Drivers et vxWorks (1)

- **Les drivers dans UNIX :**



- Pilotes/drivers orientés caractères et orientés blocs. Accès en mode "raw".
- Notion de pseudo-driver (drivers caractères).

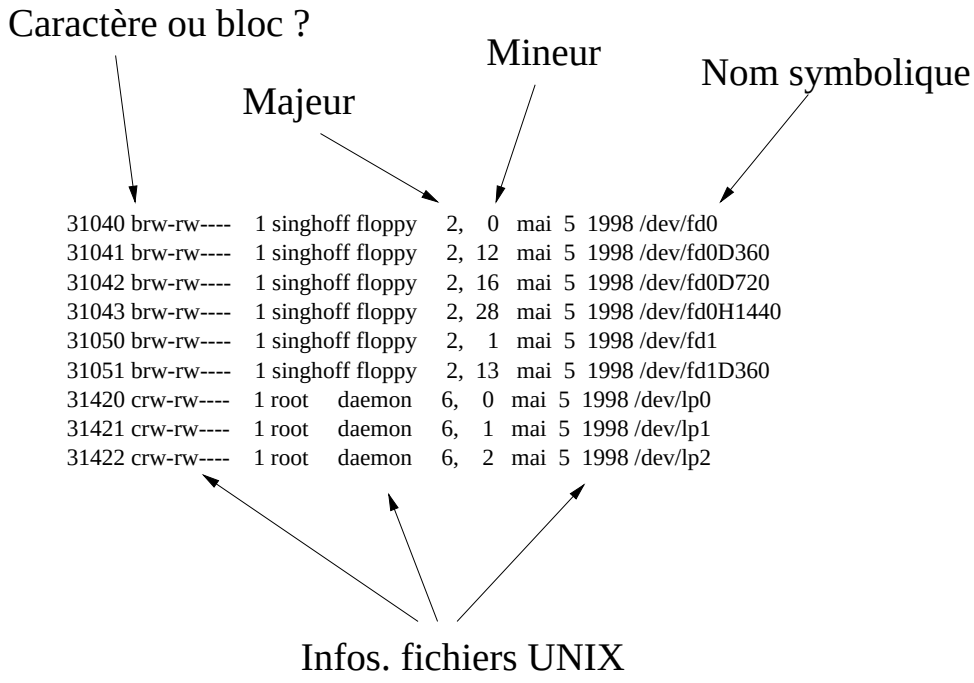
Drivers et vxWorks (2)

- Un driver est un composant indépendant du reste du système d'exploitation.
- **Interface d'un driver (caractères) :**

$close$	Fermeture du périphérique.
$creat$	Création du périphérique.
$open$	Ouverture du périphérique.
$remove$	Destruction du périphérique.
$read$	Lecture sur le périphérique.
$write$	Ecriture sur le périphérique.
$ioctl$	Interface de contrôle : consultation ou modification d'attributs.

  - Selon la catégorie du périphérique, l'intégralité des opérations n'est pas forcément implantée.
  - Les interfaces "blocs" sont généralement plus complexes.

Drivers et vxWorks (3)

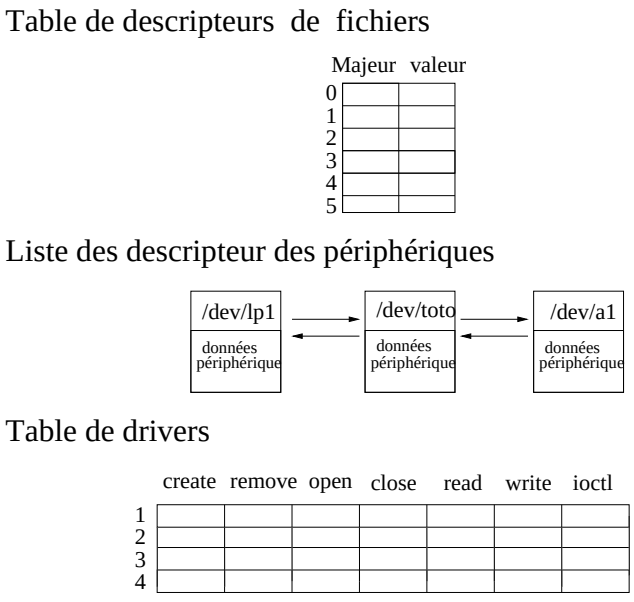


- **Désignation d'un périphérique :**
  - Nom de fichier ($/dev/xxx$). Nom symbolique destiné aux utilisateurs permettant :
    1. Création/utilisation de programmes manipulant indifféremment fichiers et périphériques.
    2. Droits UNIX.
  - Type, majeur et mineur $\Longrightarrow$ désignation destinée au système. Le majeur désigne un driver donné (table de drivers). Le mineur est local à un driver.

Drivers et vxWorks (4)

- **Spécificités d'un driver sur VxWorks :**
  - Drivers et applications partagent le même espace d'adressage (exécutif).
  - Comme les tâches, le code du driver est préemptible.
  - Concurrence possible lors de l'exécution du code du driver.
  - Pas de notion de mineur.
  - Chargement et retrait dynamique des drivers.
  - Stucture simplifiée du système d'E/S.

Drivers et vxWorks (5)



- **Structure du système d'E/S :**
  - Une liste de périphériques.
  - Une table de descripteurs de fichiers commune à toutes les tâches.
  - Une table de drivers indexée par le majeur.

Drivers et vxWorks (6)

- **Composants logiciels d'un driver sur vxWorks :**
  1. Un descripteur de périphérique : structure C mémorise les données associées à chaque périphérique.
  2. Une fonction C permettant d'instancier la structure 1) pour chaque périphérique physique.
  3. Une fonction C pour chaque opération sur le périphérique ($read$, $write$, $open$, $creat$, $remove$, $ioctl$ et $close$).
  4. Une fonction C pour enregistrer le driver dans le système E/S ; enregistrement des fonctions 3).

- **Installation d'un driver :**
  1. Chargement en mémoire des modules objets constituant le driver.
  2. Pour chaque périphérique physique, allocation d'un descripteur de périphérique.
  3. Enregistrement des fonctions $read$, $write$, $open$, $creat$, $remove$, $ioctl$ et $close$ dans le système E/S et récupération du majeur associé au driver.

Drivers et vxWorks (7)

- **Installation du driver :**

```
int demoDrvMajor=-1;

STATUS demoDrv () {
  demoDrvMajor = iosDrvInstall (demoOpen, (FUNCPTR)NULL, demoOpen,
                        demoClose, demoRead, demoWrite, demoIoctl);
}
```

Drivers et vxWorks (8)

- **Descripteur de périphérique :**

```
typedef struct {
  DEV_HDR devHdr;
  char data[MAX_SIZE];
} demo_dev;
```

- **Allocation d'un descripteur de périphérique :**

```
STATUS demoDevCreate (char* name) {
  demo_dev *pDev;

  if ((pDev = (demo_dev *) malloc (sizeof (demo_dev))) == NULL)
    return (ERROR);
  strcpy(pDev->data,"");
  return (iosDevAdd ((DEV_HDR *) pDev, name, demoDrvMajor));
}
```

Drivers et vxWorks (9)

- **Fonction C pour chaque opération sur un périphérique :**

```
int demoOpen (demo_dev* pDev, char* name, int mode) {
  pDev->mode=mode;
  return ((int) pDev);
}

int demoRead (demo_dev* pDev, char* buffer, int maxbytes) {
  strncpy(buffer,pDev->data,maxbytes);
  return (strlen(buffer)+1);
}

int demoWrite (demo_dev* pDev, char* buffer, int maxbytes) {
  strncpy(pDev->data,buffer,maxbytes);
  return (strlen(pDev->data)+1);
}
...
```

Drivers et vxWorks (10)

- **Exemple d'application :**

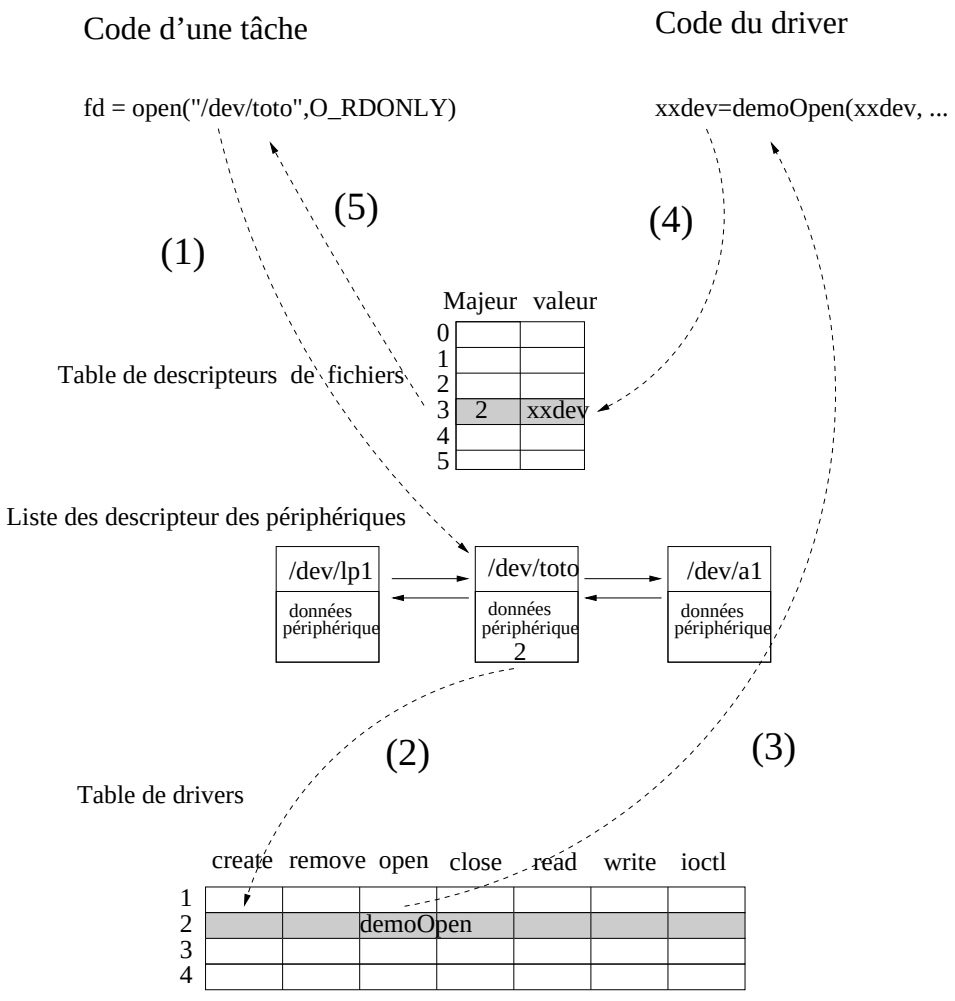
```
int fd=-1;

void open_file(char* name, int mode) {
  fd=open(name,mode,0);
}

void read_file() {
  char buff [100];
  int nb=read(fd,&buff,20);
  printf(" read ; size = %d ; text = %s\n",nb,buff);
}

void write_file(char* buff) {
  int nb=write(fd,buff,20);
  printf(" write ; size = %d\n",nb);
}
```

Drivers et vxWorks (11)

- **Exemple d'utilisation :**

```
-> ld < demoDrv.o
-> demoDrv()
-> demoDevCreate("/dev/toto")
-> devs
drv name
  0 /null
  1 /tyCo/0
  4 kermorvan:
  6 /pty/rlogin.M
  8 /dev/toto
-> open_file("/dev/toto",2)
-> read_file
 read ; size = 1 ; text =
-> write_file("coucou")
 write ; size = 7
-> read_file
 read ; size = 7 ; text = coucou
```

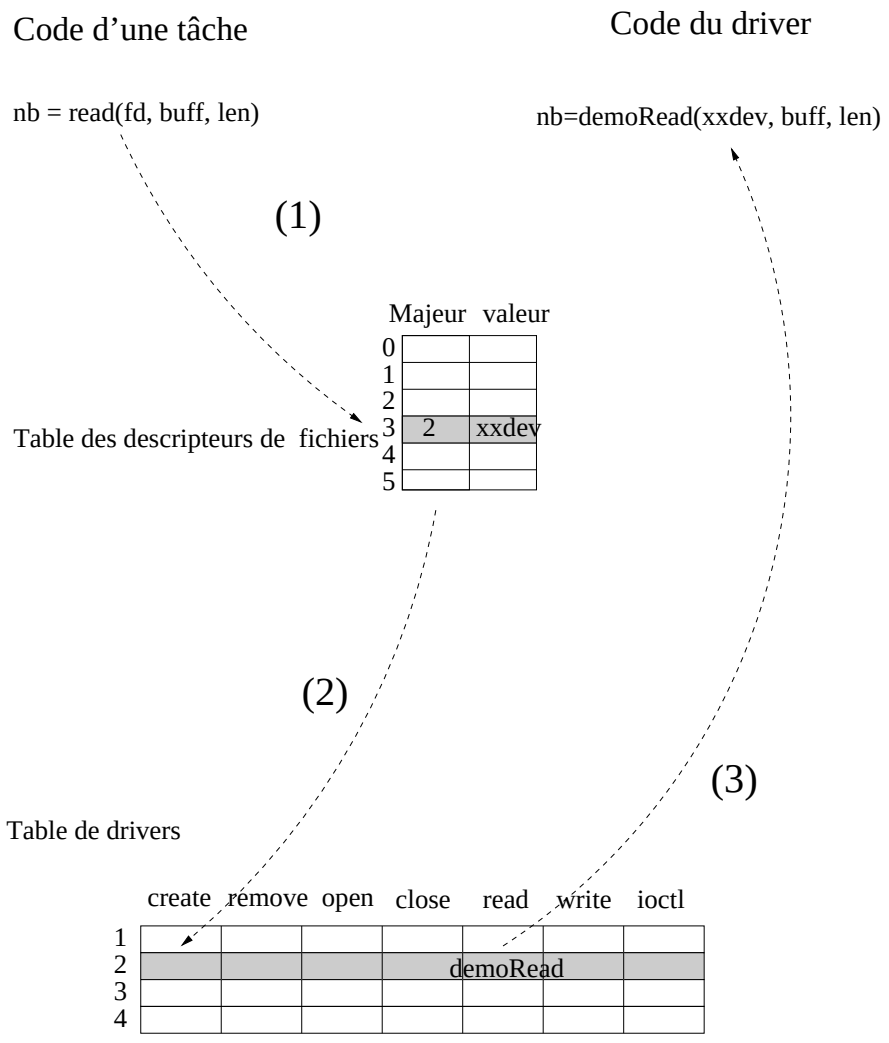
Drivers et vxWorks (12)



Drivers et vxWorks (13)

1. Recherche du périphérique dans la liste des périphériques.
2. Accès à la table de drivers par le majeur ($majeur = 2$).
3. Appel de $demoOpen$ (code du driver).
4. En retour, allocation d'une entrée dans la table de descripteurs de fichiers puis mémorisation du majeur et de l'information donnée par $demoOpen$.
5. Renvoyer le descripteur de fichier ($fd = 3$).

Drivers et vxWorks (14)



Drivers et vxWorks (15)

1. Grâce au descripteur de fichier, récupération du majeur.
2. Accès à la table de drivers par le majeur.
3. Appel de $demoRead$ (code du driver).

Sommaire

Résumé

- **Généralités :**
  - Notion de systèmes embarqués temps réel.
  - Différences entre un exécutif et un système d'exploitation.
  - Modularité, architecture hôte/cible, portabilité, BSP.
  - Etat du marché.

- **VxWorks :**
  - Abtractions : tâches, adressage virtuel unique, outils de communication et de synchronisation.
  - Utilisation de l'environnement de compilation et d'exécution.
  - API de l'exécutif (ordonnancement, outils de communication et de synchronisation).

Sommaire

Références (1)

[DEM 94] I. Demeure and J. Farhat. « Systèmes de processus légers : concepts et exemples ». *Technique et Science Informatiques*, 13(6):765–795, décembre 1994.

[DEN 66] B. Dennis and E. C. Van Horn. « Programming Semantics for Multiprogrammed Computations ». *Communications of the ACM*, 9(3):143–155, March 1966.

[GAL 95] B. O. Gallmeister. *POSIX 4 : Programming for the Real World* . O'Reilly and Associates, January 1995.

[GHO 94] K. Ghosh, B. Mukherjee, and K. Schwan. « A survey of Real Time Operating Systems ». Technical Report, College of Computig. Georgia Institute of Technology. Report GIT-CC-93/18, February 1994.

Références (2)

[J. 93] J. M. Rifflet. *La programmation sous UNIX*. Addison-Wesley, 3rd edition, 1993.

[RIF 95] J. M. Rifflet. *La communication sous UNIX : applications réparties*. Ediscience International, 2rd edition, 1995.

[RIV 97] Wind River. *VxWorks : programmer's guide*. Wind River System, March 1997.

[TIM 00] M. Timmerman. « RTOS Market survey : preliminary result ». *Dedicated System Magazine*, (1):6–8, January 2000.

[VAH 96] U. Vahalia. *UNIX Internals : the new frontiers*. Prentice Hall, 1996.