Module Master M1

Systèmes temps réel et Informatique Industrielle
Chapitre III : Systèmes d'exploitation temps réel

Présenté par : Prof. Kholladi Mohamed-Khireddine
Département d'Informatique
Facultés des Sciences Exactes
Université Echahid Hamma Lakhdar d'El Oued
Tél. 0770314924
Email. kholladi@univ-eloued.dz et kholladi@yahoo.fr
Site Web. www.univ-eloued.dz
http://kholladi.doomby.com/ et http://kholladi.e-monsite.com/

III - Systèmes d'exploitation temps réel

III.0 – Rappel

a - Système d'exploitation

La figure III.1 illustre la position du système d'exploitation entre les utilisateurs et le matériel. En plus, on les différentes composantes du système d'exploitation. On constate que les utilisateurs effectuent des appels système et reçois les interruptions du matériel.

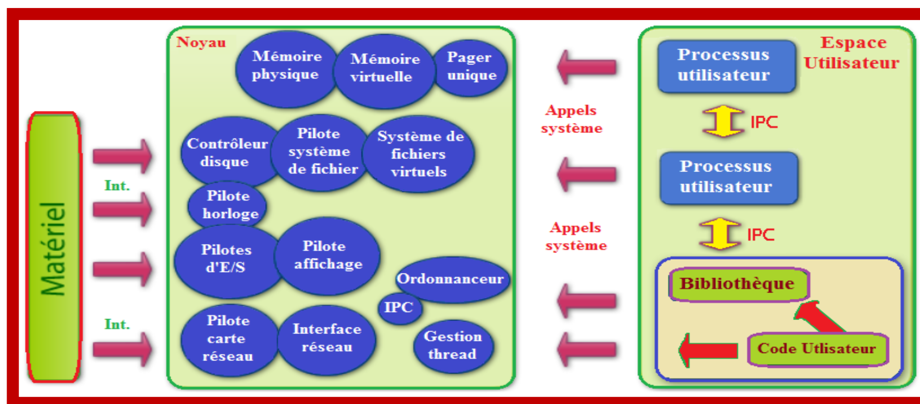


Figure III.1 – Le système d'exploitation intermédiaire entre le matériel et les utilisateurs

b - Organisation générale d'un système d'exploitation Temps réel (OS TR)

Le noyau d'un système d'exploitation (kernel en anglais) : partie fondamentale de l'OS qui gère les ressources de l'ordinateur et permet aux différents composants matériels et logiciels de communiquer entre eux. Le noyau fournit des mécanismes d'abstraction du matériel, notamment de la mémoire, du (ou des) processeur(s), et des échanges d'informations entre logiciels et périphériques matériels (voir la figure III.2).

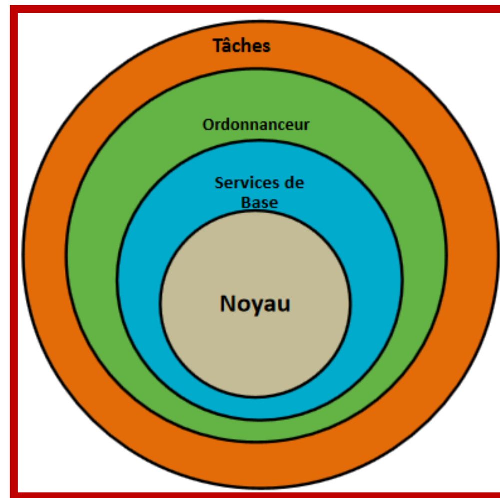


Figure III.2 – Organisation générale d'un système d'exploitation Temps réel

c - Notion de zone critique

Une zone critique peut être une structure de données, une ressource partagée, une section de code au timing délicat, une zone de mémoire. Une zone critique ne peut être interrompue une fois commencée. Il faut donc désactiver les interruptions à l'entrée de la section critique et les réactiver à la sortie. Une seule tâche a le droit d'accéder à une zone critique. Elle doit être protégée pour que les autres tâches ne puissent modifier les données ou l'état du matériel. La technique des sémaphores doit donc être utilisée.

c - Notion d'exclusion mutuelle

Lorsqu'une ressource n'est pas partageable entre plusieurs tâches, un mécanisme doit en assurer l'accès exclusif. Un tel mécanisme est appelé exclusion mutuelle : lorsqu'une tâche utilise la ressource, aucune autre tâche ne peut l'utiliser. Par exemple, une imprimante peut être utilisée par plusieurs processus (Word, Excel, et autres logiciels) mais son accès est exclusif. Sinon, on pourrait avoir un mélange de fichiers si plusieurs processus se mettent à imprimer en même temps.

d - L'exclusion mutuelle

Elle peut être réalisée de différentes façons

1. Le verrouillage matériel est réalisé à l'aide d'un drapeau binaire qui indique si la ressource est libre ou utilisée. Chaque tâche va tester le drapeau avant d'utiliser la ressource, si elle est déjà utilisée par une autre tâche, elle va se mettre en attente,

2. Le verrouillage du CPU permet d'éviter l'accès concurrent à une ressource non partageable. La distribution du CPU d'une tâche à l'autre est interdite pendant cet accès.
3. Les services du noyau tels que les sémaphores et les mutex.

e - Notion d'ordonnanceur

L'ordonnanceur est le composant du noyau responsable de la distribution du temps CPU entre les différentes tâches. Il est basé sur le principe de la priorité : chaque tâche possède une priorité dépendant de son importance (plus une tâche est importante, plus sa priorité est élevée). Il attribue le processeur à la tâche exécutable (non dormante et non bloquée) de plus haute priorité. Lorsque qu'une tâche T_2 plus prioritaire que la tâche active T_1 passe de l'état bloqué à l'état exécutable, T_1 est suspendue et l'ordonnanceur attribue le processeur à T_2. C'est le principe de la préemption. La préemption permet un temps de réponse des tâches optimum.

f - Notion de changement de contexte

Quand l'ordonnanceur doit transférer l'usage du processeur d'une tâche à l'autre, il opère un changement de contexte. Le contexte représente l'état du processeur à un moment donné. La tâche suspendue doit pouvoir continuer son exécution sans être affectée, la première opération à effectuer est donc la sauvegarde de l'état du processeur au moment de la suspension. Le noyau possède pour chaque tâche non dormante un espace mémoire réservé à cet effet. Les données temporaires utilisées par la tâche suspendue doivent être préservées lors des opérations de la nouvelle tâche active. Ces données sont organisées sous forme de pile contenant le contexte des appels de sous routines en cours (adresse de retour, valeur des registres) ainsi que les paramètres et les variables temporaires de ses sous-routines. Une tâche peut être suspendue à tout moment et l'usage du processeur transféré vers une autre tâche susceptible d'appeler des sous-routines et d'allouer des variables temporaires. Chaque tâche possède sa propre pile. Le temps requis pour un changement de contexte est déterminé par le nombre de registres à sauvegarder et à restaurer par le CPU.

g - Notion de Processus

Le mot processus vient du latin *pro* (au sens de vers l'avant) et de *cessus*, *cedere* (aller, marcher) ce qui signifie donc aller vers l'avant, avancer. Le terme processus est employé dans plusieurs domaines avec une signification spécifique à chacun des domaines comme la biologie, l'écologie, l'informatique, les mathématiques, la physique, la chimie, l'armée, etc. Un processus informatique est une tâche en cours d'exécution. On appelle aussi processus

l'image de l'état du processeur et de la mémoire au cours de l'exécution d'un programme. Le concept processus est le plus important dans un système d’exploitation (SE). Tout le logiciel d’un ordinateur est organisé en un certain nombre de processus (processus système, processus utilisateur). Un processus est un programme en cours d’exécution, univoquement identifié par le système par un entier positif appelé process id ou pid. Un processus est caractérisé par un contexte d’exécution :

- Segments de texte, de données et pile,
- Compteur spécifiant l’instruction suivante à exécuter,
- États des registres.

Afin de permettre les changements de contexte (context switch), l’information qui concerne un processus est stockée dans un bloc de contrôle (Process Control Block ou PCB). Dans la plupart des systèmes d’exploitation, le PCB est composé de deux zones :

- La première contient les informations critiques dont le système a toujours besoin (jamais swappée);
- La deuxième contient les informations utilisées uniquement lorsque le processus est à l’état élu.

Dans L’arbre des processus comme sur la figure III.3, tout processus a un processus père, sauf le processus init(pid_1), lancé au démarrage de l’ordinateur. Si un processus père est terminé, ses fils sont adoptés par le processus 1. Le système gère à tout moment un arbre de processus.

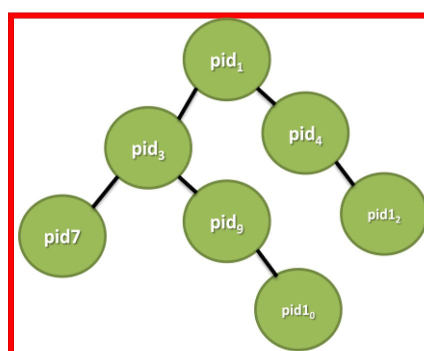


Figure III.3 - Arbre des processus

h - Etats d’un processus

Les différents états d’un processus sont :

- Nouveau : il est en cours de création

- Actif : ses instructions sont en cours d'exécution, le processus dispose de toutes les ressources dont il a besoin.
- Bloqué (En attente) : le processus attend qu'un évènement se produise ou a besoin d'au moins une ressource autre que le processeur physique.
- Prêt : il attend d'être affecté à un processeur (dispose de toutes les ressources à l'exception du processeur physique).
- Terminé : le processus a fini l'exécution

Il est important de signaler qu'un seul processus peut être actif sur n'importe quel processeur à tout moment. Toutefois, plusieurs processus peuvent être prêts et bloqués. Le passage de l'état actif à l'état bloqué est en général volontaire, ou à la suite d'une réquisition. Le passage de l'état bloqué à l'état prêt est dû à une action externe au processus. La figure III.4 illustre les différences états d'un processus.

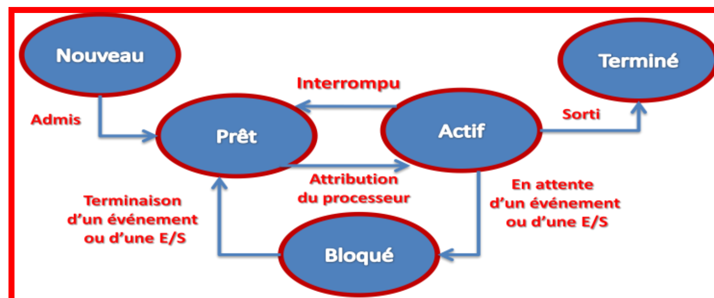


Figure III.4 – Les différents états d'un processus

i - Commutation de l'unité centrale (UC) entre les processus P_i et P_k

Commutatio : procédure noyau en mode privilégié, pose du point de reprise pour le processus interrompu et chargement du point de reprise du processus élu comme sur la figure III.5.

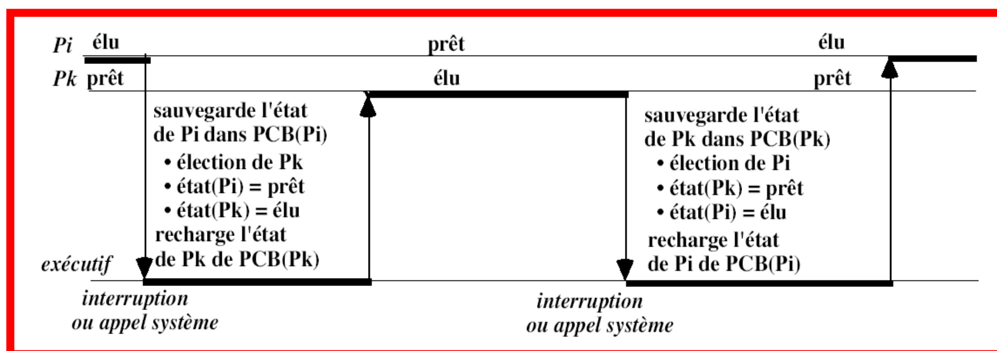


Figure III.5 – Commutation de l'UC entre deux processus

j - Passage de l'environnement de P_i à celui de P_k (schéma pour Unix, Linux, Posix)

La figure III.6 illustre le passage de l'environnement entre deux processus dans le monde Unix ou Lunix.

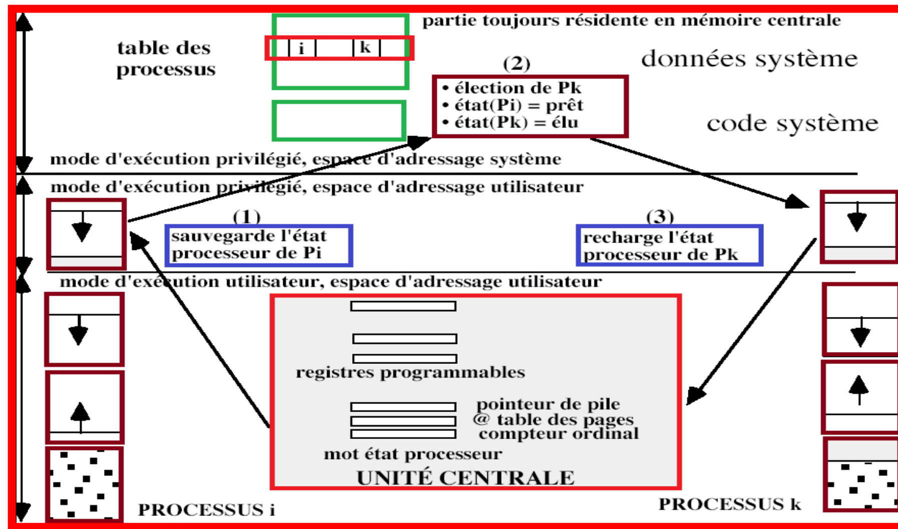


Figure III.6 – Passage de l'environnement entre deux processus

k - Processus

Un processus est un programme en cours d'exécution. Il est caractérisé par :

- Un numéro d'identification unique (PID);
- Un espace d'adressage (code, données, piles d'exécution);
- Un état principal (prêt, en cours d'exécution (élu), bloqué, …);
- Les valeurs des registres lors de la dernière suspension (CO, PSW, Sommet de pile…);
- Une priorité;
- Les ressources allouées (fichiers ouverts, mémoires, périphériques …);
- Les signaux à capturer, à masquer, à ignorer, en attente ainsi que les actions associées;
- Autres informations indiquant, notamment, son processus père, ses processus fils, son groupe, ses variables d'environnement, les statistiques et les limites d'utilisation des ressources, etc.

Le système d'exploitation maintient dans une table appelée table des processus les informations sur tous les processus créés (une entrée par processus : bloc de contrôle de processus PCB). Cette table permet au système d'exploitation de localiser et de gérer tous les processus.

k.1 - Descripteurs de processus

L'existence d'un processus est matérialisée par une structure de données contenant :

- l'état,
- le nom externe,
- l'identificateur du propriétaire,
- les données d'ordonnancement (priorité, etc.),
- les pointeurs vers l'image mémoire,
- les données de comptabilité.

Les descripteurs de processus sont regroupés dans une table des processus.

k.2 - **Table des processus**

La figure III.7 illustre la structure des tables des processus.

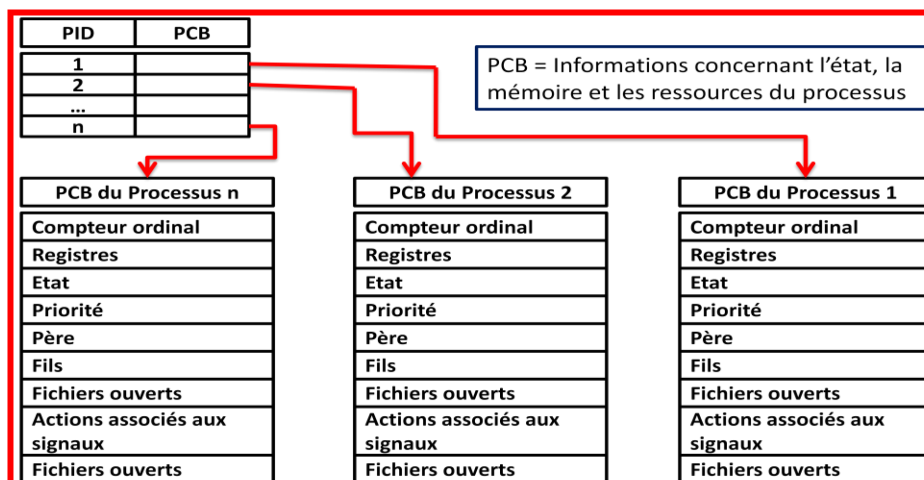


Figure III.7 – Tables des processus

k.3 - Opérations de base sur les processus

- Identification
- Création de nouveaux processus
- Terminaison
- Suspension
- Interne (sleep)
- Externe (attente d'une ressource, wait)
- Chargement de code

- Gestion (statut, accounting, etc.)

k.4 - Identification des processus

Le tableau suivant indique les classes de processus pour l'identification. L'administrateur système (root) a l'uid 0. Le processus init a le pid 1.

Identification	Significations
pid	Entier positif unique identifiant le processus.
ppid	pid du processus père.
uid	Identifiant de l'utilisateur réel (celui qui a lancé le processus).
euid	Identifiant de l'utilisateur effectif (au moment où l'instruction est exécutée).
gid	Identifiant du groupe de l'utilisateur réel.
egid	Identifiant du groupe de l'utilisateur effectif.

l - Opérations de base sur les processus

l.1 - Création : pid_t fork(void)

Demandée par un autre processus.

- Allocation et initialisation d'un descripteur,
- Copie en mémoire du programme à exécuter ➔ arbre des processus en cours.

Créé dans l'état prêt ou suspendu.

Un processus (père) peut en créer un autre (processus fils) par l'appel système fork(). L'appel fork() duplique le contexte du processus père. Il retourne la valeur 0 au fils, et le pid du fils au père comme sur la figure III.8.

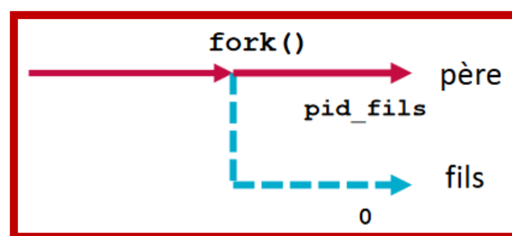


Figure III.8 – Appel Fork ()

l.2 - Terminaison de processus

Quatre possibilités :

1. Arrêt ou sortie normale volontaire (appel exit() ou fin de main())
2. Arrêt pour erreur ou suicide volontaire (erreur interne, appel abort() ou kill() à soi-même.

3. Arrêt pour erreur fatale involontaire (erreur externe, kill() par l'O.S.)
4. Processus est arrêté ou tué par un autre involontaire (erreur externe, appel – système kill() par un autre processus)

I.3 - Destruction

Peut être demandée par :

- Le processus lui-même,
- Un autre processus,
- Le noyau.

Libération des ressources associées et génération d'un événement.

I.4 - Blocage

Passage en mode bloqué en attente d'un événement externe (condition logique).

- Fin d'une entrée/sortie
- Disponibilité d'une ressource
- Interruption

Peut être demandé par le processus ou forcé par le système → préemption.

I.5 - Déblocage

Passage en mode prêt d'un processus bloqué lorsque l'événement attendu se produit.

I.6 - Activation

Passage en mode exécution d'un processus prêt.

II - Processus et Threads

Deux notions similaires, implémentation imbriquée :

a - Processus : Objet lourd

- Espace mémoire virtuelle séparé, etc.
- Droits d'accès par processus.

b - Thread : Objet plus léger

- Inclus dans un processus,
- Tous les threads partagent l'espace mémoire virtuelle,
- Une pile par thread,

- Un seul utilisateur pour tous les threads d'un processus.

  En français : thread ↔ fil

Les processus légers ou threads doivent permettre plusieurs flots de contrôle séquentiels concurrents dans un processus, de partager le maximum d'information entre ces flots (donc diminuer le volume mémoire nécessaire), d'assurer une latence de création nettement plus réduite que les processus et d'autoriser l'ordonnancement au niveau des threads.

o - Notion de sémaphore

Le concept de sémaphore est utilisé pour contrôler l'accès à une ressource. Lorsqu'une tâche accède à une ressource non partageable, le sémaphore à l'entrée de celle-ci devient bloqué et le reste tant que la tâche n'a pas relâché la ressource. Il empêche ainsi tout autre tâche à accéder à cette ressource. Par exemple, dans la signalisation ferroviaire, il ne doit y avoir qu'un train au maximum par tronçon de voie ferrée. Lorsqu'un train entre dans ce tronçon, aucun autre train ne peut y entrer tant que le premier train ne l'a pas quitté. Un sémaphore peut être généralisé au cas où une ressource est accessible par n tâches simultanément. Dans ce cas, le sémaphore peut prendre n + 1 états qui seront représentés par un compteur. Un sémaphore possède une valeur entière s qui représente le nombre de tâches qui accèdent à la ressource qu'il surveille. Sa valeur est positive ou nulle et est uniquement manipulable à l'aide de deux opérations : wait(s) et sinal(s).

- wait(s) est l'action d'attente. Si le sémaphore est libre (s supérieur à 0), la tâche continue son exécution et la valeur de s est décrémentée.
  - Si le sémaphore est bloqué (s = 0), la tâche est placée à la fin de la file d'attente du sémaphore.
- sinal(s) est l'action de signalisation. Si le sémaphore est libre, il reste libre et sa valeur est incrémentée.
  - Si le sémaphore est bloqué et qu'aucune tâche n'est en attente, il devient libre.
  - Si le sémaphore est bloqué et qu'au moins une tâche est en attente, alors la première tâche de la file d'attente du sémaphore est débloquée.

p - Notion de Mutex

Un sémaphore qui ne peut prendre que deux états (libre et bloqué) est appelé sémaphore binaire ou mutex. Dans ce cas, une seule tâche peut avoir accès à la ressource. Un mutex est un sémaphore binaire, garantissant l'exclusion mutuelle.

III.1 – Introduction

a - Conception de RTOS (Real Time Operating System)

Deux approches :

- Extension d'OS généraux par des fonctionnalités TR (notamment ordonnanceurs adaptés, timers) :
  - Ex. Linux-rt, RTX Real-Time Extension for Windows.
- OS spécialisés (Exécutifs temps réel) :
  - Noyaux rapides, petits et adaptés,
  - Ex. QNX, VxWORKS

Avantages pour les deux approches :

- Extensions OS : compatibilité avec l'existant, terrain connu,
- OS spécialisés : performance, adaptation aux besoins.

b - Système d'exploitation temps réel

Il doit fournir à l'utilisateur un environnement lui permettant de mettre en œuvre facilement son application, en lui cachant un certain nombre de problèmes (gestion des périphériques, des fichiers, des interruptions, etc.) et être constitué d'un ensemble de primitives chargées de fournir cette fonctionnalité :

- Souvent (mais pas toujours) exécutées en mode noyau (appels système ou system calls),
- Interfacées avec l'utilisateur par des fonctions dites systèmes (de la libc dans les environnements à la UNIX).

Il doit avoir des fonctionnalités spécifiques pour gérer les contraintes temporelles :

- Le comportement du système doit être prédictible en termes de temps de réponse,
- Il doit fournir les outils pour aider à respecter les échéances.

c - Architecture de l'application : tâches et noyau temps réel

La figure III.9 illustre l'architecture de l'application des tâches avec le noyau temps réel.

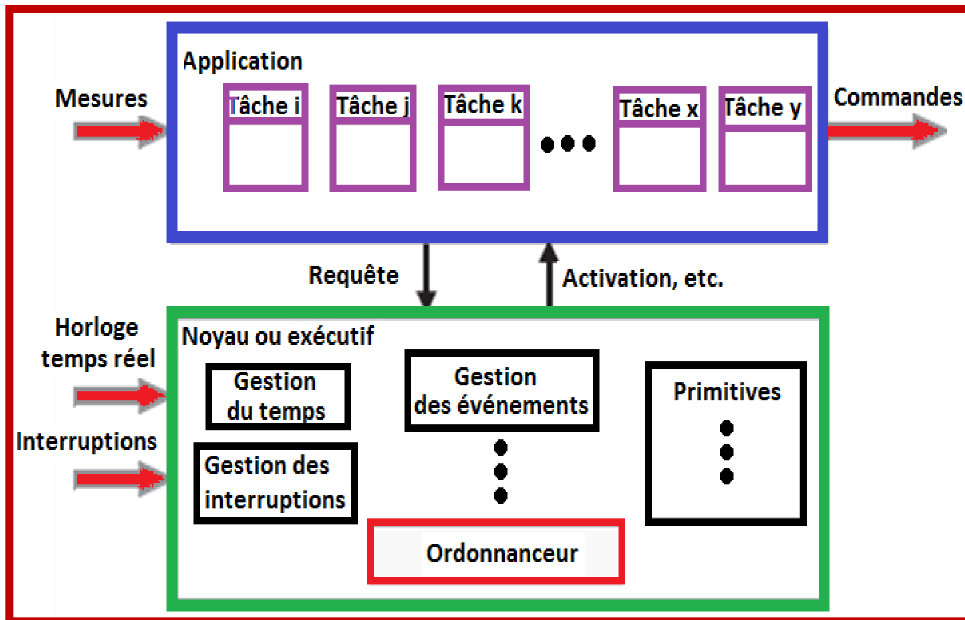


Figure III.9 - Architecture de l'application – Tâches et noyau temps réel

d - Extensions d'OS généralistes : Linux-rt

La figure III.10 illustre les extensions des systèmes d'exploitation généralistes, tel que Linux-rt.

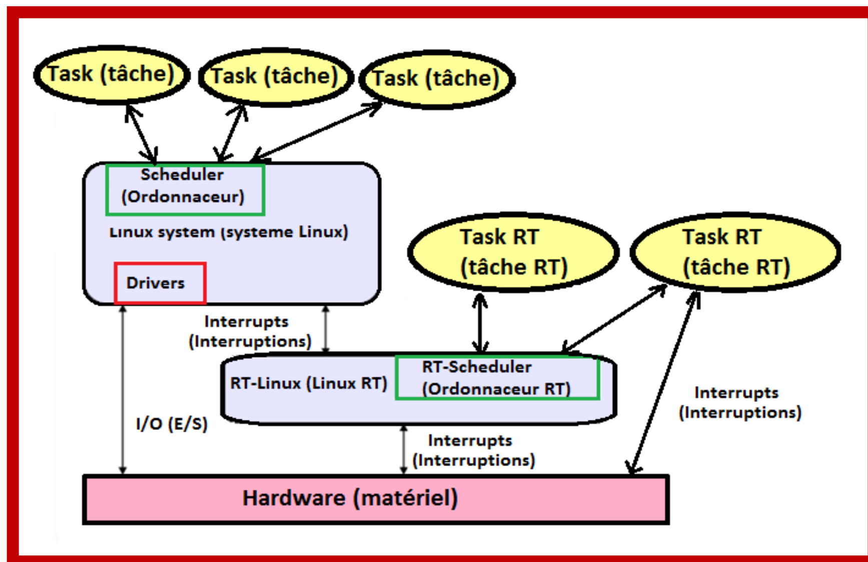


Figure III.10 – Extensions d'OS généraliste

e - Exécutif TR RT-OS spécialisé

La figure III.11 illustre la conception d'un exécutif temps réel (TR-OS) spécialisé.

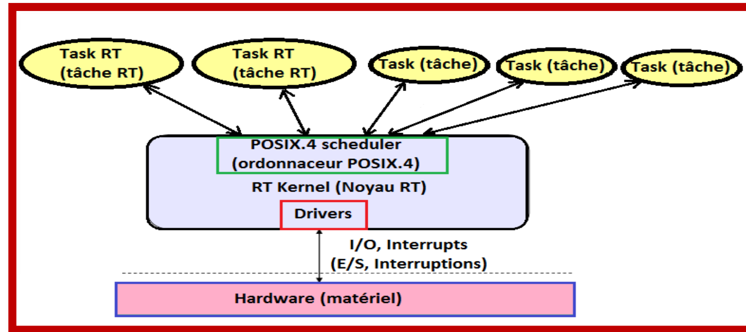


Figure III.11 – Exécutif TR-OS spécialisés

f - Système d'exploitation temps réel

Les langages utilisés pour développer les applications de contrôle-commande avec un noyau temps réel comme sur la figure III.12:

1. Gestion des interruptions,
2. Ordonnancement,
3. Relations entre les tâches).

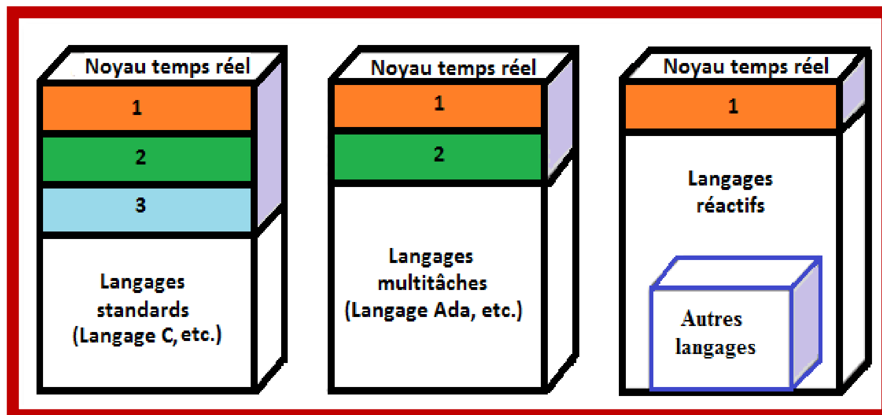


Figure III.12 – Système d'exploitation temps réel

III.2 - Les systèmes du marché

a – Les systèmes d'exploitation temps réel

Il existe plusieurs dizaines de systèmes d'exploitation temps réel

- La plupart sont des **OS** généralistes :
  - VxWorks,
  - QNX,
  - ChorusOS (VirtualLogix),
  - pSOS+,

  - VRTX,
  - Windows CE,
  - Xenomai.
- Beaucoup plus de systèmes spécialisés développés dans le cadre d'un secteur d'activité (automobile, avionique, téléphonie, etc.) :
  - OSEK/VDX,
  - IPhone, Android, Windows Phone 7.
- Ouverts ou propriétaires.

b - Motivations pour des systèmes généralistes

La plupart des applications ont besoin de services **généraux** :

- Accès à des fichiers,
- Accès au réseau,
- Souci de minimiser les phases de développement,
- Minimiser les coûts,
- Réutilisation des composants.

Il y a un souci de portabilité des applications, indépendance vis-à-vis de la plate-forme d'exécution.

c - Motivations pour des systèmes généralistes

- Performances :
  - Adéquation au matériel utilisé.
- Utilisations très spécifiques.
- Marché captif :
  - Automobile,
  - Avionique,
  - Télécoms.
- Quand le coût n'est pas un argument, etc.

III.3 - La norme POSIX

Les services sont fournis par des fonctions spécifiques du système d'exploitation. La norme **POSIX** propose une interface assurant la portabilité des applications :

- Portable Operating System Interface,

- Dans un environnement à l'UNIX,
- Développée par l'IEEE.

Pour le temps réel

POSIX1.b :

- Ordonnancement,
- Signaux,
- Timers, horloges,
- Communication.

POSIX1.c :

- Threads.

III.4 - Architecture d'un RTOS

Dans la plupart des cas, on a un noyau ou micronoyau fournissant les fonctions de base et des couches logicielles fournissant les fonctions plus élaborées, comme sur la figure III.13.

Figure III.13 – Architecture d'un RTOS

III.5 - Prise de contrôle par le noyau

La figure III.14 illustre le mécanisme de prise de contrôle par le noyau.

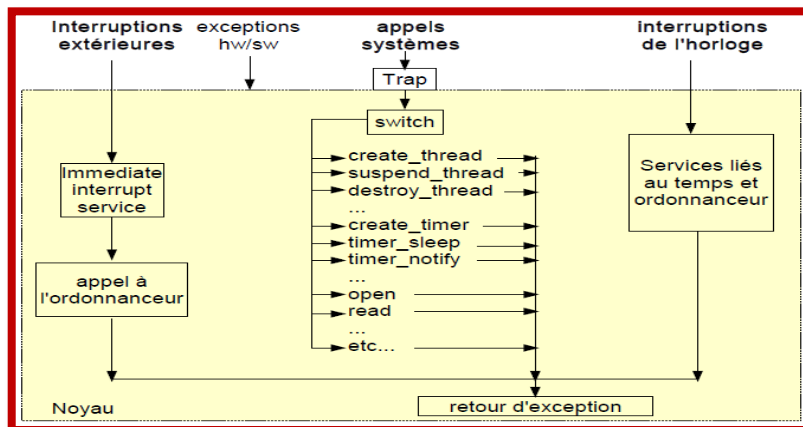


Figure III.14 – Mécanisme de prise de contrôle par le noyau

III.6 - Appels systèmes

Les fonctions exécutées par le noyau au nom d'une tâche utilisateur permettent d'accéder de façon sure à des ressources privilégiées. Ce qui implique un changement de contexte d'exécution par la sauvegarde du contexte d'exécution de la tâche et le passage de la CPU en mode privilégié noyau. Une fois la fonction terminée, le noyau exécuté un retour d'exception, la CPU repasse en mode normal et l'ordonnanceur est appelé et la tâche de plus haute priorité prend la CPU. Dans beaucoup de processeurs embarqués, il n'y a pas de mode privilégié et l'appel système est traité comme une procédure ordinaire.

III.7 - Services liés au temps

L'ordonnancement des activités est au cœur du travail du noyau mis en œuvre périodiquement et dès qu'une tâche se termine. L'activation périodique de l'ordonnanceur est provoquée par une horloge dédiée (pas celle de la CPU), qui génère une interruption à chaque tick (~ milliseconde ou ms). À chaque interruption, le noyau gère tous les timers, en déclenchant éventuellement les actions liées aux timers qui viennent d'expirer, gère les budgets de temps d'exécution des tâches et met à jour la liste des tâches prêtes et invoque l'ordonnanceur.

III.8 - Interruptions extérieures

Le plus souvent dues à l'interaction avec du matériel :

- Occurrences d'évènements externes liés à des activités sporadiques dues à des entrées/sorties, tels Capteurs, interfaces réseaux, périphériques (disques),
- Temps de réponse très variables.
- Prise en charge par la CPU en passant par un mode spécial interruption :
    - Niveaux de priorités dépendant de l'origine de l'interruption,
    - Désactivation de la prise en compte des interruptions,
    - Sauvegarde du contexte d'exécution et branchement à un point spécial de l'OS,
    - Réactivation des interruptions et exécution d'une routine dédiée (ISR),
    - Reprise du cours normal d'exécution.
- Si une interruption de priorité supérieure se produit pendant l'exécution de l'ISR, elle est prise en charge immédiatement par le système ou sinon, on attend la fin du traitement de l'interruption en cours.

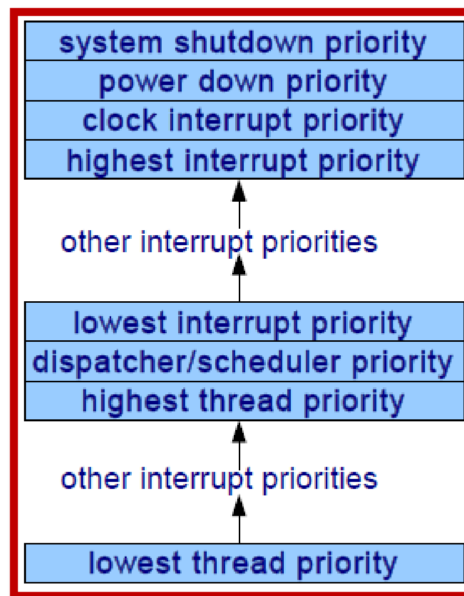


Figure III.15 – Interruptions extérieures

En cas de partage de la même ligne d'interruptions (fréquent sur les systèmes embarqués) :

- Polling des dispositifs qui partagent la ligne : le dispositif responsable de l'interruption se signale à l'OS,
- Ou bien utilisation d'un vecteur d'interruption : Le dispositif responsable passe à l'OS un vecteur contenant l'adresse spécifique à laquelle se brancher.
- Le plus souvent l'ISR ne traite pas complètement l'interruption : temps le plus court possible dans le mode interruption et réveil d'une tâche dédiée (dans l'espace noyau ou utilisateur et ordonnancée).

III.9 - Services liés au temps

Au moins une horloge présente dans les RTOS, à ne pas confondre avec l'horloge de la CPU est dans les systèmes POSIX : CLOCK_REALTIME. Pour chaque horloge, on a :

- Compteur,
- Liste de timers,
- Traitant d'interruption (handler) pour incrémenter le compteur et voir quels timers ont expire.

La résolution est liée à la capacité à traiter les interruptions, de 10 ms à quelques centaines de microsecondes (µs) et peut être très améliorée dans certaines architectures en utilisant l'horloge de la CPU (Time Stamp Counter). POSIX1.b : chaque thread/processus peut

avoir ses propres horloges et timers, bases sur des temps absolus ou relatifs, synchrones ou asynchrones.

III.10 - Services liés à l'ordonnancement

Implémentation d'algorithmes d'ordonnancement

- Algorithmes à priorités fixes : SCHED_FIFO et SCHED_RR.
- Algorithme à priorités variables : EDF.
- Blocage de la préemption.
- Le plus souvent par inhibition de l'ordonnanceur.
- Pendant un temps le plus court possible.
- Gestion des serveurs de tâches apériodiques.

III.11 - Communication et synchronisation

a - Fonctionnalités essentielles pour l’activité des tâches

- Communication : échange d'informations de contrôle et échange d'informations de données.
- Synchronisation pour définir le moment et les conditions où ces échanges peuvent avoir lieu.
- Communication par files de messages : simple et efficace, possibilité de communiquer dans un environnement distribue, possibilité de synchronisation par des mécanismes d’écriture et de lecture bloquants, possibilité de notification asynchrone d’écriture dans une file vide, possibilité de définir des niveaux de priorité des messages, et possibilité d’implémenter un protocole d’héritage de priorité.
- Communication par mémoire partagée : très rapide, nécessite des moyens de synchronisation, possible de communiquer dans un environnement distribué (projection de fichiers sur un disque), complexe et très risqué !, etc.
- Mécanismes de synchronisation : essentiels dans un environnement multithread.
- Plusieurs mécanismes : sémaphores, mutex, variables conditionnelles, et verrous lecteurs/écrivain.
- Peuvent permettre la mise en place des protocoles d’héritage de priorité ou de priorité plafonnée pour prendre en compte les problèmes d'inversion de priorité ou de deadlock.

III.12 - Notification par événement

- Nécessaire de notifier de façon asynchrone l'occurrence de faits importants : expiration d'un timer, réception d'un message, terminaison d'une entrée-sortie asynchrone, etc.
- Le plus souvent (POSIX) par le mécanisme de signaux : asynchronous Procedure Calls sous Windows.
- Notification par un mécanisme analogue à celui des interruptions, quand le thread est notifié par un signal :
  - Il interrompt le cours de son exécution,
  - Le processeur passe en mode interruption,
  - Un traitant (handler) exécuté,
  - Le processeur repasse en mode normal et invoque l'ordonnanceur.
- La notification peut aussi s'effectuer de façon synchrone et le thread qui attend l'événement se bloque jusqu'à l'occurrence du signal.
- Limitations des signaux : pas nombreux, pas de mise en queue, pas de déterminisme dans l'ordre de prise en compte et ne transportent pas d'information.
- En partie résolu par les signaux temps réel de POSIX.

III.13 - Études de cas

- VxWorks
- OSEK/VDX
- Extensions Temps Réel pour Linux : Xenomai

III.13.1 - Études de cas : VxWorks

RTOS (système d'exploitation temps réel) propriétaire produit par Wind River (filiale d'Intel) :

- Le plus utilisé dans le monde de l'embarqué,
- Rendu célèbre par la mission Pathfinder sur Mars,
- Équipe la plupart des missions NASA (Curiosity),
- Dernière version : VxWorks 7 (février 2014).

Il y a peu d'informations sur l'architecture du système d'exploitation (OS) :

- Non UNIX, mais interface conforme à POSIX
- Construit autour d'un micronoyau (WIND)

- Les applications, les protocoles de communication sont complètement séparés du noyau: évolution du système plus facile et sure,
- Accent mis sur la connectivité,
- Supporte les architectures multiprocesseurs, symétriques et asymétriques, aussi bien que monoprocesseur,
- Supporte les architectures avec ou sans MMU.
- Faible empreinte mémoire : 20 kB pour le micronoyau et adaptable à la configuration utilisateur.
- Disponible sur de nombreuses architectures :
    - ARM (9, 11),
    - Intel Pentium (2, 3, 4, M),
    - Intel XScale (IXP425, IXP465),
    - MIPS (4K, 5K, ...),
    - PowerPC,
    - SuperH (4, 4a).

III.13.2 - Études de cas : Linus

- Linux n'est pas un RTOS (système d'exploitation temps réel).
- Points forts :
    - Fiable,
    - Bon marché,
    - Performant,
    - Portable (conforme à POSIX),
    - Ouvert aux autres systèmes.
- Points faibles :
    - Empreinte,
    - Opensource (licence GPL).
- Il existe des solutions pour faire de Linux un RTOS :
    - Adjonction d'un co-noyau temps réel,
    - Modifier Linux pour avoir un noyau entièrement préemptibles.

a - Approche noyau entièrement préemptibles

- Modifier le noyau pour que les tâches non temps réel et dont les temps d'exécutions ne sont pas connus n'interfèrent pas avec les tâches temps réel.

- Complexe (le noyau est gros).
- Mais seuls le cœur du noyau et quelques pilotes doivent être modifiés (travail d'experts).
- Patch PREEMPT_RT.

b - Real Time Linux Wiki (https://rt.wiki.kernel.org/)

- Prise en compte des interruptions par des threads noyaux,
- Implémentation de l'héritage de priorité,
- Remplacement des spinlocks par des mutex,
- Mise en place de compteurs à haute précision pour exprimer les délais et les échéances,
- Approche généralement suffisante pour le temps réel mou ou ferme.

c - Approche Co-noyau

- Solution classique (a existé même pour Windows : RTX),
- Impossible de faire confiance au noyau GPOS pour les aspects temps réel : délègue ceux-ci à un noyau spécialisé et le noyau Linux continue à servir les tâches classiques.
- Le noyau temps réel intercepte toutes les interruptions matérielles et les traite avant de les passer éventuellement au noyau Linux (PIC virtuel),
- Linux fonctionne avec une priorité inférieure à celle du noyau temps réel,
- Mais, nécessite de porter sur le noyau temps réel tous les pilotes dont on attend une réponse temps réel et les appels aux fonctions de la glibc peuvent avoir des latences importantes.
- Plusieurs approches : support pour l'exécution des tâches temps réel dans l'espace utilisateur (Xenomai, RTAI) et des tâches temps réel uniquement dans l'espace noyau (modules) : RTLinux/GPL.

III.13.3 - Études de cas : Xenomai

- Basé sur les spécifications ADEOS (Adaptive Domain Environment for Operating Systems) de Karim Yaghmour.
- Projet initialement créé pour faciliter le portage des applications temps réel vers un RTOS de type Linux (virtualisation).
- La version que nous utilisons est Xenomai-2.6.4 (septembre 2014).

- Abstraction des propriétés communes des RTOS traditionnels (couche générique H/W (HAL) et S/W (SAL) et peaux pour émuler les RTOS traditionnels) comme sur la figure III.16.

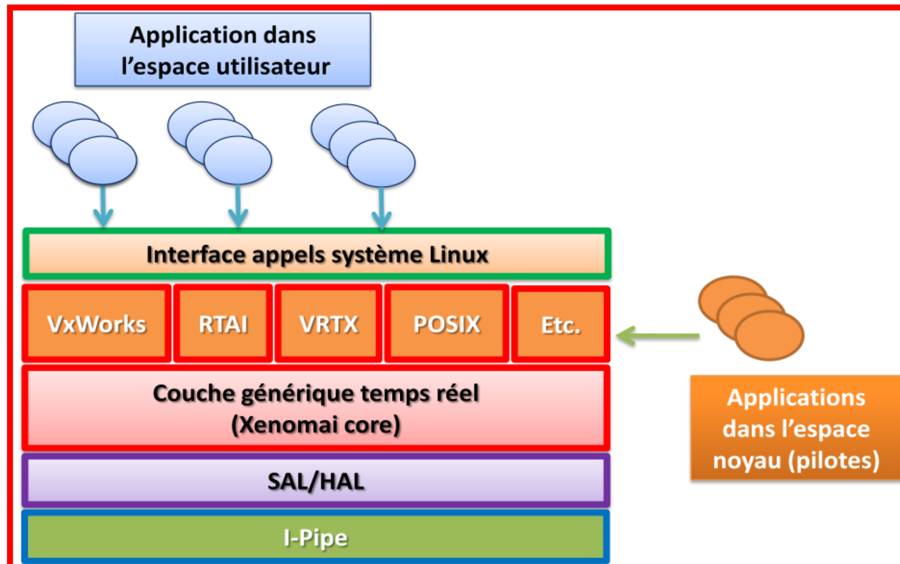


Figure III.16 – Architecture Xenomai

III.13. 4 - Interrupt Pipeline

Pour organiser la distribution des interruptions de façon à ce que le noyau Linux ne retarde pas leur prise en compte par le RTOS :

- Implémentation d'ADEOS,
- Organise un ensemble de domaines connectes par le pipeline : partagent le même espace d'adresses et implémentés sous forme de module comme sur la figure III.17.

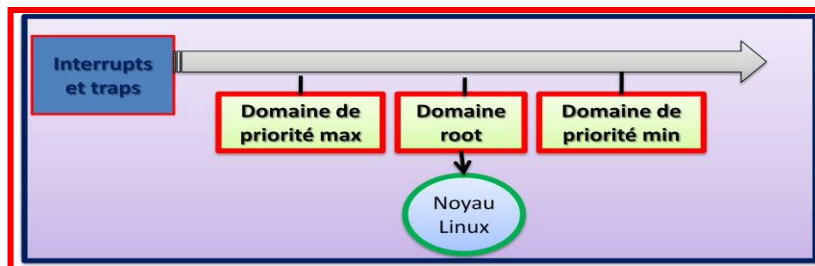


Figure III.17 – Architecture ADEOS

a - Distribution prioritaire des interruptions suivant le schéma de protection optimiste des interruptions

Un domaine peut passer en mode bloqué (stalled) :

- Les interruptions arrivantes ne sont plus distribuées aux handlers du domaine :
- Les interruptions arrivantes sont accumulées et ne sont plus transmises aux domaines de priorité inferieure,
- Les domaines de priorité supérieure continuent à recevoir les interruptions et à les traiter.
- Au déblocage, les interruptions accumulées sont traitées et éventuellement passées aux domaines de priorité inférieure.

b - Les événements systèmes sont propagés par le même pipeline abstrait

- Notifications synchrones d'exceptions (division par zéro, accès mémoire invalide, etc.), d'actions exécutées par le noyau Linux (pagination, etc.),
- Ne peuvent pas être retardées, à l'inverse des interruptions.

III.14.5 - Xenomai Core

Pour fournir toutes les ressources systèmes utilisées par les peaux : briques génériques pouvant être spécialisées et groupées dans un module : le noyau Xenomai.

Il est constitué de :

- Un objet thread temps réel contrôle par un ordonnanceur temps réel (FIFO par niveau de priorité),
- Un objet générique interruption,
- Un objet allocateur de mémoire ayant un temps de réponse prédictible,
- Un objet générique synchronisation dont sont dérivés les sémaphores, mutex, files de messages, etc.
- Un objet gestion du temps.

a - Peaux Xenomai

À partir des fonctions de Xenomai Core, plusieurs API ont été développées :

- Native,
- POSIX (toutes les fonctions de la bibliothèque glibc),
- RTDM (pour le développement de drivers temps réel).

Et aussi :

- VxWorks, pSOS+, VRTX, uiTRON, et RTAI.

La peau native est la plus complète en termes de fonctionnalités directement fournies : toutes ces fonctionnalités peuvent être obtenues avec la peau POSIX (code portable) et on n'étudiera que la peau POSIX voir la figure III.18.

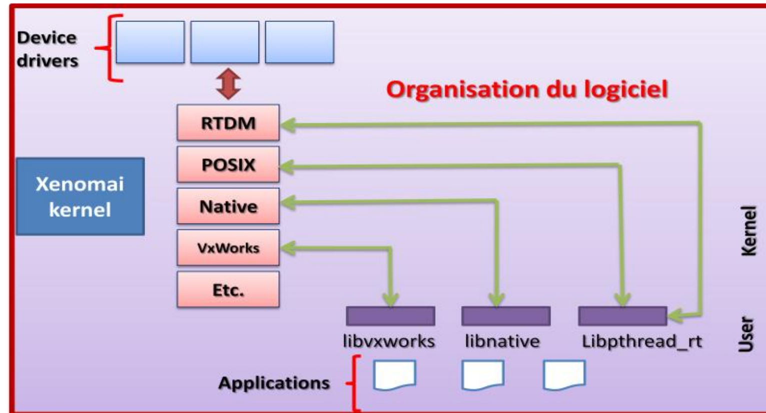


Figure III.18 – Architecture des peaux de Xenomai

III.14.6 - Coexistence de Linux et Xenomai

L'interaction des tâches temps réel avec les tâches non temps réel selon le mécanisme du real time shadowing, pour les tâches non temps réel qui deviennent temps réel, objet RT shadow attaché à la tâche qui peut être alors ordonnancée par l'ordonnanceur de Xenomai, et partage des structures task_struct (spécifique Linux) et xnthread (spécifique Xenomai) par les ordonnanceurs comme sur la figure III.19.

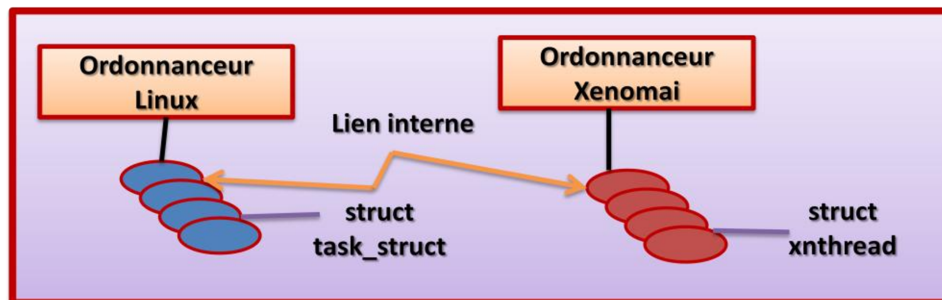


Figure III.19 – Coexistence de Linux et Xenomai

Les threads temps réel peuvent s'exécuter dans l'espace noyau sous forme de modules noyau et dans l'espace utilisateur (threads Xenomai). À ne pas confondre avec les threads Linux classiques, non temps réel, les threads Xenomai peuvent s'exécuter dans le domaine de

plus haute priorité du pipeline (mode d'exécution primaire) et dans l'espace Linux (mode d'exécution secondaire), mais avec des temps de latence d'ordonnancement un peu plus grands. Quand le noyau Xenomai ne sait pas répondre à une demande de service (appel système), c'est Linux qui prend le controle du thread, qui passe donc en mode secondaire. Pour que les threads Xenomai en mode secondaire puissent être temps réel, il faut un schéma de priorités commun pour le noyau temps réel et le noyau Linux, si un thread Xenomai en mode primaire passe en mode secondaire, le noyau Linux va hériter de sa priorité. Un thread Xenomai en mode primaire ne va préempter un thread en mode secondaire que si sa priorité effective est supérieure. Un thread Linux en mode SCHED_FIFO sera toujours préempté par un thread en mode primaire et sera en compétition avec les threads en mode secondaire. Des temps d'exécution prédictibles par le blocage des interruptions destinées à Linux par un domaine ADEOS intermédiaire : le bouclier à interruptions (interrupt shield), active dès qu'un thread Xenomai tourne en mode secondaire. Un noyau Linux avec un ordonnancement à grain fin pour les threads Xenomai en mode secondaire. La gestion des inversions de priorité au niveau du pipeline Xenomai comme sur la figure III.20.

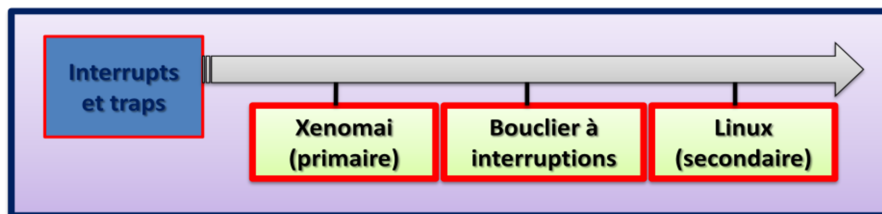


Figure III.20 – Architecture du pipeline Xenomai

a - Propagation des interruptions

Elle doit faire appel au service de propagation pour toutes les interruptions prises en compte, Le noyau temps réel de Xenomai reçoit en premier les interruptions :

- Il les traite,
- Il les marque pour être passés dans la suite du pipeline,
- Quand le dernier traitant est terminé, Xenomai donne la CPU au thread Xenomai de plus haute priorité.

Quand il n'y a plus de threads Xenomai à exécuter, la CPU est donnée au bouclier qui va les passer au noyau Linux s'il est désactivé, ou les bloquer s'il est activé. Deux modes de propagation pour ADEOS, par domaine et par interrupt (implicite ou explicite), Xenomai utilise le mode explicite :

- Chaque traitant,
- Si aucun traitant n'a été déclaré, l'interruption est automatiquement passée au noyau Linux.

III.14.7 – RTDM

Couche intermédiaire entre une application temps réel et les services fournis par un pilote : les pilotes à protocole (communication par échange de message) réseaux temps réel et bus CAN ainsi que les pilotes pour des périphériques nommés.

III.14.8 - Xenomai : développements récents comme sur la figure III.21

- Xenomai-3 : première release en octobre 2015.
- Deux possibilités pour l'utilisateur :
  - L'approche dual core (dite Cobalt) ressemblant à Xenomai-2 mais avec beaucoup d'améliorations en interne,
  - Une nouvelle approche (dite Mercury) basée sur le noyau Linux natif (éventuellement patch avec PREEMPT_RT).
- Les deux approches utilisent une nouvelle interface (dite copperplate) entre les API.

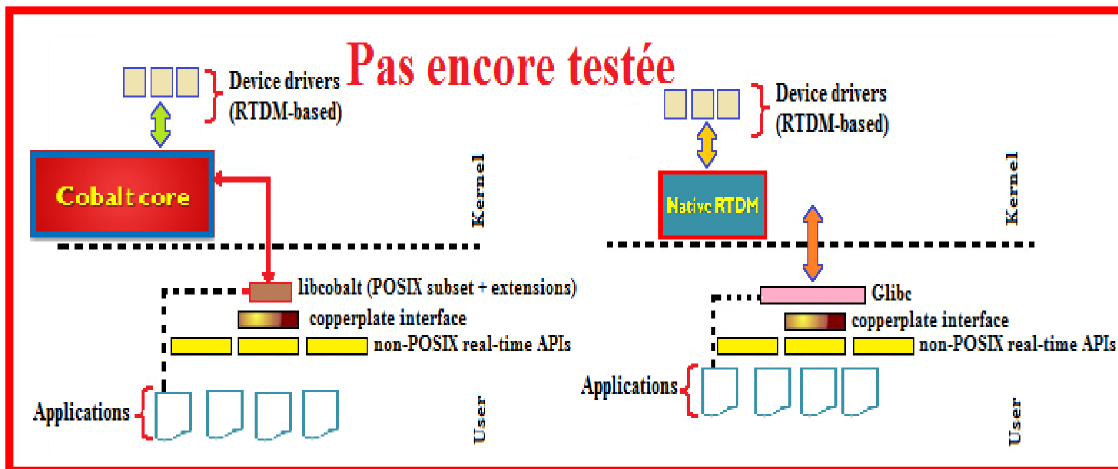


Figure III.21 – Développement récent de Xenomai

III. 15 - OSEK/VDX (comme sur la figure III.22)

C'est un exécutif développé dans le cadre des applications embarquées pour le contrôle automobile (**automotive**). Voir le site web : http://www.osek-vdx.org.

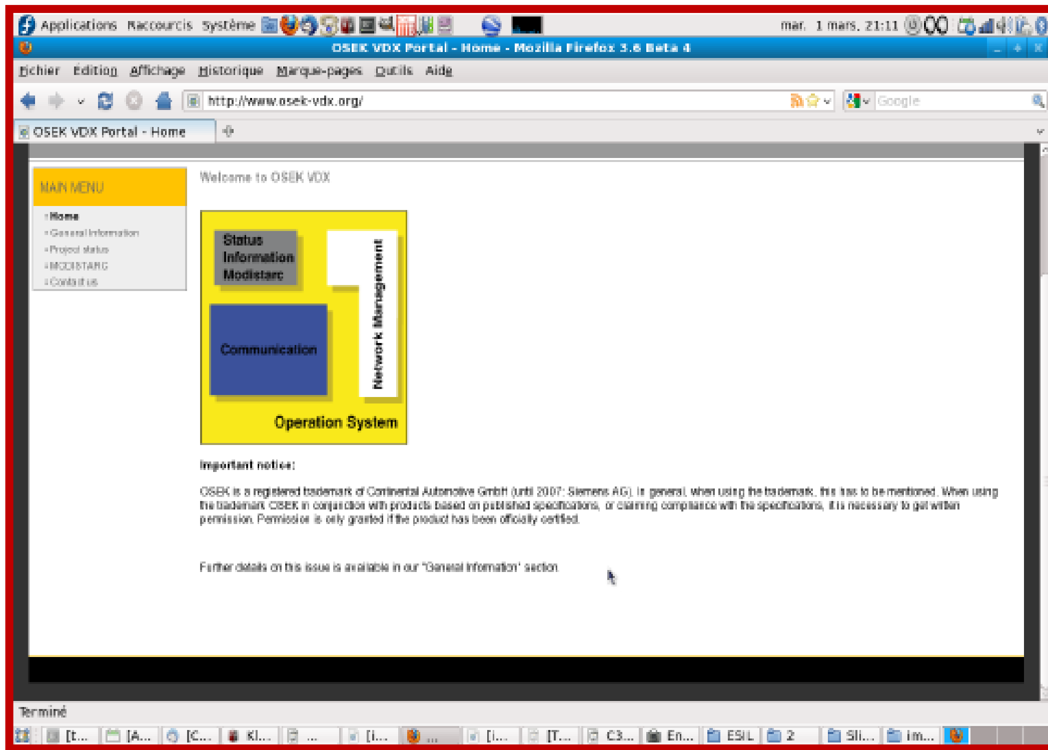


Figure III.22 – Portail d'OSEK/VDX

Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (BMW, Bosch, Daimler Chrysler, Opel, Siemens, VW et l'université de Karlsruhe) :

- Open Systems and the Corresponding Interfaces for Automotive Electronics,
- Systèmes Ouverts et les Interfaces Correspondantes pour l'Electronique Automobile.

Vehicle Distributed eXecutive (Renault).

L'exécutif développé dans le cadre des applications embarquées pour le contrôle automobile (automotive) par le consortium européen (franco-allemand) depuis 1995 pour les configurations mono processeur avec des contraintes temps réel strictes. Pour une grande variété d'applications, pour les phases de mise au point aussi bien que pour la production (gestion des erreurs) indépendant du langage de programmation (syntaxe à la ANSI-C) et la portabilité et réutilisation du software.

III.15.1 - Organisation du consortium (2011)

La figure III.23 illustre la structure organisationnelle du consortium organisé en comité de direction, comité technique et groupe des utilisateurs.

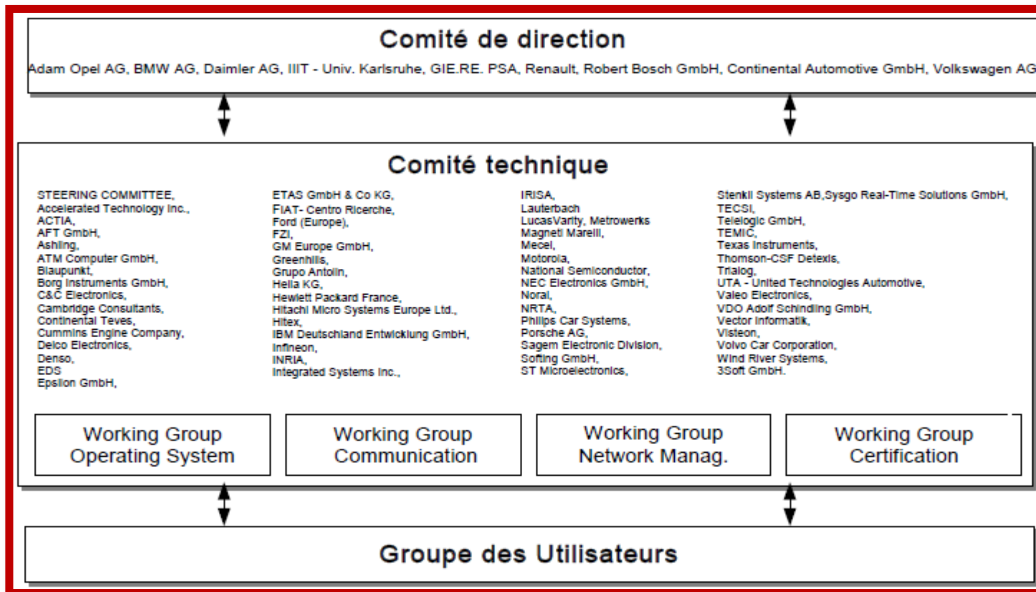


Figure III.23 – L'organigramme organisationnel du consortium

III.15.2 - Exigences spécifiques

Elles sont dues au contexte de développement des applications automobiles :

- L'**OS** est configuré et mis à l'échelle de façon statique (spécification statique du nombre de tâches, de ressources, de services),
- L'**OS** est capable de tourner à partir de mémoires **ROM**,
- L'**OS** fournit la portabilité des tâches applicatives,
- L'**OS** doit se comporter de façon prédictible et documentées et se conformer aux exigences temps réel des applications automotives,
- L'**OS** doit permettre l'implémentation de paramètres de performance prédictibles.

III.15.3 - Architecture générale

Elle est organisée autour des ECUs (Electronic Control Unit). Il y a trois entités comme sur le tableau suivant et la figure III.24 :

N°	Entités	Signification	Signification
1	OSEK-OS	Real time operating system	Environnement d'exécution des ECUs
2	OSEK-COM	Communication	Échange de données entre et inter ECUs,
3	OSEK-NM	Network management	Configuration, registration et monitoring des ECUs.

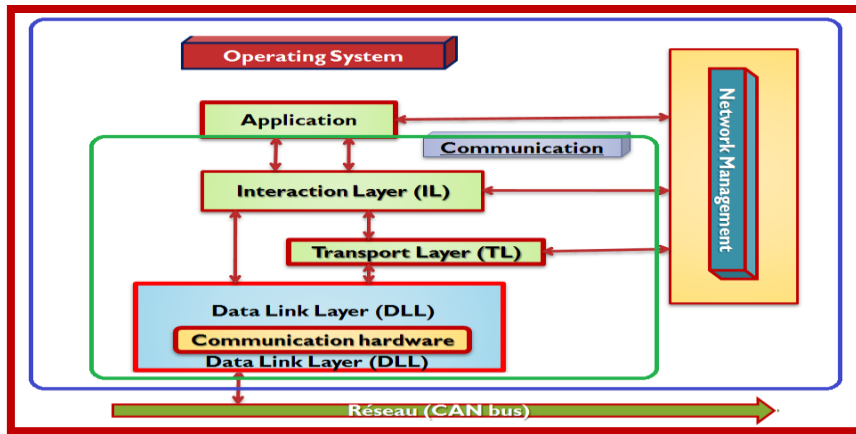


Figure III.24 – Architecture générale

OSEK fournit un environnement d'exécution pour les applications qui peuvent s'exécuter indépendamment les unes des autres. Deux interfaces utilisateurs fournies par OSEK pour gérer les entités qui veulent accéder concurremment à la CPU : les routines de gestion d'interruption (ISR) gérées par l'OS et les tâches basique sou étendues. Les trois niveaux d'exécution comme sur la figure III.25 :

- Interrupt,
- Niveau logique (fonctionnement de l'ordonnanceur),
- Niveau tâche.

Les trois niveaux vont en ordre décroissant de priorité. Les niveaux de priorité à l'intérieur d'un niveau d'exécution sont définis de façon statique.

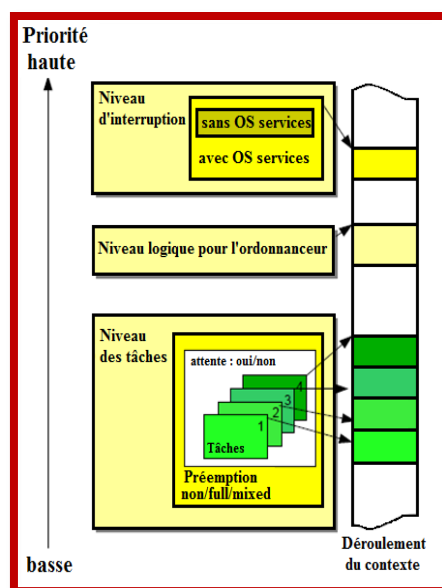


Figure III.25 – Les trois niveaux d'exécution

Les services du système d'exploitation gèrent les trois niveaux d'exécution, ainsi que :

1. L'administration des tâches,
2. L'administration des événements (synchronisation des tâches),
3. L'administration des ressources matérielles partagées,
4. Les compteurs et alarmes,
5. La communication entre tâches par messages,
6. Le traitement des erreurs.

III.15.4 - Les tâches

a - Définition

Une tâche est un programme qui s'exécute comme s'il était le seul à utiliser le CPU. Il possède une priorité, du code à effectuer, une partie de la mémoire et une zone de pile. Une tâche peut être dans différents états comme le montre la figure III.26 et le tableau suivants :

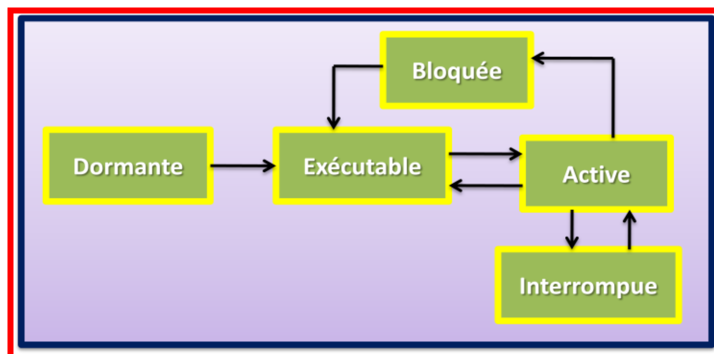


Figure III.26 – Les différents états possibles d'une tâche

N°	États de tâche	Définition
1	Dormante	La tâche est en mémoire mais n'est pas considéré par l'ordonnanceur,
2	Exécutable	La tâche est prête à être exécutée mais n'a pas le CPU,
3	Active	La tâche est exécutée par le CPU,
4	Bloquée	La tâche est en attente d'un signal ou d'une ressource pour poursuivre.
5	Interrompue	La tâche est Suspendue par une interruption

b - Passation d'un état à un autre

Le tableau suivant illustre les possibilités de passage d'un état à autre d'une tâche donnée.

Etat d'origine	Nouvel Etat	Evènement provoquant la transition
Dormante	Exécutable	La tâche est activée
Exécutable	Active	L'ordonnanceur donne le CPU à la tâche
Active	Bloquée	La tâche est en attente d'une ressource (bloquée par un sémaphore).
Bloquée	Exécutable	La ressource dont la tâche était en attente est libérée
Active	Exécutable	L'ordonnanceur donne le CPU à une autre tâche (préemption)
Active	Interrompue	Une interruption stoppe la tâche
Interrompue	Active	Le traitement de l'interruption est terminé

c - Les tâches d'OSEK/VDX

c.1 - Tâches basiques

La figure III.27 illustre les différents états et transitions des tâches basiques OSEK/VDX.

- Sans points bloquants,
- Rend le processeur :
  - À la terminaison,
  - Préemption par une tâche de priorité supérieure,
  - Sur occurrence d'un interrupt,
  - Trois états :
    - Running : la tâche à la CPU,
    - Ready : la tâche a été préemptée,
    - Suspended : la tâche est terminée et attend une requête de réactivation

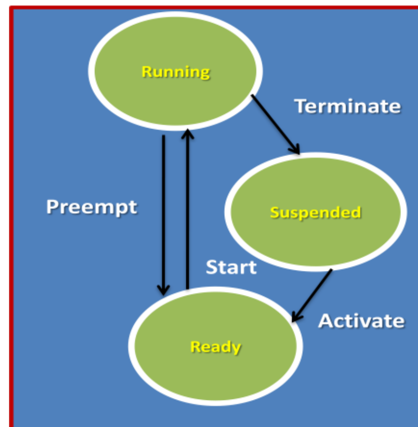


Figure III.27 – Différents états et transitions des tâches OSEK/VDX basiques

b - Tâches étendues

La figure III.28 illustre les différents états et transitions des tâches basiques OSEK/VDX. En plus des propriétés des tâches basiques, elles peuvent invoquer des services bloquants, qui possèdent un état supplémentaire : waiting, pendant lequel elles attendent l'allocation de la ressource bloquante**.**

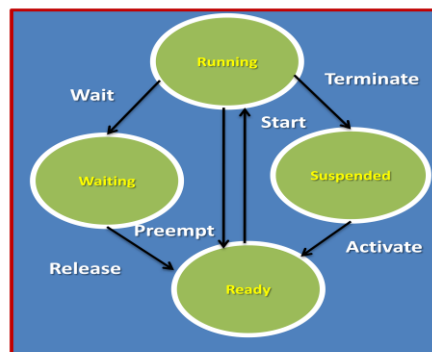


Figure III.28 – Différents états et transitions des tâches OSEK/VDX étendues

Les tâches basiques n'ont pas d'autres points de synchronisation que leur début et leur fin : des applications avec des points internes de synchronisation seront implémentées par plusieurs tâches basiques et peu gourmandes en termes de ressources système (**RAM**). Les tâches étendues peuvent se suspendre pour attendre une information intermédiaire. Elles peuvent implémenter une application cohérente en une seule tâche et elles demandent un peu plus de ressources système.

1. Activation :
   a. Deux primitives : ActivateTask et ChainTask.
2. Après l’activation, la tâche est prête dès la première instruction.

3. Suivant la classe de conformité, la requête d'activation est prise en compte ou non :
    a. Pas d'instances multiples d'une tâche,
    b. Les requêtes multiples sont éventuellement mises en queue pour être prises en compte à la terminaison.
4. Terminaison :
    a. Ne peut être provoqué que par la tâche elle-même,
    b. Deux primitives : TerminateTask et ChainTask.
5. Activation des tâches périodiques à l'aide de compteurs
6. Changement de contexte :
    a. Par appel à la primitive Schedule,
    b. Par décision de l'ordonnanceur.
7. Priorité :
    a. À priorité égale, FIFO par rapport à l'activation,
    b. Une tâche préemptée est première de la liste d'attente.

III.15.5 - Politique d'ordonnancement

Valeur des priorités statiques (sauf quand on utilise l'algorithme de priorité plafonnée). Les trois politiques : préemptif, non préemptif et mixte. L'ordonnanceur est considéré comme une ressource qui peut être réservée, empêchant ainsi temporairement la préemption (appel à la primitive GetRessource). L'appel à l'ordonnanceur est effectue dans les conditions classiques (terminaison, opération bloquante, activation d'une tâche de priorité supérieure, appel à ReleaseResource, retour d'interrupt) pas d'appel pendant la routine d'interrupt.

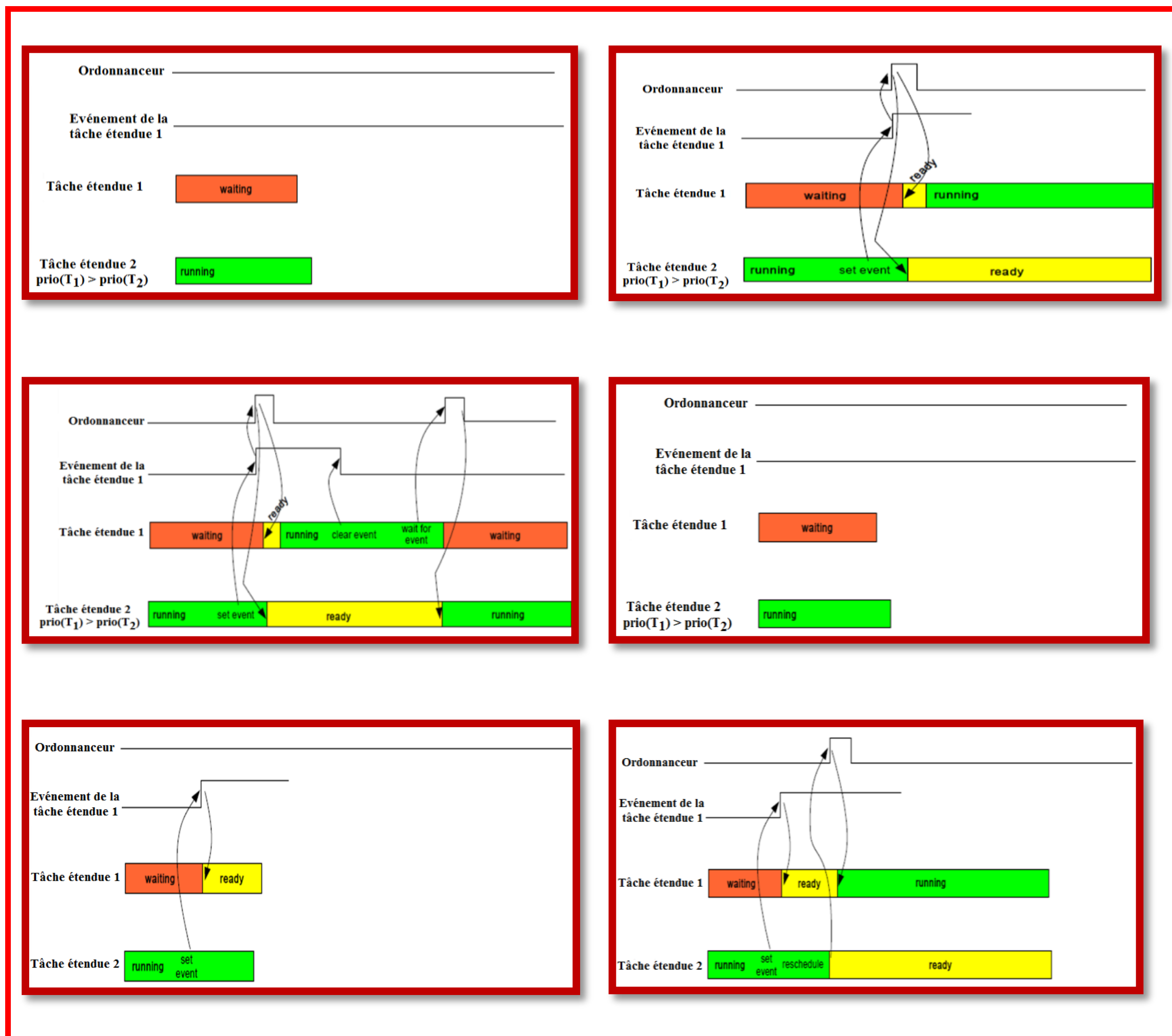
III.15.6 - Traitement des interruptions

Par une routine dédiée : ISR, généralement dans le contexte de la tâche interrompue, acquitte l'interruption sur le contrôleur et la transforme en action pour l'application (e.g. Event). Il existe deux modèles d'ISR :

- Type 1 : routine où l'on n'a pas besoin de faire un appel système ➔ seulement un retard
- Type 2 : déclarée comme telle ➔ appels systèmes pour l'entrée et la sortie de la routine d'interruption. pas d'appel à l'ordonnanceur pendant son exécution, pas d'appel bloquant.

III.15.7 - Synchronisation par événement

Elle Utilise le modèle synchrone, c'est-à-dire qu'un événement est généré et attendu de façon explicite. On a le type privé appartenant au consommateur, utilisable uniquement pour des tâches étendues et le critère pour la transition waiting ➔ ready quand un au moins des événements attendus est présent. Un événement est persistant, c'est-à-dire qu'une tâche dans l'état running demandant à attendre un événement déjà arrive va rester dans cet état. L'effacement d'un événement doit se faire de façon explicite (permet de simuler des événements fugaces). On va voir la synchronisation de tâches étendues préemptibles à travers l'exemple suivant de deux tâches T_1 et T_2 avec prio(T_1) > prio(T_2) comme sur la figure III.29. T_1 est bloquée dans l'attente d'un événement que va envoyer T_2.



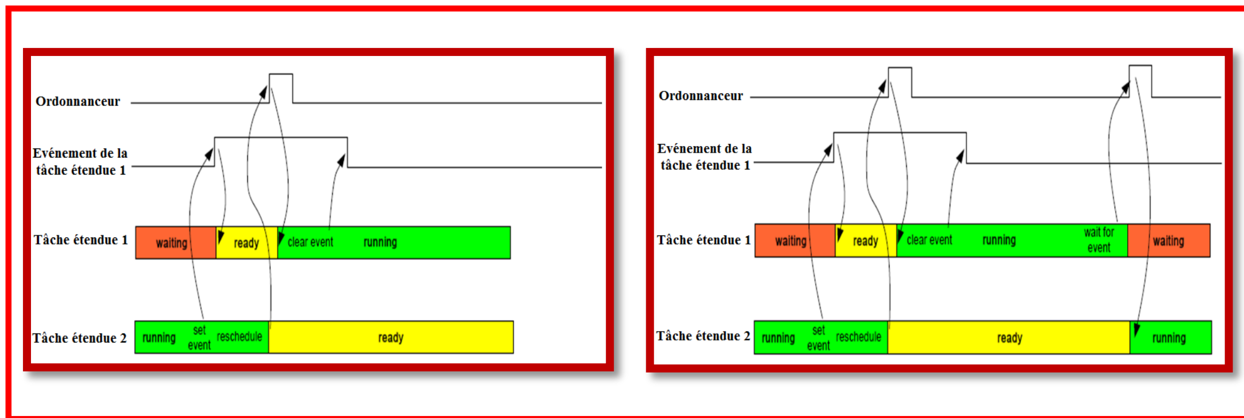


Figure III.29 – Les figures de synchronisation par événements

III.15.8 - Gestion des ressources

OSEK-VDX permet la pleine gestion des ressources critiques :

- Algorithme de la priorité plafonnée pour éviter les phénomènes d'inversion de priorité et l'inter blocage,
- L'algorithme de priorité plafonnée peut être optionnellement étendu aux accès à des ressources partagées par des ISR :
  - Pour assigner les priorités plafonds, des priorités virtuelles plus grandes que celles des tâches sont assignées aux interrupts,
  - Sinon l'OS n'exécutera une ISR que si toutes les ressources qui y seront demandées sont libres.
- l'OS interdit l'accès imbrique à une même ressource.
- L'ordonnanceur est une ressource.

III.15.9 - Alarmes et compteurs

Pour le traitement des événements récurrents, on a Timers et nombre de tours de l'arbre à cames. Enregistrés par un objet compteur qui compte les ticks et reasse à zéro quand il atteint la valeur maximale. Plusieurs alarmes peuvent être associées à un compteur. Une tâche peut être associée à l'alarme (de façon statique, à la génération du système). À l'occurrence de l'alarme, au choix :

- Activation de la tâche,
- Signalisation d'un événement de la tâche,
- Exécution d'une routine dans le contexte des systèmes TR.

III.15.10 – Communication

Par des messages dont la structure est définie à la génération du système :

- Messages d'événements :
    - Messages mis en queue dans une file de message. Mais il ne peut y avoir qu'une seule tâche en attente :
        - Transition d'état demandée, alarmes, etc.
- Messages d'état :
    - Message au **tableau noir**, non mis en queue mais contenu dans un buffer à une place, sur écrits quand un nouveau arrive :
        - Résultat d'un capteur, etc.
- Modèle de communication asynchrone (les appels en lecture ou écriture ne sont jamais bloquants).

Resynchronisation sur la fin de l'opération par :

- Polling,
- Activation d'une tâche en fin d'opération,
- Signalisation d'un événement en fin d'opération,
- Exécution d'une routine de callback,
- Occurrence d'une alarme si l'opération ne s'est pas terminée dans un délai de garde.

Type de communication défini de façon statique :

- 1 à 1 (mais on peut définir une liste à la génération et ne choisir le destinataire qu'à l'exécution),
- 1 à n, chaque destinataire recevant le message.

III.15.11 - Développements récents

De plus en plus de technologies assistées : Freinage et direction. Il y a beaucoup plus exigeantes en termes de déterminisme et de contraintes temporelles. Il y a émergence de réseaux sécuritaires (TTP/C, TTCAN, Flexay). OSEKtime : OS temps réel qui peut gérer, en plus des tâches classiques d'OSEK, des tâches Time Triggered (TT).