

→ Petite introduction théorique au temps réel !

Christophe Buffenoir

EN DEUX MOTS Nous entendons souvent parler du temps réel. Mais que cela signifie-t-il exactement ? Les concepts associés à cette technologie sont utilisés pour les logiciels embarqués dans les avions, les voitures, les trains, mais également les usines, les imprimantes, etc. Nous verrons au fil de ce dossier ce que cette notion apporte et comment la mettre en pratique.

Contrairement aux légendes que nous pouvons entendre autour de nous, le temps réel ne correspond pas à un logiciel qui répond le plus rapidement possible aux requêtes. En réalité, un logiciel temps réel répond selon des contraintes de temps déterminées.

Ainsi, pour répondre aux perturbations lors du pilotage d'un drone, les contraintes seront fortes et nécessiteront une certaine rapidité.

Au contraire, pour surveiller le niveau d'une cuve de 60 mètres cube avec une arrivée et une sortie d'eau ne pouvant dépasser 10 L/min, le logiciel de contrôle disposera d'un temps bien plus long.

Les exemples donnés ici montrent l'intérêt du temps réel : savoir réagir à un stimuli extérieur au système que nous appelons « STITR » (pour Système de Traitement de l'Information Temps Réel). Celui-ci peut être embarqué, c'est-à-dire incrusté dans un système comme par exemple une voiture ou un avion.

Le STITR permet l'automatisation d'un processus. Avant, les diverses opérations d'une usine étaient réalisées à la main, en tirant des cordes, en tournant des manivelles, etc. Aujourd'hui, un calculateur se charge d'actionner des éléments de contrôle (vannes automatiques, moteurs, pompes).

L'homme contrôle uniquement le bon fonctionnement et demande des opérations précises (réglage d'une consigne et arrêt du système entre autres). Non seulement, cette solution permet de réaliser des produits plus fiables et précis mais, en plus, elle limite les risques d'accident.

Les informations à traiter, quant à elles, ont une validité en fonction du temps. Elles ne

doivent surtout pas surgir trop tôt ou trop tard. C'est donc à ce niveau que nous parlons de contraintes temps réel. Nous pouvons les classer parmi trois grandes catégories. Tout d'abord les contraintes strictes correspondent à l'invalidité des informations à la fin de l'échéance.

Les contraintes critiques concernent les informations provoquant des troubles graves de fonctionnement – voire provoquer une mort humaine dans certains cas particuliers – si le délai n'est pas respecté. Cette dernière situation est évidemment à éviter à tout prix lors de la conception. Enfin, les contraintes dites « relâchées » gardent une certaine validité, moins importante, pour l'information après l'échéance.

La conception

Le développement de logiciel temps réel peut se réaliser selon trois approches différentes. La conception synchrone, la plus utilisée, consiste en l'écriture d'une simple boucle infinie. Les instructions sont exécutées les unes à la suite des autres. Aucun mécanisme bloquant ne doit être présent sous peine de geler le système.

L'approche multitâche préemptive sera détaillée plus bas. Dans ce cas, plutôt que de concevoir une seule boucle, l'application est décomposée en plusieurs boucles s'exécutant en parallèle. Un noyau temps réel est alors nécessaire pour gérer l'activation des différentes tâches. Enfin, l'approche objet est la plus récente. Elle reprend les principes de la POO.

Pour la conception d'un STITR multitâche préemptif, nous devons définir les différents agents actifs. Ces agents représentent différentes activités indépendantes pouvant communiquer entre elles. Chacun d'entre eux doit en plus répondre à des contraintes temporelles précises. En effet, après l'évènement déclencheur, l'agent doit réagir dans un temps minimal et surtout avant la limite de temps maximum. De même, nous pouvons déterminer un temps minimal avant lequel l'agent doit avoir fini son activité.

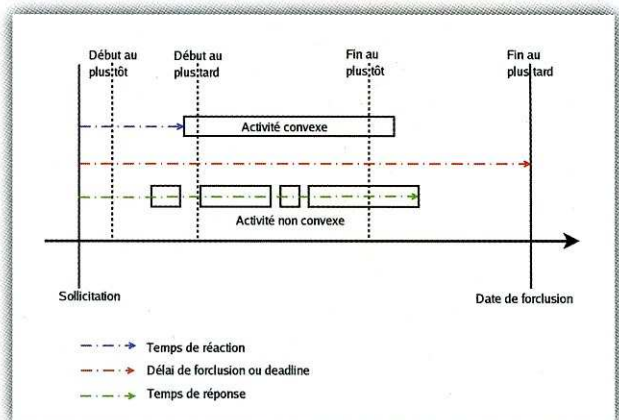


Fig. 1 : Les contraintes temporelles, la base du temps réel

Bien sûr, la principale contrainte reste la nécessité de terminer l'action avant la limite ultime, la date de forclusion. Toutes ces contraintes sont valables que l'activité soit convexe (elle se déroule alors sans interruption) ou non. La figure 1 nous montre plus clairement la succession de ces échéances.

Comme nous venons de le voir, une activité a des contraintes en fonction d'un événement déclencheur, d'une condition d'activation. Mais sous quelle forme se présente cet événement ?

Cela peut être tout simplement une communication avec une autre activité. Ou encore une interruption déclenchée matériellement, provoquée par l'arrivée d'un signal spécifique sur le processeur appelé « interruption ». Mais cet événement peut également être la fin d'un blocage volontaire.

Attention au vocabulaire ! Nous parlons bien ici d'activité et non pas de tâche. Une activité peut effectivement être une interruption. Dans ce cas, elle arrête l'exécution du programme en cours : elle est prioritaire.

Nous parlons alors de préemption. L'interruption permet par exemple d'informer une tâche qu'un capteur a changé d'état. La gestion d'une horloge est un exemple typique d'utilisation d'une interruption. Cette dernière accède alors à un compteur afin d'activer les tâches périodiques au bon moment.

La gestion des activités nécessite aussi des mécanismes de synchronisation. Par exemple, si un signal sonore doit être émis lors de la pression d'un bouton. Une première tâche gère le buzzer et une seconde l'attente de pression.

Une synchronisation entre les tâches permet d'éviter le retentissement de l'alarme au mauvais moment. La solution généralement employée est le sémaphore. Celui-ci est constitué d'un entier et d'une file FIFO (*First In, First Out*).

L'entier ne doit surtout pas être modifié en dehors des deux opérations prévues à cet effet. La première consiste à « prendre » le sémaphore, c'est-à-dire à se placer au bout de la file d'attente.

La seconde opération, « donner », signifie que la partie protégée du programme s'est achevée et libère la ressource. Le listing 1 présente l'algorithme générique d'un sémaphore.

Listing 1

```
Structure Semaphore
{
    Valeur : entier
    Attente : file
}

Fonction Donner(Sem : Semaphore)
{
    Si Sem.Valeur < 0
    Alors Débloquer_suivant (Sem.Attente)
    FinSi
```

```
Sem.Valeur = Sem.Valeur + 1
}

Fonction Prendre(Sem : Semaphore)
{
    Sem.Valeur = Sem.Valeur - 1
    Si Sem.Valeur < 0
    Alors Attendre(Sem.Attente)
    FinSi
}
```

La synchronisation peut se réaliser avec un sémaphore binaire. Elle ne concerne alors que deux tâches entre elles.

La valeur du sémaphore est initialisée à zéro. L'une des deux tâches sera bloquée tant que le sémaphore ne sera pas donné par l'autre tâche.

Une autre pratique concerne le partage des ressources. Un driver matériel, une variable commune, ou autre mécanisme ne pouvant

être utilisé par plusieurs tâches simultanément, sont des ressources non partageables.

Dans ce cas, soit l'opération d'accès est atomique (c'est-à-dire non divisible comme l'écriture d'un booléen) soit un sémaphore s'impose.

Ce dernier s'appelle alors « Mutex » ou « sémaphore d'exclusion mutuelle ». Sa valeur est initialisée à 1.

Avant d'accéder à une ressource, la tâche doit prendre le Mutex. Elle le donnera à la fin de l'opération. Cependant, le blocage doit être le plus court possible afin de pouvoir satisfaire les contraintes temporelles.

Non seulement les tâches se synchronisent entre elles mais, en plus, des moyens de communication inter-tâches sont disponibles. Si un tampon est nécessaire, une lecture et une écriture simultanées possible ou encore une synchronisation obligatoire, une file de messages est recommandée.

Cette dernière est également appelée « boîte aux lettres » pour son fonctionnement. Les messages sont envoyés par un ou plusieurs expéditeurs. L'unique destinataire récupère alors les « lettres » dans leur ordre d'envoi, c'est-à-dire en FIFO.

Par contre, si notre communication ne nécessite aucune des trois propriétés énumérées ci-dessus l'emploi d'un drapeau (une variable globale) protégé par un sémaphore binaire suffit amplement.

Souvent, le RTOS intègre directement le sémaphore dans les primitives du drapeau. Le rendez-vous, quant à lui, permet l'échange de données avec synchronisation mutuelle.

Le rôle du RTOS

Les différentes notions et mécanismes ci-dessus ne peuvent avoir lieu sans un noyau central orchestrant ce système.

Le RTOS (*Real-Time Operating System*) intervient alors. Ce dernier offre les différents services (comme les communications entre tâches et les mécanismes de synchronisation) et assure l'ordonnancement des tâches, l'exécution des interruptions, la gestion du temps, mais aussi les ressources système. Beaucoup de RTOS utilisent une HAL (*Hardware Abstraction Layer*), c'est-à-dire une couche basse permettant d'utiliser les fonctionnalités du RTOS sans se préoccuper des spécificités matérielles de la plateforme.

L'ordonnanceur (*scheduler* en anglais) est l'élément central du RTOS. Il sert à déterminer la tâche à exécuter.

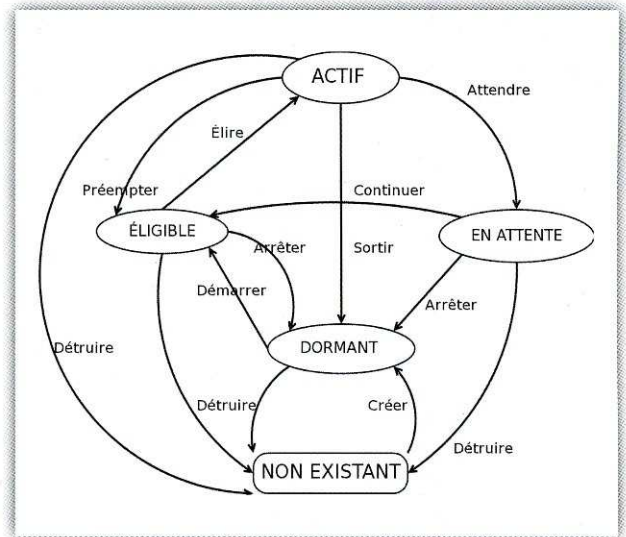


Fig. 2 : Les différents états possibles pour une tâche

En plus de cela, il attribue un état à chaque tâche : active, en attente, éligible à l'exécution ou encore inexistante.

Certains états sont spécifiques au RTOS utilisé, mais le principe reste le même. Tout d'abord une tâche doit être créée. Elle est alors dormante. Elle sera démarrée par une autre portion de code. Elle atteint alors l'état d'éligibilité et elle est prête pour s'exécuter.

Cependant, le processeur est une ressource non partageable gérée par l'ordonnanceur. La tâche devra donc attendre que la place se libère.

À la fin de son exécution, elle retournera soit à l'état d'éligibilité, soit à celui d'attente : la tâche a terminé son traitement pour atteindre une instruction bloquante.

Seulement, l'ordonnanceur choisit une tâche éligible selon des règles strictes.

Des stratégies différentes existent afin de satisfaire les spécificités des systèmes conçus. Certaines d'entre elles se basent sur des critères temporels, comme la stratégie EDF (*Earliest Deadline First*) activant la tâche dont le délai de forclusion se termine le plus tôt. La LLF (*Least Laxity First*) active la tâche ayant la plus petite laxité (c'est-à-dire au plus petit temps de marge entre la fin de la tâche et la fin du délai de forclusion).

Mais en général, les stratégies basées sur des priorités sont préférées pour leur déterminisme. La priorité peut être fixe ou changée dynamiquement au cours de l'exécution selon un mécanisme simple (aléatoire ou cyclique) ou basé sur une autre stratégie d'ordonnancement.

Une panoplie d'outils est ainsi disponible pour obtenir le comportement souhaité par le système. Le *Rate Monotonic* est la stratégie à priorités fixes la plus utilisée. Les priorités sont attribuées à la conception. Elles sont généralement inversement proportionnelles à la période d'activation.

Il subsiste cependant un problème non négligeable. Prenons le cas d'une ressource non partageable. Une tâche de très

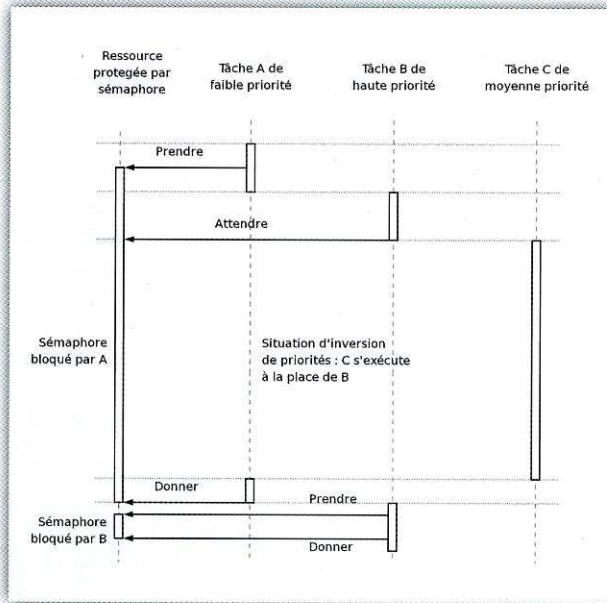


Fig. 3 : Une inversion de priorité dangereuse

basse priorité peut la bloquer alors qu'une autre, de priorité plus forte, tente d'y accéder à cette même ressource. Nous assistons alors à un blocage bien trop long de l'application, c'est-à-dire le temps que la tâche de basse priorité finisse le traitement. Elle sera régulièrement interrompue. Ce problème est connu sous le nom d'inversion de priorité : une tâche prioritaire est soumise à l'exécution d'une autre moins prioritaire. Deux solutions courantes sont proposées. La première, appelée « forçage de priorité », attribue la plus grande priorité existante à la tâche bloquante. La seconde, « héritage de priorité », attribue la plus forte des priorités des tâches bloquées à la tâche bloquante. Au lieu d'écrire soi-même l'algorithme du sémaphore, il est fortement conseillé d'utiliser les services fournis par l'éditeur du RTOS. En effet, ce genre de subtilités est implanté d'origine.

De même, une API générique et indépendante de la plate-forme permet d'utiliser facilement des périphériques matériels comme les ports séries et parallèles, l'écran ou tout autre grâce aux pilotes. Le plus simple et le plus sûr reste de bien étudier la documentation. Se pose alors la question fatidique : quel RTOS utiliser pour mon application ? Il n'existe pas de réponse universelle mais, dans tous les cas, des renseignements doivent être pris afin de respecter les normes.

Des réglementations existent et influent fortement sur le choix du système. Particulièrement pour l'aéronautique, la norme DO-178B doit être respectée à la lettre. Pour l'automobile, il s'agit de l'OSEK. Le développement temps réel peut nécessiter des contraintes très fortes en fonction de la criticité de l'application. Pour des besoins plus généraux, nous trouvons de nombreux systèmes. Le plus connu s'appelle VxWorks, édité par la société Wind River (cette entreprise est également citée pour ses actions sur les systèmes BSD, notamment FreeBSD).

Malheureusement, ce RTOS de qualité est également réputé pour son prix. Les dérivés de Linux, comme RTAI, LynxWorks et d'autres, sont l'alternative à la mode. Cependant, pour ces derniers, un véritable noyau temps réel tourne en avant-plan et exécute Linux en tâche de très basse priorité. Ce principe exige un développement vertical contraignant. Deux autres RTOS, libres, peuvent attirer l'attention : RTEMS et eCos (présenté dans l'article ci-après).

Christophe Buffenoir,

<http://www.buffenoir.org>,