Introduction à l'Embarqué, Temps réel & Robotique

Cours 2 : Quelques notions sur les systèmes embarqués & Temps Réels

Alexander Saidi
ECL - LIRIS

Janvier 2017

Plan de la suite

- Une introduction aux systèmes Embarqués (embedded).
- Objectifs et missions essentielles d'un S.E.
- Noyau (Kernel) : la partie essentiel du SE

  ☞ Tous les "embarqués" n'ont pas besoin d'un SE
- Une introduction aux systèmes Temps réels (SETR = *RTOS*).
- Système Temps Réel : encore plus *sensible* au noyau
- **Ce par quoi le Tempe Réel (TR) embarqué** *non trivial* **est concerné**
- Robotique : programmation Robot & stratégies : un exemple

L'Embarqué

Exemple de systèmes embarqués

L'Embarqué (suite)

Définition (tentative) :

Embedded system = Système Informatique conçu pour effectuer une ou plusieurs fonction dédiées.

• Par exemple, un système de contrôle de train qui est conçu spécifiquement dans ce but et ne peut pas être utilisé pour autre chose, pa rexemple faire de la publication.

• Caractéristiques (et contraintes fortes) :
  ◦ CPU pas forcément puissant
  ◦ Peu de mémoire
  ◦ Peu ou pas d'E/S
  ◦ Eventuellement des contraintes Temps Réelles
  ◦ Ressources limitées (car cout, taille, besoin d'énergie, ...)

Systèmes Embarqués (TR ou Non)

Figure 1 – Exemples de systèmes embarqués (Joseph Sifakis Laboratoire Verimag)

- L'informatique est devenue **diffuse** (*ubiquitous computing*).
- Les dispositifs automatiques **embarqués** remplacent de plus en plus les processus manuels.

Systèmes Embarqués (TR ou Non) (suite)

- Nous sommes envahis par les **Systèmes Embarqués** :
  - Smartphone, GPS, ...
  - Voiture : *vous n'avez pas attaché votre ceinture; ce n'est pas bien !*,
  - Téléphone portable, aspirateur automatique, pilote automatique...

Définition-1

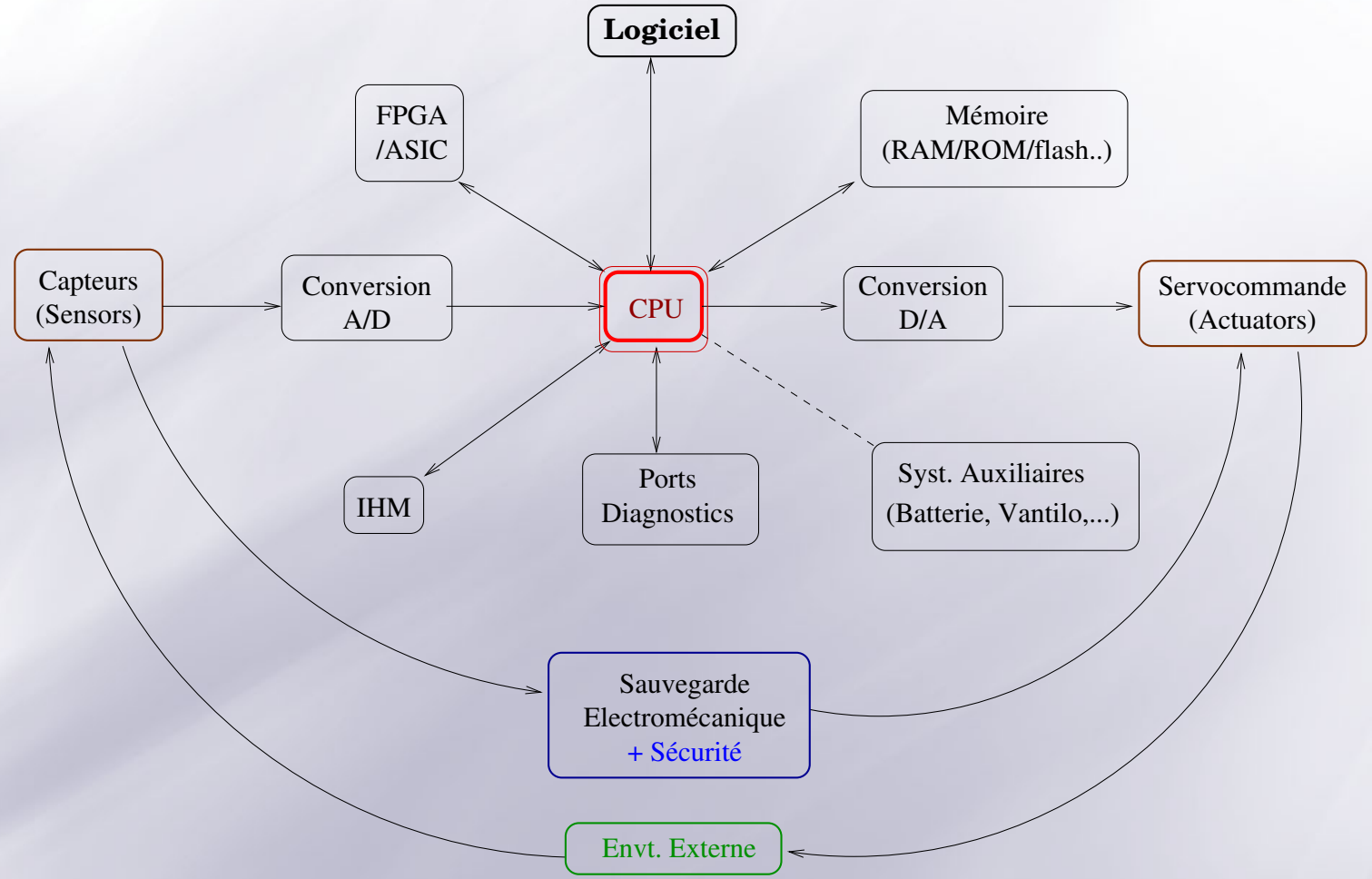
Un système embarqué est un système mécatronique (avec la partie mécanique éventuellement réduite) autonome dédié à une tâche précise.

- Le logiciel (soft) et le matériel sont intimement liés, l'un noyé dans l'autre (vs. PC)

- En général, un tel système ne possède pas d'entrée-sortie "std" (e.g. un vrai clavier/écran comme un ordinateur).

☞ Le système peut être TR ou non (+contraintes temporelles).

Un système embarqué (et/ou TR) typique



Éléments de Conception d'Embarqué

- Le concepteur est **Pluridisciplinaire** : info, électronique, réseaux, sécu.
  - ➛ En plus, il faut savoir optimiser les coûts ....!
- **Un "bon" système embarqué** conçu doit être :
  - ➛ Robuste et Simple (souvent gage de robustesse).
  - ➛ Fiable et fonctionnel,
  - ➛ Sûr (en particulier si des vies en jeu)
  - ➛ Tolérant aux fautes
  - ➛ Autres : encombrement, poids, consommation électrique, coût, temps de développement (concurrence !)
- On aura tendance à sur-dimensionner (si méconnaissance des évènements/it).

Exemples de systèmes embarqués

Office systems and mobile equipment	Building systems	Manufacturing and Process Control
Answering machines	**Air conditioning**	**Automated factories**
Copiers	**Backup lighting and generators**	**Bottling plants**
Faxes	**Building management systems**	**Energy control systems**
Laptops and notebooks	**CTV systems**	**Manufacturing plants**
Mobile Telephones	**Fire Control systems**	**Nuclear power stations**
PDAs, Personal organisers	**Heating and ventilating systems**	**Oil refineries and related storage facilities**
Still and video cameras	**Lifts, elevators, escalators**	**Power grid systems**
Telephone systems	**Lighting systems**	**Power stations**
Time recording systems	**Security systems**	**Robots**
Printer	**Security cameras**	**Switching systems**
Microwave	**Sprinkler systems**	**Water and sewage systems**

Figure 2 – [Thanks to ENSEIRB, Patrick Kadionik-2005]

Exemples de systèmes embarqués (suite)

Transport	Communications	Other equipment
Aeroplanes	**Telephone systems**	**Automated teller systems**
Trains	**Cable systems**	**Credit card systems**
Buses	**Telephone switches**	**Medical Imaging equipment**
Marine craft	**Satellites**	**Domestic Central Heating control**
Jetties	**Global Positioning System**	**VCRs**
Automobiles		
Air Traffic Control		
Signalling Systems		
Radar Systems		
Traffic Lights		
Ticketing machines		
Speed cameras, Radar speed detectors		

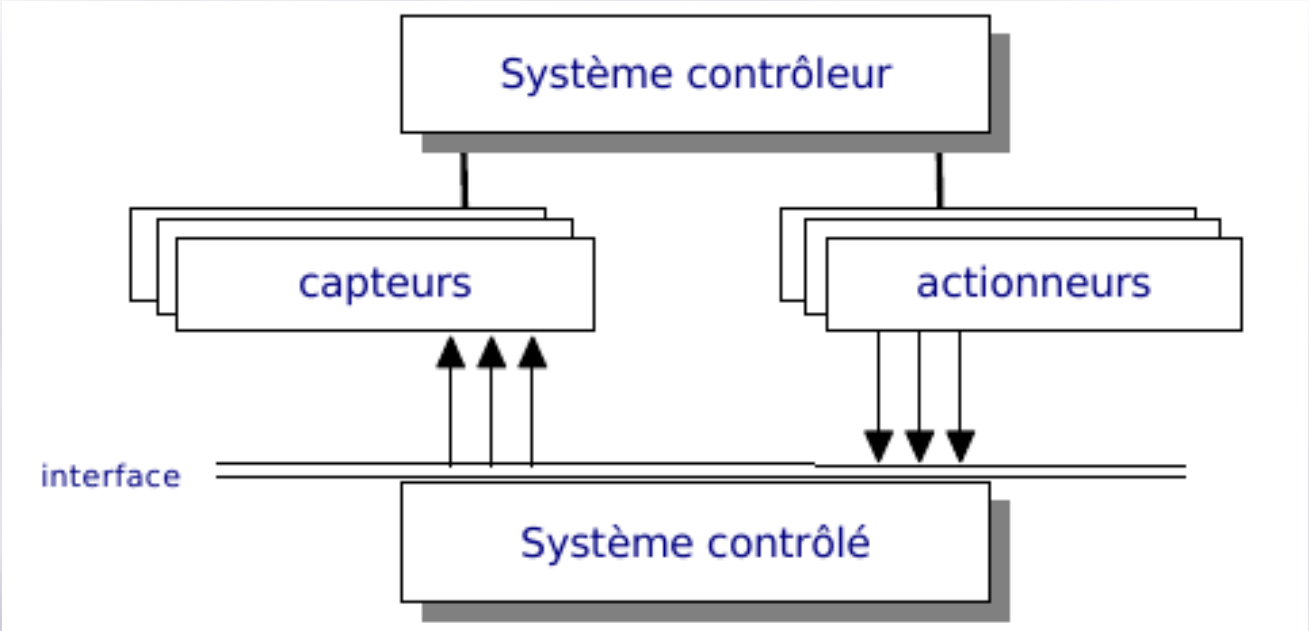
Figure 3 – ... suite exemples SE embarqués (par domaine)

Systèmes Embarqués (et TR) Libres

- Linux (e.g. ROS/Debian) de plus en plus utilisé dans l'embarqué :
  - Libre et disponible, stable et efficace, pas de *royalties*, ouvert, différentes distributions disponibles, communauté rapide et réactive (en cas de besoin d'aide), connectivité IP std...
    - Version *patchée* pour RT (XINOMAI)
  - Porté sur toute familles : x86, PPC, ARM, MIPS, 68K, ColdFire...
  - Beaucoup d'extensions (pour passer de Linux à Linux embarqué).
  - Taille noyau modeste (e.g. . 800 k pour *uClinux* sur un Coldfire).
  - Distributions différentes pour diverses applications : PDA, routeur, téléphone, ...
  - Optimisation avec gestion dynamique de modules
  - S'accommode de l'absence de MMU (ex. uClinux)
- ☞ Attention MMU : gain de performances mais code à surveiller.

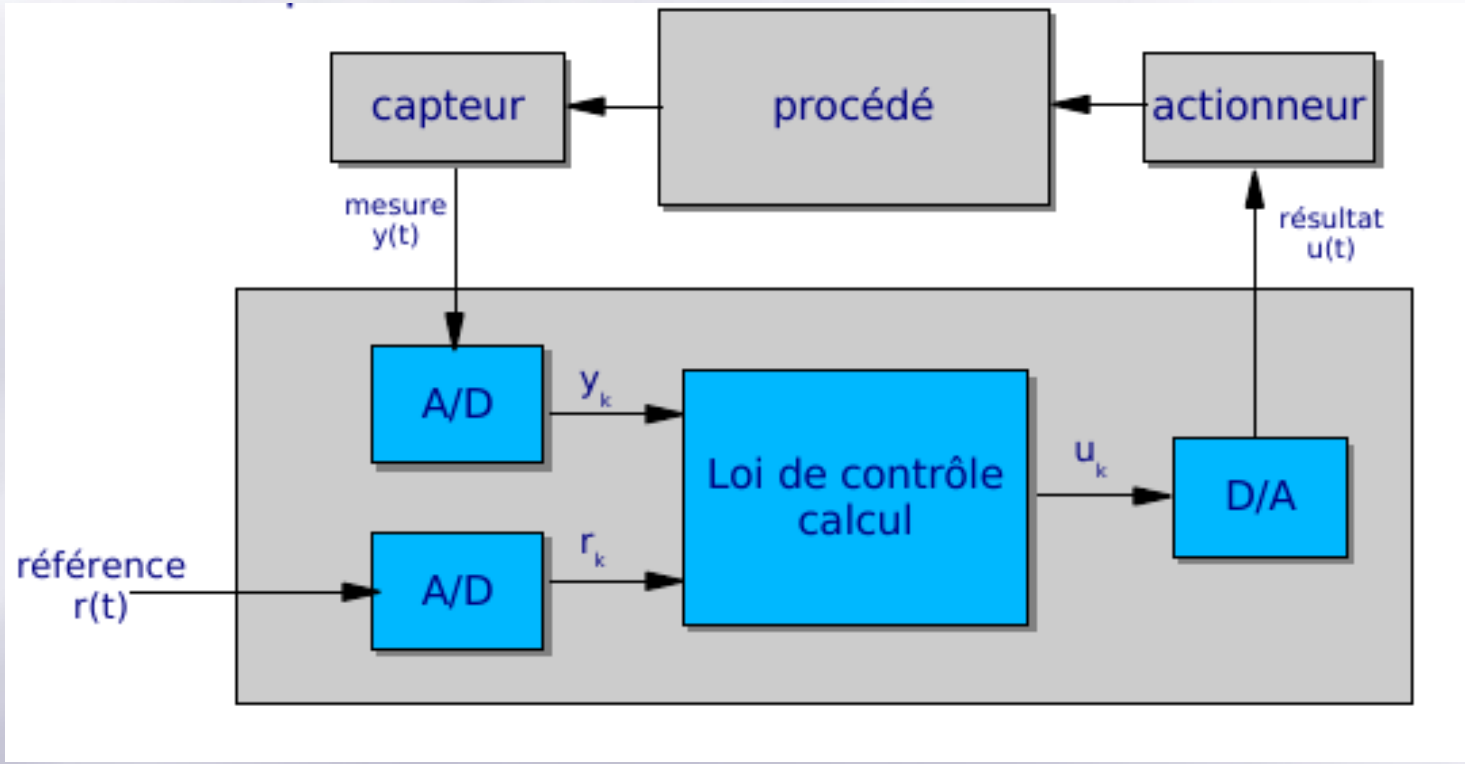
Rappel schéma OS (TR) embarqué

- Rappel schéma générale et synthétique d'un OS embarqué :



Rappel schéma OS (TR) embarqué (suite)

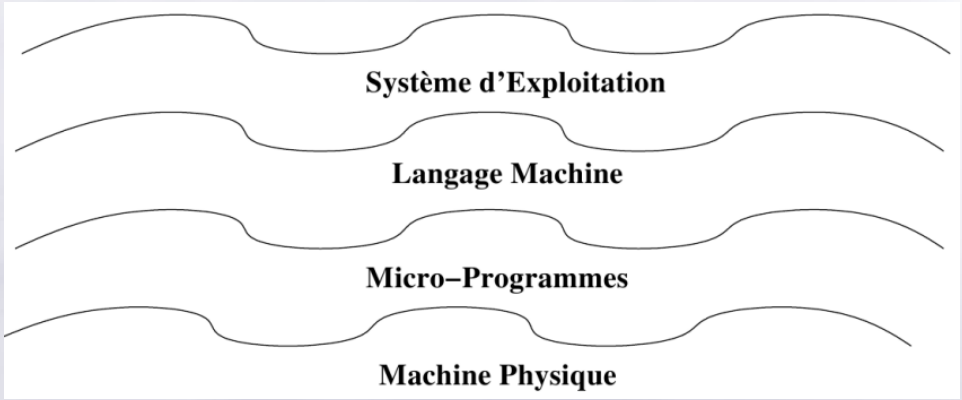
Exemple de Régulation ou **asservissement** simple par contrôle numérique



- Ce schéma général donne lieu au paradigme **S P A** (voir +loin). ../..

L'essentiel d'un S.E.

Une vue fonctionnelle et par couche :



• Un **RTOS** (*real time operating system*) vient remplacer la couche "Système d'exploitation" (qui est quelconque).

◦ Peut être directement lié à la machine ϕ.

☞ Sur un PIC, la couche "Système d'Exploitation" est (par défaut) absente.

◦ PIC : Programmable Intelligent (interruption) Controler

L'essentiel d'un S.E. (suite)

- Le découpage en couches n'est pas figé (sauf les plus basses)
- Suivants les contraintes, certaines fonctions sont confiées aux logiciels ou aux matériels.

- Normalement, toute fonction *logicielle* peut être confiée au *matériel* et vice versa.
  - Le choix est d'ordre économique ou d'efficacité (technique).
  - De nos jours, avec l'extension de l'usage des processeur *esclaves*, les fonctions d'un SE sont plutôt exécutées par le matériel.

> L'approche **fonctionnelle** d'un SE permet une vue indépendante de son implantation.

- Un SE ◦ Noyau

../..

Noyau d'un SE

- Le noyau d'un SE est lui-même un **logiciel**
  = La partie centrale la plus **critique** : impose des **performances élevées**.
  - Sa conception / programmation est particulièrement délicate.
- Le noyau (d'un SE) = un logiciel qui assure :
  - la communication logiciel-matériel (échanges d'infos)
  - la gestion de l'exécution des divers logiciels (tâches) d'une machine
    - lancement de tâches diverses, divers ordonnancement, ...
  - la gestion des ressources matérielles (mémoire, processeur, périphes...).
  - la sécurité de fonctionnement, etc.
- fournit des mécanismes d'abstraction du matériel
  - indépendance (du code) vs. matériels

Noyau d'un SE (suite)

- La frontière entre les fonctionnalités d'un SE (ci-dessus) est vague :
  - La conception du noyau est un compromis entre performance, sécurité et architecture des processeurs.

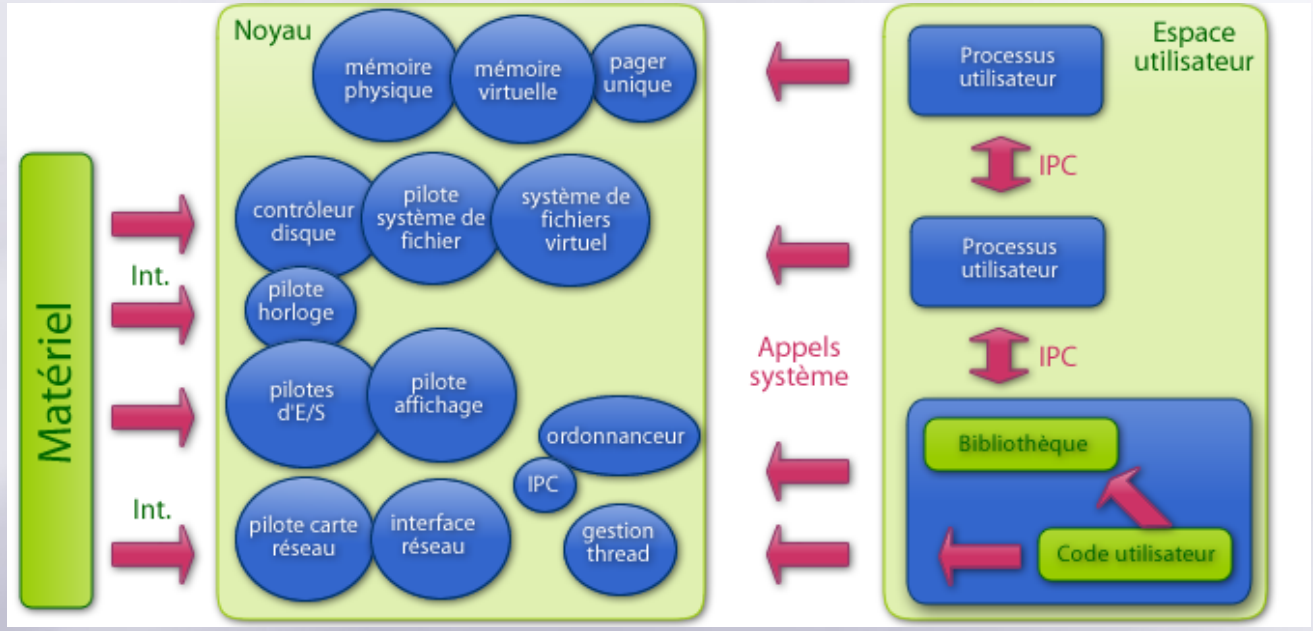


Figure 4 – Noyau (monolithique)et ses *services*

Noyau d'un SE (suite)

- Le noyau propose ses fonctionnalités (via des appels système **SVC**).
- Transmet ou interprète les informations du matériel via des **interruptions**
  - cf. les **entrées et sorties**.
- Le noyau est **prioritaire** sur l'utilisation des ressources matérielles, en particulier de la mémoire (voire, du processeur).
- Un noyau impose un *partitionnement virtuel* de la RAM physique en deux régions disjointes :

  l'espace noyau (réservé et protégé) et *l'espace utilisateur*.

- Le noyau structure également le travail des développeurs :
  - le développement de code dans l'espace noyau est plus délicat (vs. l'espace utilisateur) car la mémoire n'y est pas **protégée**.
- C'est le cas des OS actuels : Linux, Windows, Mac OS X, etc.

Noyau et ses services

- L'allègement du code noyau diminue la taille du noyau (cf. micro noyau).
- Certains SEs peuvent conserver les pilotes pour le matériel en leur sein.
  - Cette implantation à l'intérieur du noyau est faite dans **l'unique but** d'augmenter les performances.
  - Mais pose le problème de spécialisation matérielle vs. performances.
  - Pour cette raison, on préfère les solutions modulaires / logicielles
- D'autres peuvent implanter les fonctionnalités suivantes dans l'espace utilisateur plutôt que dans le noyau lui-même.
  - les pilotes matériels,
  - les fonctions réseaux
  - systèmes de fichiers ou les services : impression, sauvegarde, etc.

☞ Le rôle premier d'un noyau est de gérer les **processus** (voir +loin)

Noyaux monolithiques non modulaires

- Monolithique : **l'ensemble des fonctions du système et des pilotes** sont **regroupés** dans un seul bloc de code binaire.
  - P. Ex. : d'**anciennes** versions de Linux, certains **BSD** ou certains vieux Unix.
- **Pros** : Simplicité du concept, excellente vitesse d'exécution
  - ➛ les noyaux monolithiques ont été les premiers développés.
- **Cons** : Mais très difficile à faire évoluer/maintenir .

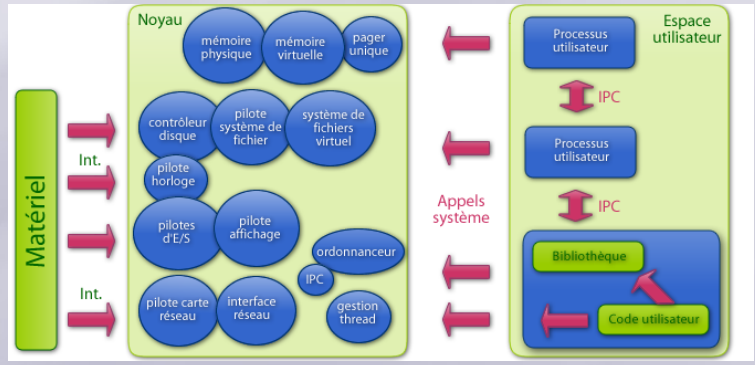


Figure 5 – Architecture Noyau monolithique

Noyaux monolithiques modulaires

- Réponse aux problèmes des monolithiques ⇒ monolithiques **modulaires**.
  - Seules les parties fondamentales du système (dit **Micro noyau**) sont regroupées dans un bloc de code unique,
  - Les autres fonctions (e.g. les pilotes mat.) sont regroupées en modules
  - Séparés du code et du binaire du noyau.
- Utilisé dans un grand nbr de SE : Linux, des BSD ou Solaris (actuels).
  - + **Chargement/déchargement à chaud** de modules.
- Le chargement à chaud des modules (pilotes) dans Linux :
  - ➛ augmente les possibilités de configuration, et soulage le système,
  - ➛ des syst. de fichiers peuvent être chargés de manière indépendante,
  - ➛ un pilote de périphérique peut être chargé/déchargé, etc.
  - Mais il faut gérer les dépendances.

Compromis : Micro-noyau enrichi

- Cherche à minimiser les fonctionnalités dépendantes du noyau
  - Place la plus grande partie des services du SE à l'extérieur de ce noyau (sous forme de **serveurs** indép., dans l'espace utilisateur).
  - Un petit nbr de fonctions fondamentales dans l'espace "micro-noyau".
- **A noter** : *serveur* vs. *module*.

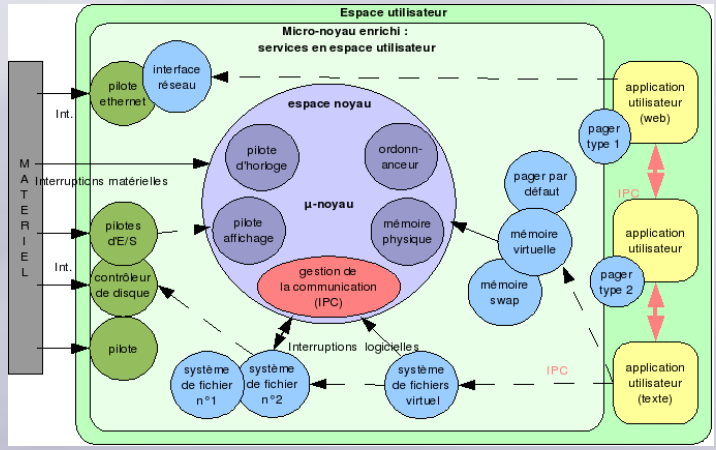


Figure 6 – (Micro) Noyau modulaire et ses services

Noyau : aperçu d'Ordonnanceur UC

- Fonction fondamentale d'un noyau (ordonnanceur de l'UC $\approx$ *scheduler*).
- Notion très liée aux systèmes multitâches
- Son rôle : faire exécuter les tâches par **commutation** de contextes.
- Selon la stratégie adaptée, l'algorithme d'ordonnancement peut tenir compte des priorités (1 ou N processeurs)
    - Dans le but d'utiliser efficacement les ressources de la machine.

Cas multiprocesseurs :

- La plupart des ordonnanceurs modernes permet d'indiquer sur quel processeur les tâches peuvent-elles être exécutées.
- Certains permettent également de migrer des tâches sur d'autres machines d'une **grappe** de calcul.

☞ *Ordonnanceur* est une notion générale (de l'UC, de disque, de périph.,...)

Noyau : aperçu d'Ordonnanceur UC (suite)

Types d'ordonnancements (multi-tâches) :

- coopératif :

  les tâches doivent être écrites de manière à coopérer les unes avec les autres et ainsi accepter leur suspension pour l'exécution d'une autre tâche.

- préemptif :

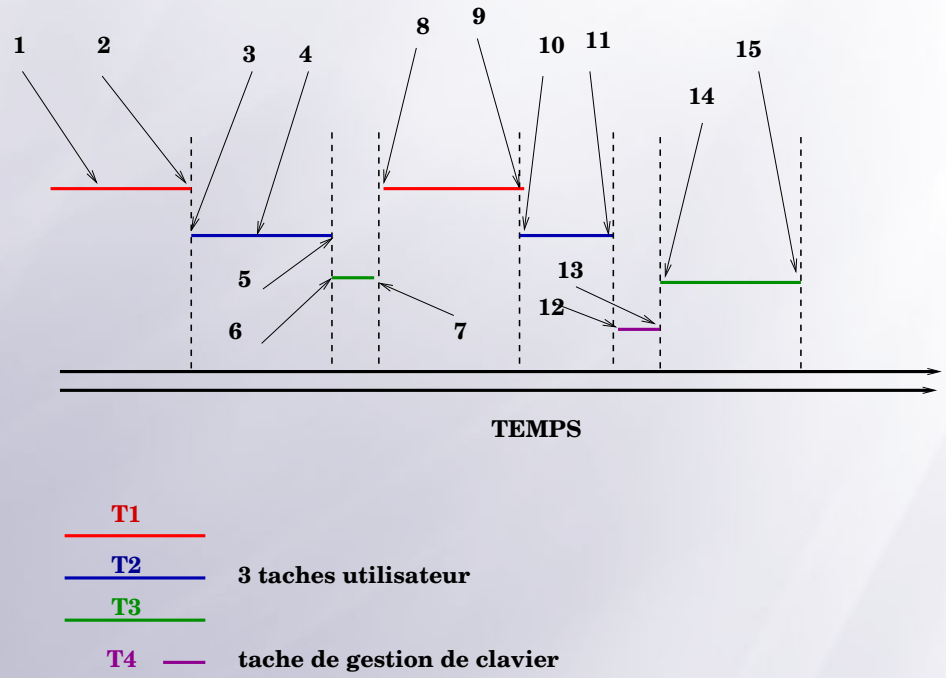
  l'ordonnanceur a la responsabilité de l'interruption des tâches et du choix de la prochaine à exécuter.

- Certains noyaux sont eux-mêmes préemptifs : l'ordonnanceur peut interrompre le noyau lui-même pour faire place à une activité (typiquement, toujours dans le noyau) de priorité plus élevée.

- Notion de latence, problème de masquage des interruptions, ...

Noyau : exemple de déroulement

1 : T1 s'exécute
2 : Fin quantum de T1
3 : Début quantum T2
4 : T2 block une ressource
(mais n'est pas suspendue)
5 : Fin quantum de T2
6 : Début T3
7 : T3 passe la main (suspendue)
8 : T1 reprendre
9 : Fin quantum de T1
10 : Début quantum T2
11 : Suspension T2 (Préemption)
(intervention opérateur)
12 : Début tâche opérateur (prioritaire)
13 : Fin opérateur
14 : Début quantum T3
15 : Fin quantum de T3
...



Noyau : Gestionnaire de mémoire

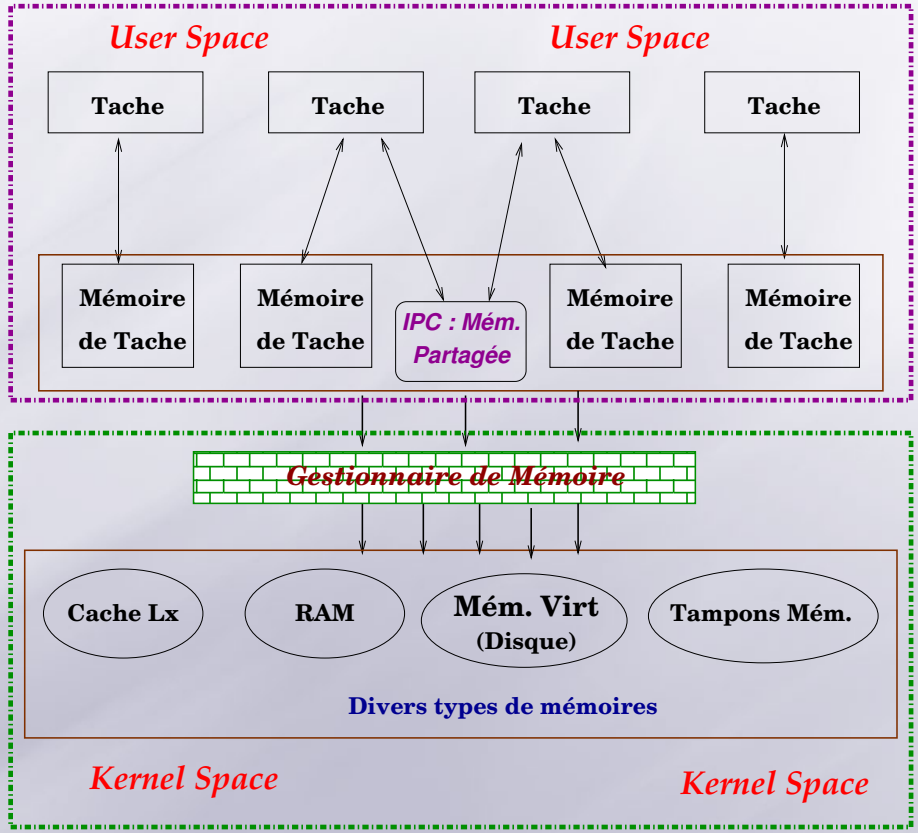


Figure 7 – espaces et le MMU

Noyau : Gestionnaire de mémoire (suite)

- Se charge d'allouer de la mémoire à des processus lorsqu'ils en ont besoin.
- Cette mémoire sera par défaut propre au processus demandeur
- **Mémoire Virtuelle** : abstraction (disque, RAM, Réseau), pagination
- Propriétés de la mémoire manipulée par tout processus :
  - est privée (protégée),
  - peut être étendue jusqu'aux capacités théoriques de la machine ;
  - un processus ne peut pas accéder à la mémoire d'un autre processus
    - (sauf allocations et autorisations spécifiques).
- **MMU** : gère l'allocation demandée par le noyau.
  - Prévient le noyau des violation
  - Le noyau décide d'arrêter / tuer le processus fautif.

Noyau : Appels systèmes

- Les appels systèmes sont des fonctions :
  - appelées depuis un programme de l'espace utilisateur ;
  - dont l'exécution (le traitement) est effectué dans l'espace noyau ;
  - dont le retour est effectué vers l'appelant dans l'espace utilisateur.
- En plus d'un changement de mode d'exécution (passage en mode SV), un appel système provoque au moins deux commutations de contextes :
  1. Contexte du processus appelant SVC;
  2. Contexte du noyau ;
- A la fin de l'exécution du service, le noyau "donne la main" au *sheduler* .
  - Le *sheduler* est aussi une fonction du noyau ! (peut être *cablé*)
  - Il choisit un processus de **reprise** (pas forcément l'appelant du SVC).
  - Voir diagramme d'états des tâches.

Noyau : Appels systèmes (suite)

• Un SVC entraine de nombreuses interactions de bas niveau (dans l'espace noyau), pour les E/S (communication avec **pilotes**)

• Le coût d'un SVC est nettement plus élevé qu'un simple appel de fonction :

- un appel simple de fonction = quelques instructions primitives (chargement et exécution en une zone mémoire),
- un appel système (SVC) a un cout en milliers ou dizaines de milliers d'instructions primitives (dont les sauvegardes de contextes),

• En général, les fonctions qui sont utilisées de manière **intense** sont déplacées dans l'espace noyau.

• Les programmes utilisateurs devraient faire un nombre restreint d'appels système de haut niveau (Ex. *read*).

Noyau : Gestion du matériel

- Se fait par l'intermédiaire de **pilotes** de périphériques.
- Un pilote= un petit logiciel léger dédié à un matériel donné
  - permet de communiquer avec ce matériel.
- Les pilotes sont généralement inclus dans l'espace noyau et communiquent avec l'espace utilisateur via SVC ,
  - il faut commuter les contextes : on a intérêt à placer les pilotes dans le noyau (question de fluidité/latence).
- Certains périphériques lents (appareils photographiques numériques, outils sur liaison série, etc.) sont/peuvent être pilotés depuis l'espace utilisateur, le noyau intervenant au minimum.

☞ Il y a toujours eu des déplacement des fonctions dans ces 2 espaces.
  - Cela dépend de ce qu'on veut (Perf., Taille, ...) !

Noyau & Gestion du matériel : exemple d'E/S

- Notion de base : *échange de signaux.*
  - Ex. imprimante : *prête, envoi données, fin, erreur out of paper, spool plein/vide, bourrage...*
- Le SE gère les communications entre les périphériques (malgré les standard différents).
  - Le SE passe par un Pilote (*Device Driver*, un par matériel)
- Le SE peut charger un module d'impression (+ un démon : e.g. *cupsd*)
- Le service impression est assuré par un logiciel :
  - Les vieux SE (DOS/CPM) s'en chargeaient eux même (zone SE)
- Le système d'ES : interface entre l'utilisateur et les périphériques (par commandes de haut niveau telles que *read, write, ...*).
  - Ces commandes provoquent un SVC (superviser call).
  - On peut sous traiter par un processeur *esclave*.

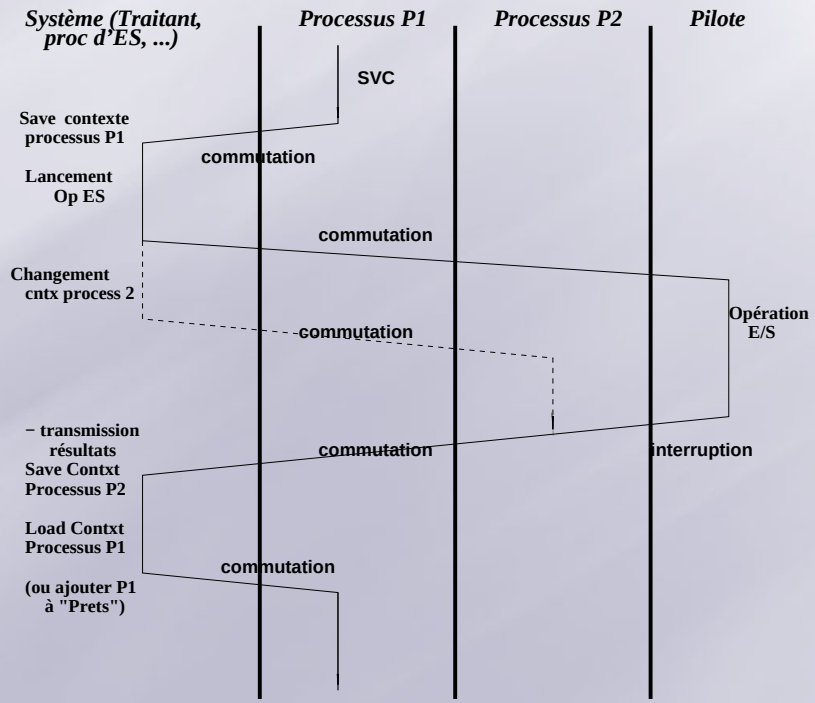
Noyau & Gestion du matériel : exemple d'E/S (suite)

Exemple : détails d'une demande d'impression :

- Le SE sous-traite à l'un de ses utilitaires (e.g. CUPS, LPR, ...)
  - ➛ Processus ou Démon ! (Le demandeur peut être mis en attente)
  - ➛ Paramètre du lancement : Printer + le fic. à imprimer
- LPR prépare une suite de *mots* (selon le type de la liaison)
  - ○ Conversion éventuelle selon le langage de l'imprimante
- Le Pilote interroge l'état de l'imprimante (par réseau ou par câble)
  - ○ Le pilote (driver) connaît les détails matériels
  - ○ États : *prêt, libre, occupé, papier, cartouche, bourrage, fin,* ...
  - ○ Le processus (e.g. LPR) n'a pas besoin de connaître les détails matériels, il s'adresse au Pilote.
  - ○ Mais le Pilote et LPR exécutent le même protocole.
- Si tout va bien, commence un série de communications :
  - ○ Envoi d'un mot (ou 1 *bit* si liaison série, un *paquet* si réseau, ...)
  - ○ Attente de AR (ou un code d'erreur); peut être asynchrone
  - ○ Refaire
- A la fin (ou en cas de pb) : Pilote ○ LPR ○ SE ○ Demandeur. ../..

Noyau & Gestion du matériel : exemple d'E/S (suite)

- Pour clarifier et simplifier les développements, les E/S sont virtualisées (ou **synthétiques**)
  - Ex. : abstractions utilisant les primitives open, close, read et write pour manipuler toutes sortes de périphériques (schéma avec *recouvrement*).



Noyau Temps Réel

- Les noyaux Temps Réels sont fonctionnellement spécialisés.
- Noyaux assez légers dont la fonction de base stricte est de garantir les temps (délai) d'exécution des tâches.
- Un exemple courant d'architecture :

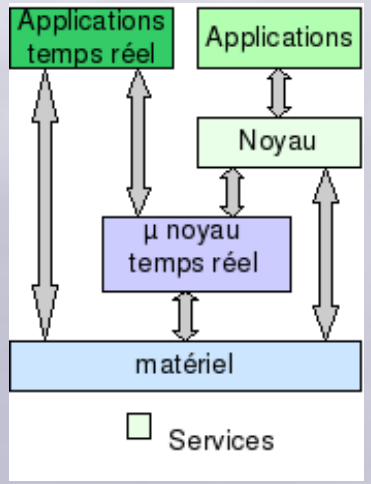


Figure 8 – Micro Noyau TR hybride simplifié (on peut "sauter" le RTOS)

Noyau Temps Réel ...

- La contrainte temps n'est pas une question de rapidité de traitement
  - ➛ La garantie du temps d'exécution en comparaison aux critères temporels de l'application industrielle
- **Diversité** : la réactivité d'un système de freinage ABS n'a pas les mêmes critères temporels que le remplissage d'une cuve de pétrole.
- Très utilisés dans le monde de la **Mécatronique embarquée**
  - ➛ conçus pour tourner sur des plates-formes matérielles limitées en taille, puissance ou autonomie (mais possiblement *multi-taches*).
- Les noyaux TR peuvent adopter (en théorie) toute architecture (ci-dessus)

Exemples :

- **VxWorks** est un noyau propriétaire temps réel très implanté dans l'industrie
- Les systèmes à base de noyau Linux se déploient énormément avec un grand succès via **RTAI** (RTLinux étant breveté).

Noyau Temps Réel ... (suite)

Cas du noyau TR hybride :

- Les noyaux TR fournissent souvent **deux** interfaces séparées,
  - L'une spécialisée dans le Temps Réel et l'autre générique.
  - Les applications Temps Réel font alors appel à la partie Temps Réel du noyau.

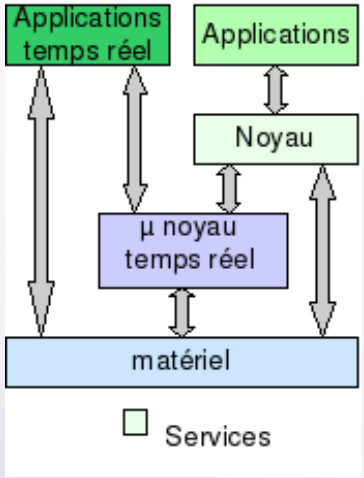


Figure 9 – Micro Noyau TR hypbride simplifié

- Une des architectures souvent retenue est un noyau **hybride** qui s'appuie sur la combinaison d'un **micro-noyau Temps Réel** spécialisé :
  - Allouent du temps d'exécution à un noyau de système d'exploitation non spécialisé.
  - Le système d'exploitation non spécialisé fonctionne en tant que service du micro-noyau Temps Réel.

Temps partagé & Temps Réel

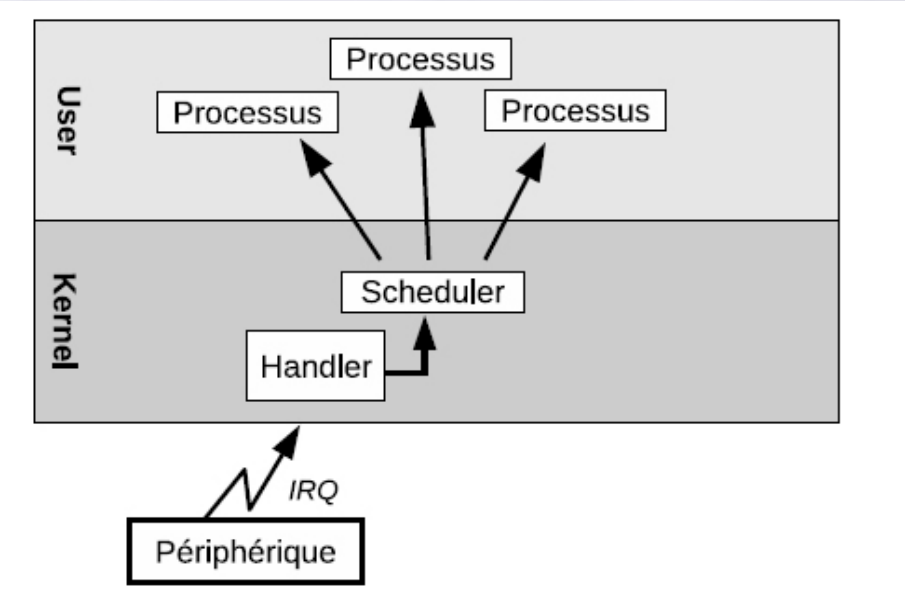
- Système **Temps Réel** vs. Système **Temps partagé**
- **Temps partagé** (comme Linux ou Windows) : question de confort
  - ➛ Le but de l'ordonnanceur est d'assurer que toutes les tâches s'exécutent finalement.
  - ➛ L'OS tient compte de la répartition/régulation de la charge, la date depuis laquelle une tâche donnée est en cours d'exec....
  - ➛ Notion de **quantum** de temps (ou *tick*).
  - ➛ Notion de priorité pour un partage équitable
  - ➛ La commande *nice* (Unix) permet d'intervenir sur les priorités.
  - ➛ Les tâches doivent pouvoir accéder aux différentes ressources partagées.

Temps partagé & Temps Réel (suite)

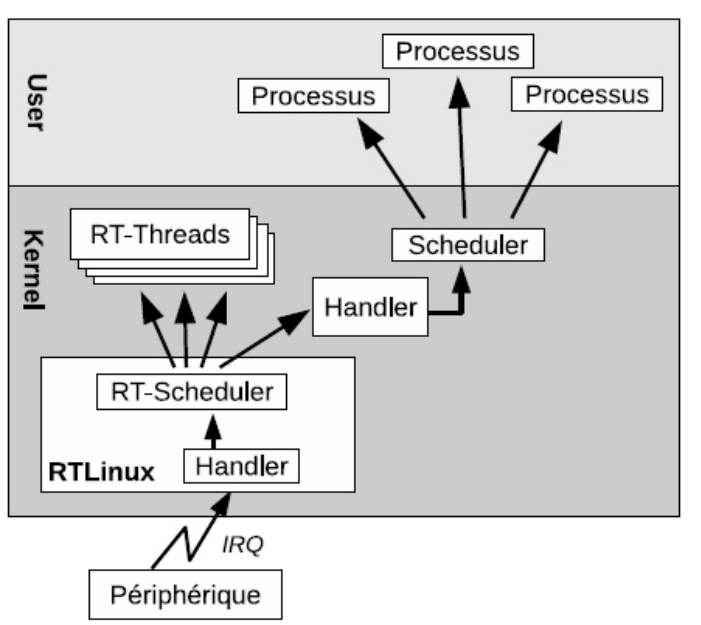
- Le temps partagé (Un*x, Windows) entraine une incertitude temporelle :
  - ➛ Si une tâche écrit sur le disque, les autres doivent attendre la disponibilité du disque (non prévisible).
  - ➛ La gestion des I/O peut aussi générer des temps morts : une tâche peut être bloquée en attente d'E/S.
  - ➛ L'OS temps partagé gère les It s de manière non optimisée (pas grave !) : le temps de latence (temps entre la réception d'un IT et son traitement) n'est pas garanti par l'OS.
  - ➛ L'utilisation de la mémoire virtuelle peut entrainer des fluctuations dans les temps d'exécution des taches....
- **Bref : Il y a du chemin pour passer du Temps Partagé au Temps Réel.**

Temps partagé & Temps Réel (suite)

Cas de Linux (et RTLinux) : une architecture simple.



Ordonnancement dans le noyau Linux standard



Ordonnancement avec RTLinux

Système Temps Réel : détails

- Un système Temps Réel (**RTOS**) :

Définition

Un RTOS est une association "logiciel-matériel" où le logiciel permet une gestion adéquate du matériel en vue de remplir certaines taches dans des limites temporelles bien précises.

- Ou encore (rappel du contexte **Temps**) :

Dans un **RTOS**

l'information, après l'acquisition et traitement, reste encore pertinente (pour être utilisée).

- Ce temps de prise en compte d'information (e.g. une It) doit rester inférieur à la période de rafraichissement de l'information.
- **Exemple** : on reçoit 2 messages très urgents presque en même temps !

Système Temps Réel : détails (suite)

- Un système Temps Réel (**RTOS**) permet de contrôler un Système (éventuellement *Embarqué*)
  - ➛ OS soumis à des contraintes de temps de réponse (*résolution*).
  - ➛ Ne veut pas forcément dire : **réponse rapide !**
    - un processeur rapide apprécié.

Exemple : dans un système de navigation d'un **drone** (*faux bourdon*), les contraintes seront fortes et nécessiteront une certaine rapidité.

Mais pour un système TR de surveillance du niveau d'une cuve d'eau de 50 m3 avec une arrivée de débit 10 l/min et une sortie, le système dispose d'un temps bien plus long.

- Un RTOS : *savoir réagir* **à temps** *à un stimuli extérieur*.
  - ➛ Le RTOS peut être embarqué (dans une voiture, avion, robot, ...)
  - ➛ Il peut se charger de l'automatisation d'un processus
- N.B. : importance de la *complexité* (temps, espace) des algorithmes !

Le temps dans les RTOS

- Dans un RTOS, les informations ont une validité dans le temps :
  - ➛ ne doivent pas surgir ni trop tôt, no trop tard.
  - ➛ constituent les bases des Contraintes de temps.
- **3 types de contraintes temporelles p/r l'information :**
  - ▶ 1- strictes : correspondent à l'invalidité des infos à la fin de l'échéance;
  - ▶ 2- critiques : correspondent aux infos provoquant des troubles (graves) de fonctionnement (à prévoir dès la conception; nécessite une logique câblée éventuelle).
    - ➛ Le non respect de ces contraintes peut même entrainer des dégâts très graves **(mort d'homme)** dans certains cas (freins auto, pilotage automatique, pilotage avion, ...).
  - ▶ 3- relâchées : l'information garde une certaine validité après l'échéance.

Le temps dans les RTOS (suite)

Important : Dans un *Système temps réel*, la justesse dépend à la fois de la justesse fonctionnelle (traitements = algorithmes) mais aussi de l'obtention des résultats dans le délai imparti. Hors ce délai, les meilleurs résultats sont inutiles !.

- $2 + 2 = 4$ n'a d'intérêt que si 4 arrive à temps !

Choix d'un Processeur (orienté par les contraintes temporelles) :

• Dans un processus chimique, si la durée de réaction est d'une seconde, ce n'est pas la peine d'avoir un processeur 32 bits performant : un 8 bit (ou même 4) fera l'affaire.

• Si ce temps est d'une $\frac{1}{10}\mu s$. (e.g. dans une réaction nucléaire), il faut alors un processeurs 32 bits performant + ce qui va autour

➛ Des exemples : ARM, ColdFire, ...

• Au besoin : on peut prévoir une logique câblée pour des Contraintes temporelles très courtes (indépendante de l'UC).

Critères temporels d'un RTOS

- Un RTOS peut gérer les Contraintes du temps de 2 manières :
  - **Soft real time** ◦ gestion **égalitaire** du temps CPU :
    - ➛ un retard de l'ordre de 1/2 seconde n'est pas grave pour un système multimédia qui peut rater l'affichage d'une image ou 2 (avec 24 images/sec.) sur la totalité.
    - ➛ Ici, On est proche des systèmes Temps Partagés.
  - **Hard real time** ◦ gestion **totalitaire** du temps CPU :
    - ➛ il faut absolument respecter les délais :
      - ◦ e.g. . un contrôleur de centrale nucléaire.

Critères temporels d'un RTOS (suite)

Hard RT :

- Un RTOS hard doit répondre à des critères fondamentaux :
  - le **déterminisme logique** : les mêmes entrées doivent produire les mêmes résultats
  - le **déterminisme temporel** : une tache donnée doit absolument être exécuté dans les délais impartis (notion d'**échéance**).
  - La **fiabilité** : l'OS doit être disponible un OS embarqué)
    - ➛ Question cruciale d'intervention d'un opérateur p/r disponibilité
  - La **prévisibilité** (prédictible) :
    - ➛ Quelque soient les contraintes temporelles (échelonnées entre quelques micro-secondes à quelques secondes) :
      on doit pouvoir prédire le comportement.

Illustration de la Prévisibilité

- Si l'on ne peut pas garantir les délais lorsqu'un système est chargé :

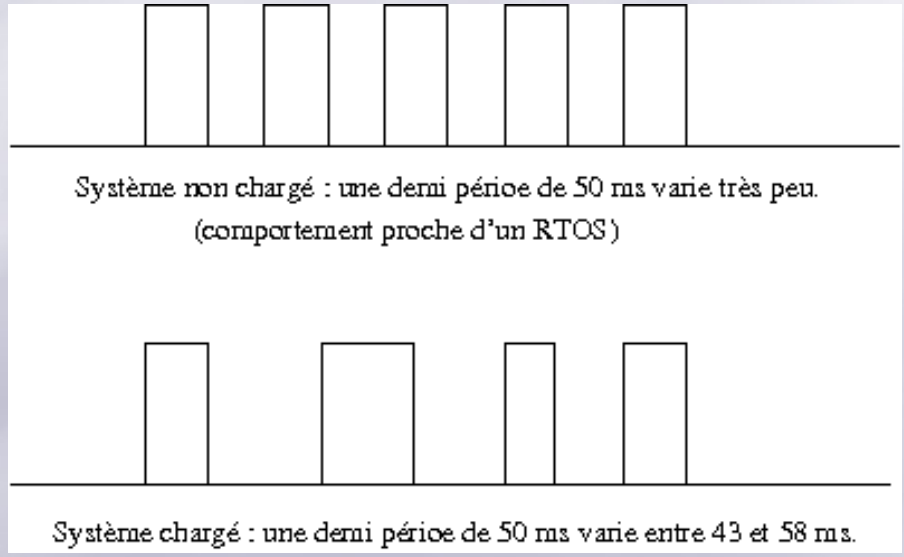
```
#include <....>
#define LPT 0x378
int main(...) {
 setuid(0);
 if (ioperm(LPT, 1, 1) < 0) {perror("pb. ioperm()"); exit(-1);}
 while (1) {
      outb(0x01, LPT);     // 1 sur la sortie LPT
      usleep(50000);       // attente 50 milli-sec

      outb(0x00, LPT);     // 0 sur la sortie LPT
      usleep(50000);
  }
 return(0);
}
```

Illustration de la Prévisibilité (suite)

• Dans ce code, on devrait avoir, sur un oscilloscope, un signal carré d'une demi période T/2 de 50 millisecondes.

• Or, dès que l'on charge le système (écriture directe sur le disque), la période n'est plus garantie.



• Sur un système TR, les délais sont bien mieux garantis (variations ± 0.1).

Fonctionnement d'un RTOS

Rappel :

• **Dans un RTOS** : on doit pouvoir acquitter un *stimuli* (p. ex une interruption = **it**), traiter l'information et éventuellement signaler un résultat à l'utilisateur (réveil d'une tâche, libération d'un *sémaphore*.)

➛ Et tout cela avant que l'it suivante n'arrive (ou alors il ne faut pas la perdre !).

➛ Comment traiter les *its* : scruter (Polling)? stocker ? ignorer ?

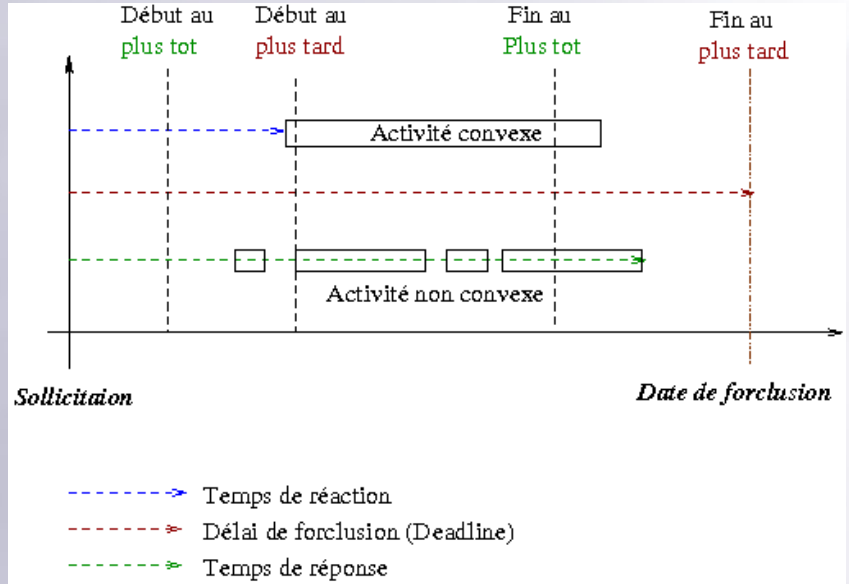
• Cela dépend du processus extérieur qui impose ses Contraintes (et non le contraire).

☞ Polling utilisable dans Mono-Tâche.

RTOS et min-max délai

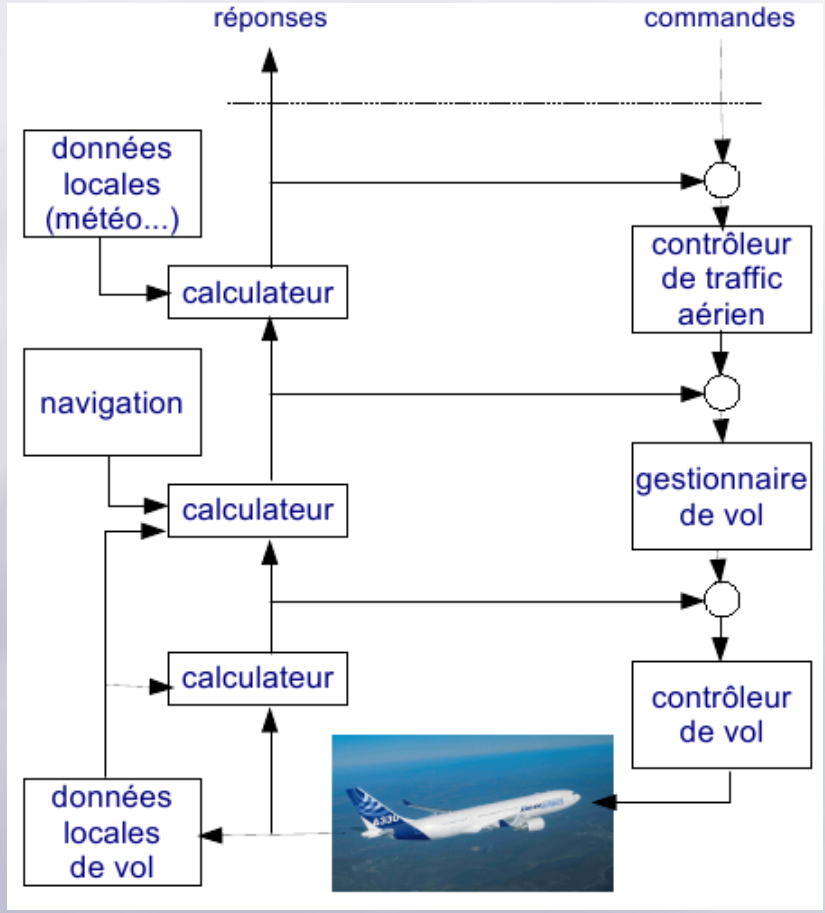
- Différentes activités pouvant communiquer ensemble ∘ *agents*.
- Chaque activité a ses contraintes temporelles précises.
  - ➛ Après l'avènement déclencheur, l'agent doit réagir dans un temps mini et surtout avant un délai maxi (Deadline) = au plus tard.
  - ➛ De même, on définit un temps minimal avant lequel l'agent doit avoir fini son activité = Fin au plus tôt.
  - ➛ On contraint également le **début** : au + tôt & au plus tard.

- La principale contrainte : finir avant le deadline.
- L'activité peut être *convexe* : se déroule sans interruption.
- Les activités peuvent avoir une condition d'activation ou un évènement déclencheur.



Exemples RTOS (Mono / Multitâches)

Rappel : Un exemple :



Exemples RTOS (Mono / Multitâches) (suite)

- Contrainte de temps dans un RTOS
  - Un RTOS "se plie" à son environnement le plus contraint.

Un exemple : contrôleur de vol (Résolution = un *cycle* = ici $1/180s \approx 5ms$.)

Tous les 1/180 s faire	1/180 s
lecture/validation des capteurs sélectionnés	
Chaque 6 cycles (30 Hz) : tâches avionique	1/30 s
lecture claviers, sélection mode	
normalisation des données, transformation des coordonnées	
mise à jour des références de trajectoire	
Chaque 6 cycles (30 Hz) : calculs	
loi de contrôle phase 2 pour la dérive	
loi de contrôle phase 2 du roulis	
loi de contrôle phase 2 du travers	
Chaque 2 cycles (90 Hz) : en utilisant les résultats de 2 cycles	1/90 s
loi de contrôle phase 1 pour la dérive	
loi de contrôle phase 1 du roulis et coordination des 2 axes	
Calcul de la loi de contrôle globale	
Sortie des commandes pour les actionneurs	
Auto-test	

Exemples RTOS (Mono / Multitâches) (suite)

Exemple (plus rare) : **BD temps réel**

- Bases de données de plus en plus utilisées dans un environnement TR :
  - gestion des états partagés de systèmes
  - gestion de trajectoires de mobiles
  - cotations financières

- La différence avec les bases de données classiques tient à :
  - la nature transitoire des informations stockées
  - la nécessité d'assurer une cohérence temporelle (absolue ou relative) des données :
    - ➛ cohérence absolue : toute donnée a un âge inférieur à un seuil
      - (question de durée de validité d'un évt.)
    - ➛ cohérence relative : la différence d'âge entre 2 données est inférieure à un seuil
      - (question du débit/rythme)

Exemples RTOS (Mono / Multitâches) (suite)

Contraintes temporelles dans les **BDs temps réel** :

Application	*Taille (octets)*	*temps de réponse moyen*	*temps de réponse maxi*	*cohérence absolue*	*cohérence relative*	*conservation*
Contrôle du traffic aérien	20000	0,5 ms	5 ms	3 s	6 s	12 h
Mission aérienne	3000	0,05 ms	1 ms	0,05 s	0,2 s	4 h
Mission spatiale	5000	0,05 ms	1 ms	0,2 s	1 s	25 ans
Contrôle de procédé en usine		0,8 ms	5 s	1 s	2 s	24 h

Approches en conception des RTOS

1- Synchrone, Déterministe : la plus courante.

➛ Une boucle infinie, exécution des instructions l'une après l'autre, aucun mécanisme bloquant ne doit se présenter; sinon système bloqué.

➛ les mêmes causes produisent les mêmes effets avec les mêmes temps d'exécution

2- Multitâche Préemptive : plusieurs boucles s'exécutant en parallèle.

➛ accès au processeur par la tâche la plus prioritaire.

➛ Il faut un noyau Temps Réel pour gérer l'activation de différentes tâches.

Autres :

- *Approche Coopérative* : peu utilisée (*Oberon*), délicat.

- *Approche Objet* : plus récente.

➛ Encapsulation d'objets/ "comportement" (compétence), exécution en parallèle.

Construction d'un RTOS informatique

A propos des deux approches principales de conception d'un RTOS.

- Autres terminologies :
  1. **Synchrone** : un *générateur* **cyclique** qui périodiquement capte la dynamique du procédé par échantillonnage, effectue des *calculs*, prépare des actions et envoie des commandes sur les actionneurs.
  2. **Asynchrone** (mutiltâches): un système **réactif** qui répond instantanément aux stimuli provenant du procédé selon la dynamique de celui-ci.
  3. **Possible** : une organisation qui **combine** ces deux approches en **ordonnançant** l'exécution d'activités **périodiques** et d'activités **apériodiques**.

Construction d'un RTOS informatique (suite)

Le déroulement des activités peut être :

1) **Cyclique / Synchrone** :

un programme unique dont la structure générale est une boucle sans fin.

- approche *monotâche*.

2) **Réactif / Asynchrone** (ou **mixte**) :

un programme (code) dont la structure est un ensemble de séquences de traitement susceptibles de se dérouler **en parallèle**.

- approches *multitâches*.

- Aperçu :
  - Avantages et limites du *mono-tâche*
  - Vers l'approche *multitâches*

Construction d'un RTOS informatique (suite)

Un exemple : on souhaite présenter à l'écran des informations provenant de k capteurs (température, humidité, vent, pression, etc) + décision selon les consignes.

- **Deux manières :**

  1 - Une seule boucle : *pour tout capteur* C_i

  consulter C_i

  décider en fonction de la consigne

  2 - Une tâche est associée à chaque capteur.

  - Dans chaque routine (exécutée en parallèle avec les autres) :

    consulter C_i

    décider en fonction de la consigne

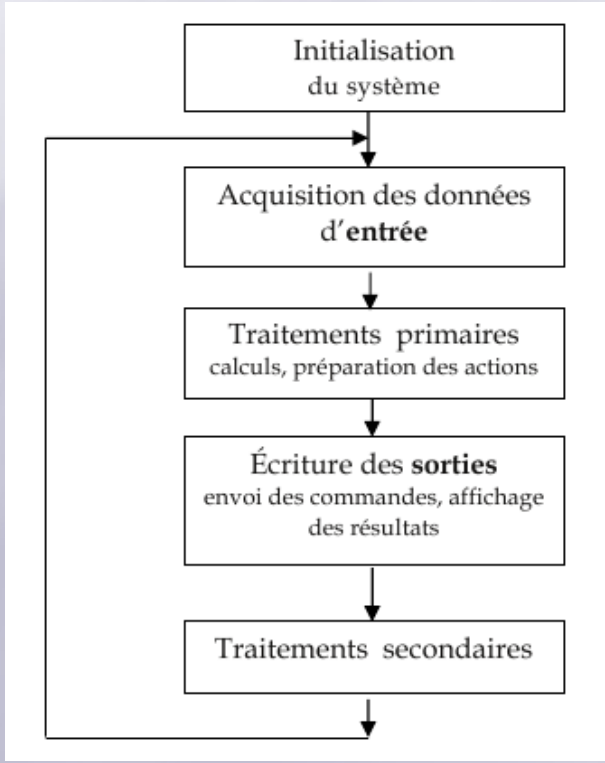
  - Centralisé / réparti : **arbitre**.

Approche Mono-Tache

- Deux approches Mono et Multi-taches exploitées dans les RTOS.
  - L'approche Mono-tâche suffit parfois (même souvent !).

Caractéristique Mono tâche :

une seule tâche sous la forme d'un programme (une boucle principale) :



Approche Mono-Tache (suite)

- Architecture simple, traitement séquentiel des entrées, des phases de calcul et des sorties.
- L'interaction du système avec son environnement peut se faire par :
  - interruption,
  - scrutation cyclique des entrées/sorties
  - une combinaison de ces deux modèles d'interaction.
- Quelques exemples d'évènements pouvant générer une interruption:
  - Réception d'un caractère par le contrôleur d'une liaison RS232,
  - Fin d'une conversion analogique-numérique,
  - Arrivée à échéance d'un compteur de temps.
- On reviendra sur les systèmes Mono-Taches (cf. les PICs sans RTOS)

Approche Mono-Tache (suite)

Une architecture possible du programme de l'application :

(Monotâche)

```
Effectuer l'initialisation générale du système ;
Initialiser le timer d'activation périodique du corps de la boucle ;
TantQue (VRAI) faire :
    Attendre la prochaine occurrence du signal périodique ;
    Acquérir les données d'entrée ;
    Effectuer les traitements primaires ;
    Mettre à jour les sorties ;
    Effectuer les traitements secondaires ;
FinTantQue
```

Les limites du monotâche

• Temps indéterminé (mais borné) entre l'apparition d'un évènement asynchrone et le traitement par le programme (par scrutation cyclique ou par interruption)

• Inadaptée si la complexité de l'application à réaliser croit.

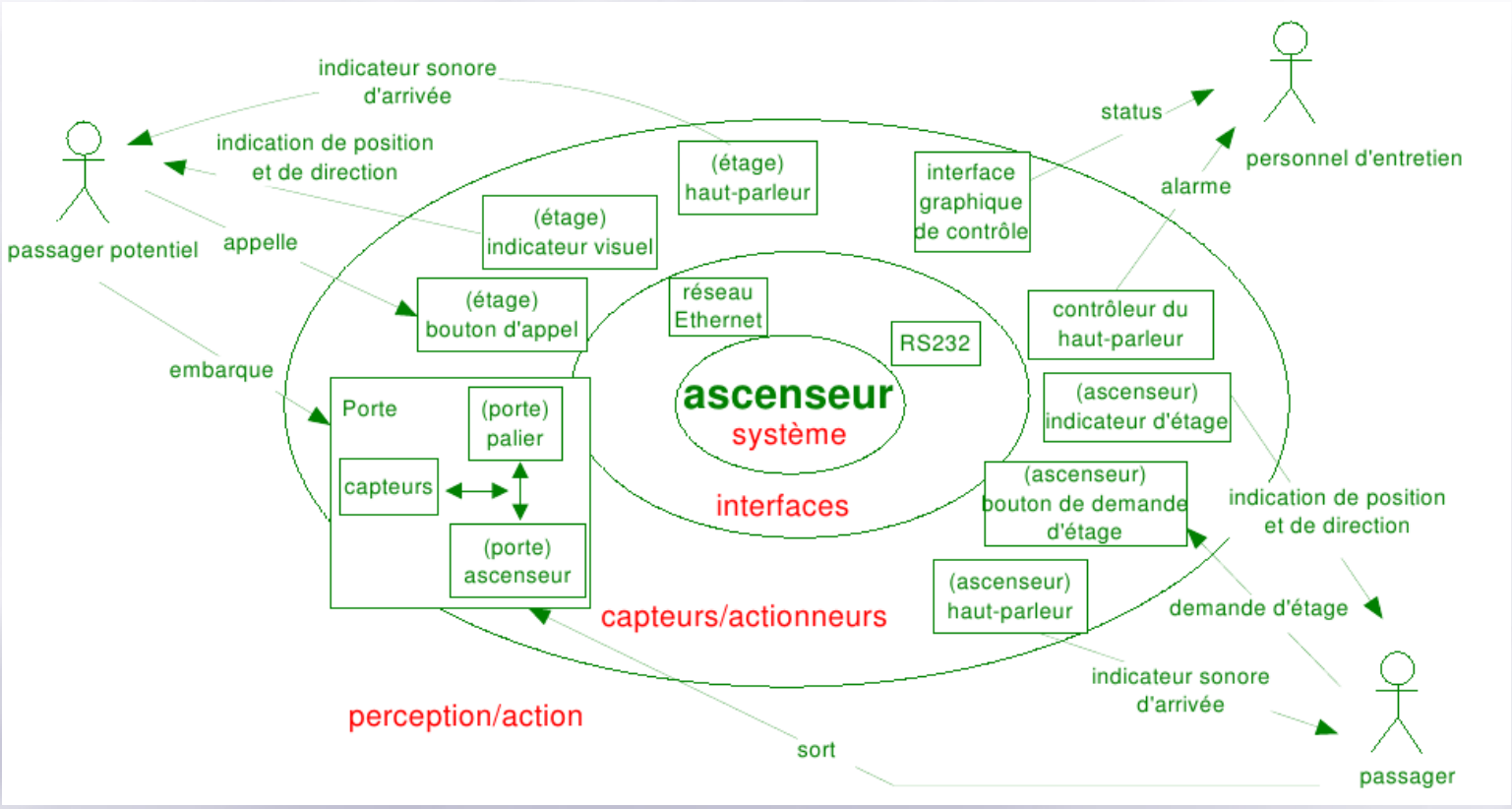
• Tributaire du nombre et surtout de la diversité des périphériques à gérer (diversité en particulier au niveau de la fréquence des traitements associés).
◦ L'accroissement des entrées/sorties à gérer peut conduire à un programme mal structuré qui devient rapidement illisible et impossible à maintenir.

• Souvent, l'ajout d'un nouveau traitement nécessite une remise en cause de l'existant.

➛ Test et intégration unitaire difficile.

Vers le Multi-Taches

Un ascenseur "compliqué" nécessitant un RTOS multitâches (asynchrone).



Conclusions provisoires (RTOS)

Un système temps réel, c'est à la fois :

- un système informatique en interaction avec son environnement,
- un système qui doit effectuer des traitement simultanés,
- un système qui effectue ses activités en respectant des contraintes de temps,
- un système dont le comportement est à la fois prévisible et déterministe,
- un système qui possède une grande sûreté de fonctionnement.

Conclusions provisoires (RTOS) (suite)

Dans une architecture Mono-Tâche ou Coopératif,

• La structuration en tâches concurrentes et coopérantes pose des problèmes spécifiques.

- ◦ Problèmes de la coopération et la communication entre les tâches,
- ◦ Problèmes d'organisation dans le temps de l'exécution des tâches.

• Les concepts et problèmes d'un système temps réel multitâche regroupés autour des quatre thèmes suivants :

1. Les tâches : définition et caractéristiques
2. L'ordonnancement des tâches : l'organisation de leur exécution
3. L'accès concurrent des tâches à des ressources partagées
4. La synchronisation et la communication entre les tâches.

Exemples simples de RTOS Multitâches

Quelques exemples simples (FreeRTOS)

```
static void vTestTask( void *pvParameters );
static int task1id = 1;
static int task2id = 2;

portSHORT main( void )
{
    xTaskCreate( vTestTask, "Task 1", 1000,
    (void *) &task1id, tskIDLE_PRIORITY + 2, NULL );
    xTaskCreate( vTestTask, "Task 2", 1000,
    (void *) &task2id, tskIDLE_PRIORITY + 2, NULL );

    printf("starting scheduler...\n");

    vTaskStartScheduler( pdFALSE );

    return 0;
}

static void vTestTask( void *pvParameters )
{
    int taskId = *((int *)pvParameters);
    printf("Task %d yielding.\n", taskId);
    taskYIELD();
    printf("Task %d resumed.\n", taskId);
    printf("Task %d yielding.\n", taskId);
    taskYIELD();
    printf("Task %d resumed.\n", taskId);

    /* Tasks must be implemented as continuous loops */
```

Exemples simples de RTOS Multitâches (suite)

```
    while(1);
}

Resultats : Task 2 Yielding Task 1 Yielding  Task 2 resumed  Task 2 Yielding  ...
```

- L'exemple précédent est un test effectué pour les aspects coopératifs du RTOS.
- L'exemple suivant teste les aspects préemptifs : ../..

Exemples simples de RTOS Multitâches (suite)

```
static void vTestTask( void *pvParameters );
static int task1id = 1;
static int task2id = 2;
static int task3id = 3;
static int task4id = 4;
xSemaphoreHandle xSemaphore = NULL;

portSHORT main( void )
{
    xTaskCreate( vTestTask, "Task 1", 1000,
    (void *) &task1id, tskIDLE_PRIORITY + 2, NULL );
    xTaskCreate( vTestTask, "Task 2", 1000,
    (void *) &task2id, tskIDLE_PRIORITY + 2, NULL );
    xTaskCreate( vTestTask, "Task 3", 1000,
    (void *) &task3id, tskIDLE_PRIORITY + 2, NULL );
    xTaskCreate( vTestTask, "Task 4", 1000,
    (void *) &task4id, tskIDLE_PRIORITY + 2, NULL );

    printf("starting scheduler...\n");

    vSemaphoreCreateBinary( xSemaphore );
    vTaskStartScheduler( pdTRUE );

    return 0;
}




/*-----------------------------------------------------------*/
static void vTestTask( void *pvParameters )
```

Exemples simples de RTOS Multitâches (suite)

```
{
    int taskId = *((int *)pvParameters);
    while(1)
    {
        if ( cSemaphoreTake( xSemaphore, 0 ) )
        {
            printf("Task %d running.\n", taskId);
            cSemaphoreGive( xSemaphore );
        }
    }
}

Tests :
Task 4 running
Task 1 running
Task 2 running
Task 3 running
Task 4 running
....
```

Opérations sur les processus

Réalisés par le SE :

- *créer / détruire*
- *mettre en attente / réveiller*
- *suspendre / reprendre*
- *modifier la priorité*

Création Processus :a été vue plus haut.

Attendre / Réveil

- "en attente" : demande de ressource (autre que l'UC) non disponible;
- "réveil" : quand le S.E. peut lui affecter les ressources (état ◦ "prêt")

Opérations sur les processus (suite)

Suspendre / reprendre

- permet de donner "son tour" à chaque processus.
- à la reprise ⇒ "prêt" ou "en attente" (selon l'état avant la suspension)
- **Équité** : ne pas suspendre un processus trop souvent/trop long temps;
- un processus en attente de données (d'un autre processus) peut être suspendu (Attente ⇒ suspension)
- le SE peut suspendre un processus (pour efficacité, performances, ...)

Changement de priorité

- Utilisé par l'ordonnanceur (augmentation/baisse)
  - ➛ Gestion des priorités et Famine / Équité

It et Signaux

- On a déjà parlé des It.
- Trois types d'it.

Hardware	Software (Exception/Trap)	Software (Instruction set)
Interrupt Request(IRQ) sent from device to processor.	Exception/Trap sent from processor to processor, caused by an exceptional condition in the processor itself.	A special instruction in the instruction set which causes an interrupt when it is executed.

- Table des signaux de Linux

Signal	Name	Description
SIGHUP	1	Hangup (POSIX)
SIGINT	2	Terminal interrupt (ANSI)
SIGQUIT	3	Terminal quit (POSIX)
SIGILL	4	Illegal instruction (ANSI)
SIGTRAP	5	Trace trap (POSIX)
SIGIOT	6	IOT Trap (4.2 BSD)
SIGBUS	7	BUS error (4.2 BSD)
SIGFPE	8	Floating point exception (ANSI)
SIGKILL	9	Kill(can't be caught or ignored) (POSIX)
SIGUSR1	10	User defined signal 1 (POSIX)
SIGSEGV	11	Invalid memory segment access (ANSI)
SIGUSR2	12	User defined signal 2 (POSIX)
SIGPIPE	13	Write on a pipe with no reader, Broken pipe (POSIX)
SIGALRM	14	Alarm clock (POSIX)
SIGTERM	15	Termination (ANSI)
SIGSTKFLT	16	Stack fault
SIGCHLD	17	Child process has stopped or exited, changed (POSIX)
SIGCONTv	18	Continue executing, if stopped (POSIX)
SIGSTOP	19	Stop executing(can't be caught or ignored) (POSIX)
SIGTSTP	20	Terminal stop signal (POSIX)
SIGTTIN	21	Background process trying to read, from TTY (POSIX)
SIGTTOU	22	Background process trying to write, to TTY (POSIX)
SIGURG	23	Urgent condition on socket (4.2 BSD)
SIGXCPU	24	CPU limit exceeded (4.2 BSD)
SIGXFSZ	25	File size limit exceeded (4.2 BSD)
SIGVTALRM	26	Virtual alarm clock (4.2 BSD)
SIGPROF	27	Profiling alarm clock (4.2 BSD)
SIGWINCH	28	Window size change (4.3 BSD, Sun)
SIGIO	29	I/O now possible (4.2 BSD)
SIGPWR	30	Power failure restart (System V)

Multi-tâche : Processus

Processus : notion de base introduite par MULTICS

- une abstraction de l'activité d'un processeur (exécute 1 processus) .

• Lorsqu'il y a plus d'un programme destiné à un processeur (= ressource non partageable), la notion de *processus* permet d'exprimer qu'un *programme* est en cours d'exécution; bien qu'il ne progresse pas (e.g. il est en attente).

• Analogie : vous amenez votre mobylette au garage pour réparation

- Le lendemain, vous les appelez. - On vous répond : *on s'en occupe*.
  - Mobylette effectivement entre les mains d'un mécanicien (*sur le pont*)
  - Mobylette en *attente* d'une pièce commandée
  - Mobylette n'a pas besoin de pièce mais attend son tour (*suspendue*)

☞ A la réception de la pièce commandée : on se charge de la mobylette dès qu'un mécanicien est libre (y a déjà une sur le pont / + prioritaire ?).

Multi-tâche : Processus (suite)

Partage du temps du processeur ➛ Notion de **commutation**

- Exemple : une dame, livre de recettes, gâteau, ...
  - recette : l'algorithme (code du programme)
  - la dame : le processeur
  - les ingrédients : données du programme
  - lire la recette, trouver les ingrédients et cuisiner (processus) un gâteau

    Mais, on sonne ! qui est-ce ?
  - On sonne ◦ Interruption, priorité, contexte, reprise, ...
- Aller Ouvrir la porte = suspendre l'activité actuelle.
- La dame reprendra la cuisine après avoir répondu (porte)
- Chaque **processus** a son propre programme (recette, ouverture porte)

1 processus : une abstraction de l'activité d'un processeur

➛ cette activité possède un code (programme), des données et un état.

Multi-tâche : Processus (suite)

- **Un programme** =

  *entité statique associée à une suite d'instructions (*recette*);*

- On distingue un programme de son exécution ⇒ notion d'**état**

- **Un processus** =

  *entité dynamique associée à la suite d'actions (*cuisiner*) réalisées par un programme (recette) lors d'une exécution particulière.*

  ◦ Cette suite est indépendante du nombre de fois et des instants où le processus ba été mis en attente de processeur.

Multi-tâche : Processus (suite)

Comme pour l'exemple du garage :

- Dire qu'un processus P est parvenu à l'action "lire la case M" signifie :
  - soit le processeur est réellement entrain d'exécuter l'instruction correspondant à l'action,
  - soit P est mis en attente, il exécutera l'action quand il aura le processeur,
  - soit P a exécuté l'action et a été mis en attente (pour la suite).
- *début d'un programme* = la 1ère instruction (sur le papier !)
- *début d'exécution d'un programme* = l'exécution de la 1ère instruction (début du processus)
- La suite d'instructions exécutées ne permet pas de décrire entièrement comment le programme est exécuté (détails).

  ➛ Information d'**Etat**.

Introduction aux États d'un processus

• Dans l'exemple de la cuisine : le processeur (la dame) est passé d'un processus (gâteau) à un autre, plus prioritaire (porte).

  ◦ Elle pourrait ignorer la porte et continuer avec le gâteau

    → Ce qu'elle fait (gâteau) serait plus prioritaire

  ◦ On l'a interrompu (elle peste !) mais elle reprendra.

• Chaque processus a son propre programme.

  ◦ Lorsqu'elle aura ouvert la porte, la dame reprendra sa recette là où elle l'avait abandonnée (commutation vers l'ancien contexte).

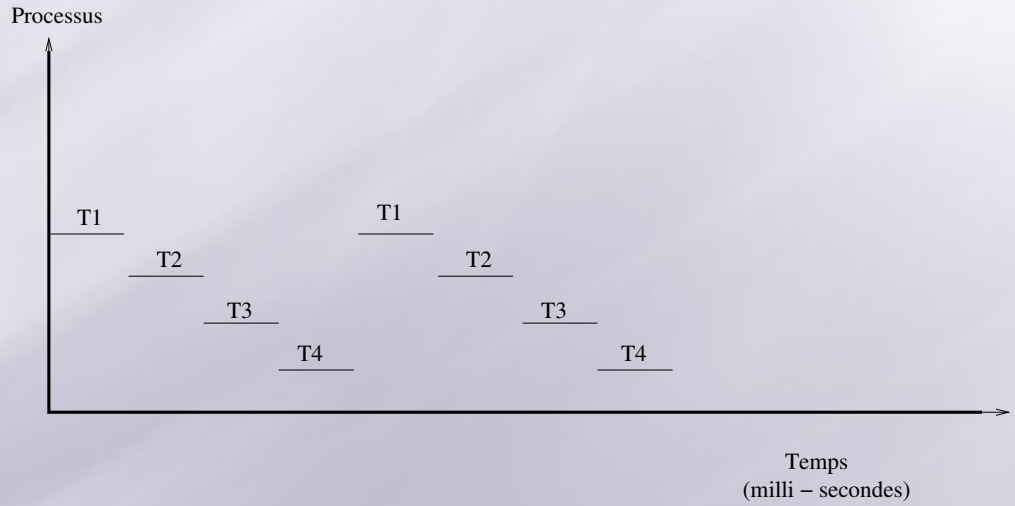
> Un processus est une activité d'un certain type qui possède un programme (code), des données en entrée et en sortie ainsi qu'un état courant.

N.B. : notion d'**arbitre** (*scheduler*) : ici, la maman était son propre Dispatcher / Scheduler (vs. une brigade de cuisiniers dirigée par un chef).

Nécessité de l'abstraction Processus

Comment plusieurs programmes peuvent tourner sur un seul processeur (dans un système multi-tâches / Temps partagé)?

- Gestion multi-tâches sur un processeur



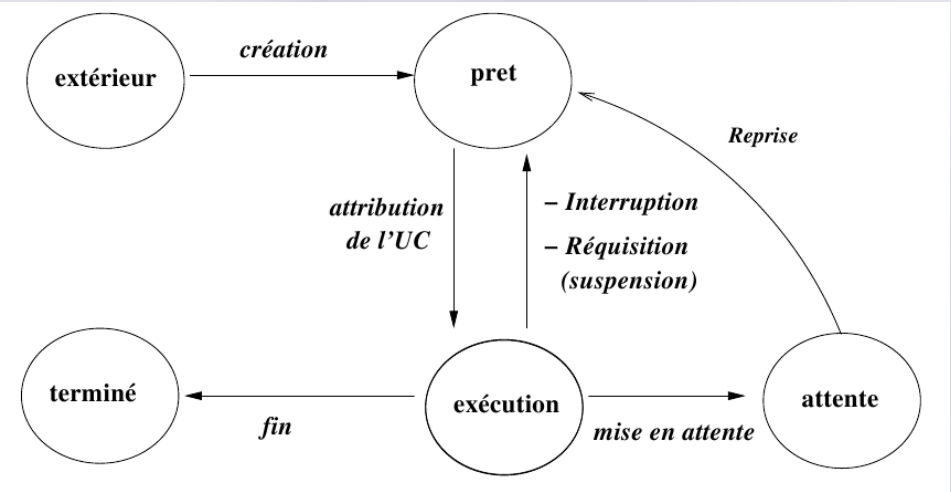
  - Voir le cas des multi-coeurs

✔ Démarrage de tout-cela : un Top départ donné par *init* suivi des processus d'intendance, ... puis ceux des utilisateurs.

États d'un processus

- Un processus ... est soit
  - réellement exécuté (possède l'UC);
  - prêt à être exécuté (en attente du processeur);
  - en attente de ressource/événement (e.g. fin d'un E/S);

Diagramme (simplifié) d'états :



États d'un processus (suite)

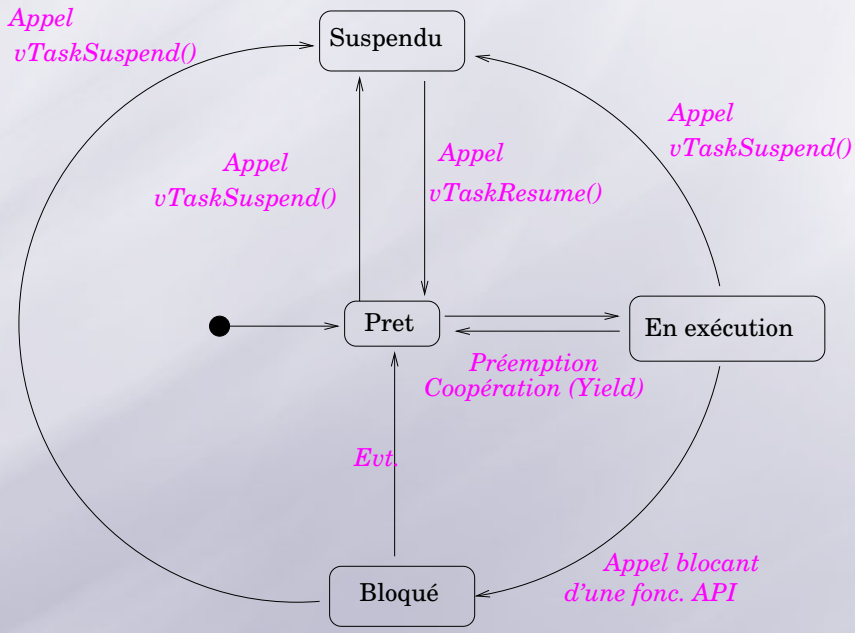
- Les états importants : "en exécution", "prêt", "en attente" (d'autre ressource que le processeur) , "terminé", "extérieur", ...
- Il y a des états associés : "bloqué", "suspendu",...
- On pourra aussi distinguer les causes d'attente et l'attente en mémoire centrale/secondaire,...
- Les états sont modifiés par le SE, sous l'effet d'un **événement**.
- Les événements peuvent être internes au processus :
  ➛ une demande d'E/S fait passer de l'état "en exécution" ⇒ "en attente".
- Les événements peuvent être externes (arrivant via le SE) :
  ➛ l'attribution d'une ressource fait passer de l'état "en attente" ⇒ "prêt".

États d'un processus (suite)

- La nécessité de conserver un *état* pour tout processus
- L'état d'un processus permet sa reprise
- Les données (de reprise) qui caractérisent un processus sont stockées dans une structure appelée *Processus Control Bloc (PCB)*

Fonctionnement d'un RTOS multi-taches

- Diagramme adapté des états spécifique RTOS (cf. *FreeRTOS*)

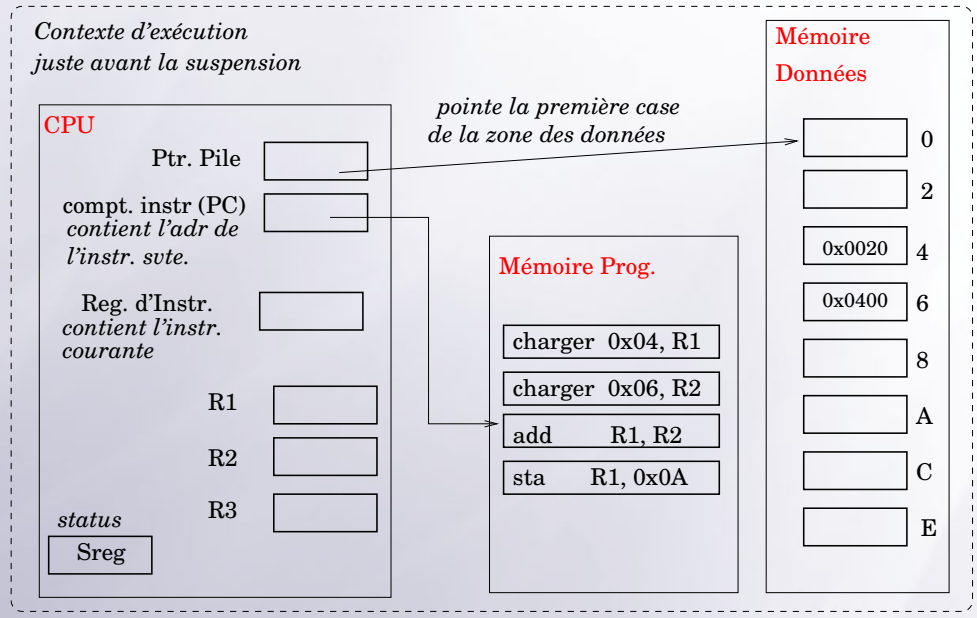


- Tous les labels (magenta) relèvent du noyau.
  - Les actions exécutées dans l'espace noyau.
  - Les files "Prêt", Bloqués", "Suspendus" dans l'espace noyau.

Exemples de fonctionnement en multitâches

Décortiquons une tâche T exécute $Z \leftarrow X + Y$:

- La tâche T est suspendue alors qu'elle allait exécuter ADD.
- Les instructions déjà exécutées par T ont placé des valeurs dans les registres R1 / R2,
  - Lorsque T sera réveillée (*resumed*), l'instruction ADD sera la première à exécuter.
  - La tache T ne sait pas si une autre a modifié R1 / R2/ ..
    → Peu importe : elle sa propre copie !



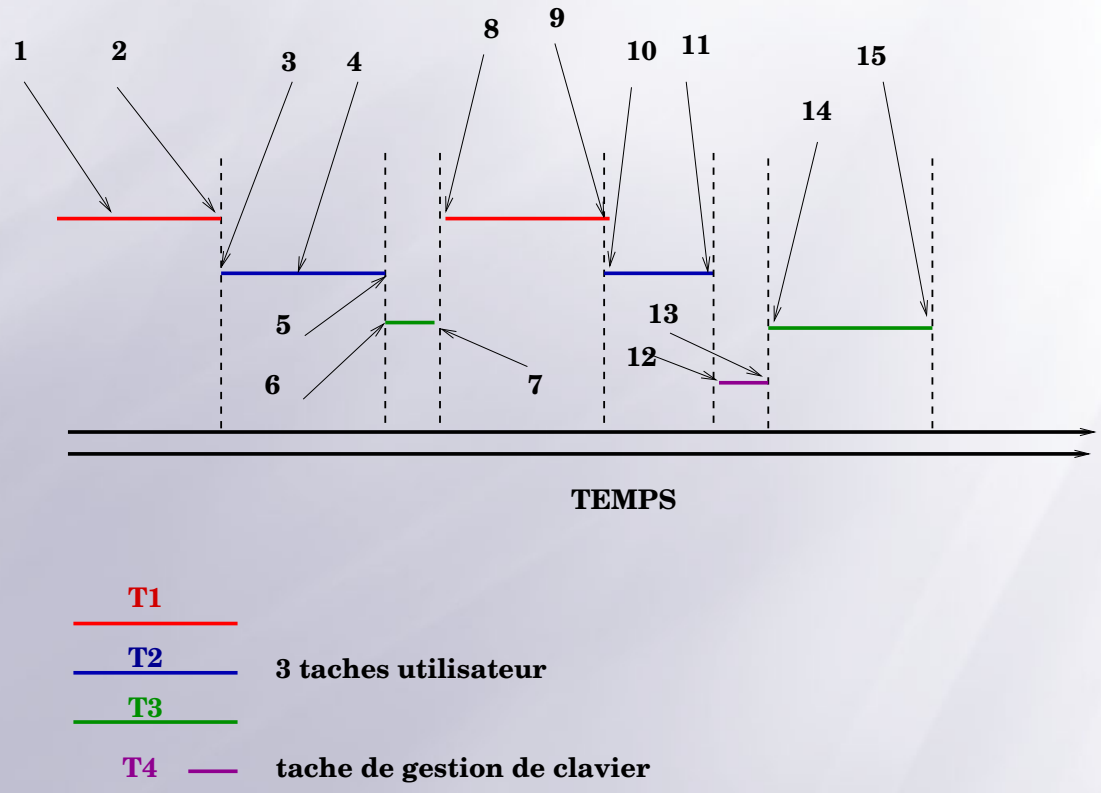
- Un **contexte** de tâche : l'ensemble des informations nécessaires pour restaurer (réveiller) une tache.

☞ On sait ce que veux dire : suspension / préemption / réquisition.

Exemples de fonctionnement en multitâches (suite)

Rappel d'un exemple de déroulement :

1 : T1 s'exécute
2 : Fin quantum de T1
3 : Début quantum T2
4 : T2 block une ressource
(mais n'est pas suspendue)
5 : Fin quantum de T2
6 : Début T3
7 : T3 passe la main
(ou est suspendue)
8 : T1 reprendre
9 : Fin quantum de T1
10 : Début quantum T2
11 : suspension T2
(intervention opérateur)
12 : Début tâche opérateur
(prioritaire)
13 : Fin opérateur
14 : Début quantum T3
15 : Fin quantum de T3



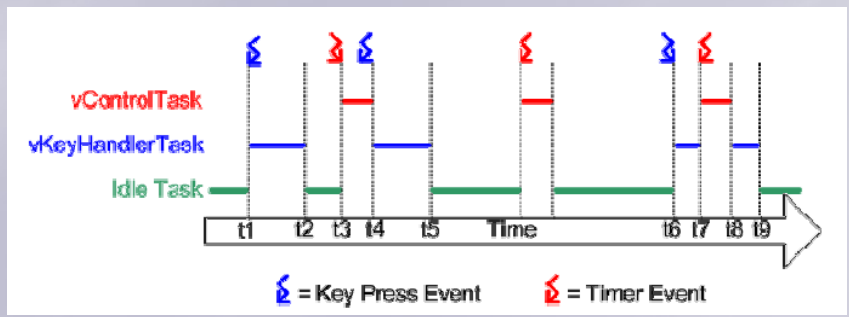
Exemples de fonctionnement en multitâches (suite)

Exemple de déroulement : Soit 3 tâches

vControlTask : tache cyclique , **vKeyHandlerTask** : traiteur (des événements) clavier, Idle.

Priorités : vControlTask > vKeyHandlerTask > Idle.

- Les événements dans le système :
  - en t1 : key pressed
  - en t3 : top timer (pour le cycles de vControlTask)
  - entre t3-t4 : key pressed
  - entre t5-t6 : top timer (interruption)
  - en t6 : key pressed



Exemples de fonctionnement en multitâches (suite)

vControlTask : tache cyclique , **vKeyHandlerTask** : traiteur (des événements) clavier.

- Priorités : vControlTask > vKeyHandlerTask > Idle.

- Au départ, aucune des tâches n'est prête : **vControlTask** attend le début de son cycle (remettre les schéma du cours 1) et **vKeyHandlerTask** attend la frappe d'une touche clavier;
  - L'UC est donné initialement à la tâche Idle (créée par l'OS).
- En t1, une touche est appuyée et *vKeyHandlerTask* peut s'exécuter.
  - ➢ Elle a une priorité supérieure à Idel ○ elle a l'UC.
- En t2, *vKeyHandlerTask* termine (a mis à jour l'affichage);
  - ➢ Elle ne reprendra qu'à la prochaine frappe. ○ Elle se suspend et Idle reprend.
- En t3, un événement *timer* intervient et *vControlTask* peut reprendre (sa priorité supérieure à Idle).
- Entre t3 et t4, pendant l'exécution de *vControlTask*, une frappe clavier a lieu.

  ➢ *vKeyHandlerTask* peut reprendre mais prio(vKeyHandlerTask) < prio(vControlTask) ○ Elle n'est pas chargée.

- En t4, vControlTask termine son cycle et doit attendre le prochain cycle (signalé par un evt. *timer*)

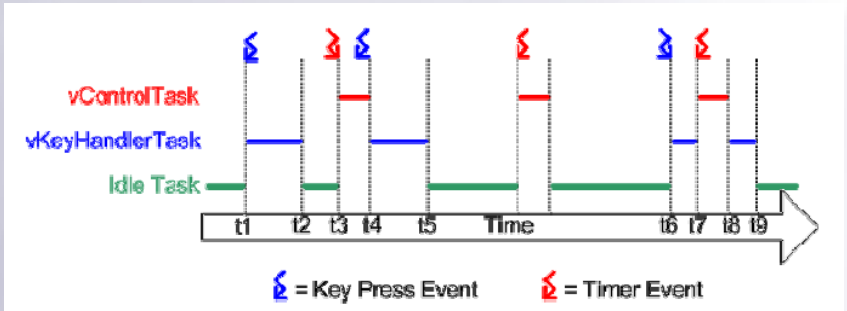
  ➢ vControlTask se suspend et *vKeyHandlerTask* peut reprendre (a une priorité plus élevée que Idle).

  ➢ *vKeyHandlerTask* est chargée et traite la frappe préc.

- En t5, la frappe est traitée et *vKeyHandlerTask* se suspend et attend la prochaine frappe.

  ➢ Encore, Idle peut reprendre.
- Entre t5 et t6, un événement *timer* est traité mais il n'y a pas d'autre frappe.
- En t6, la prochaine frappe a lieu mais avant la fin de son traitement par *vKeyHandlerTask*, un evt. *timer* arrive.
- Maintenant, les 2 taches sont éligibles :

  ➢ prio(vControlTask) > prio(vKeyHandlerTask), *vKeyHandlerTask* est suspendue avant la fin du traitement de la touche et *vControlTask* prend l'UC. ↗



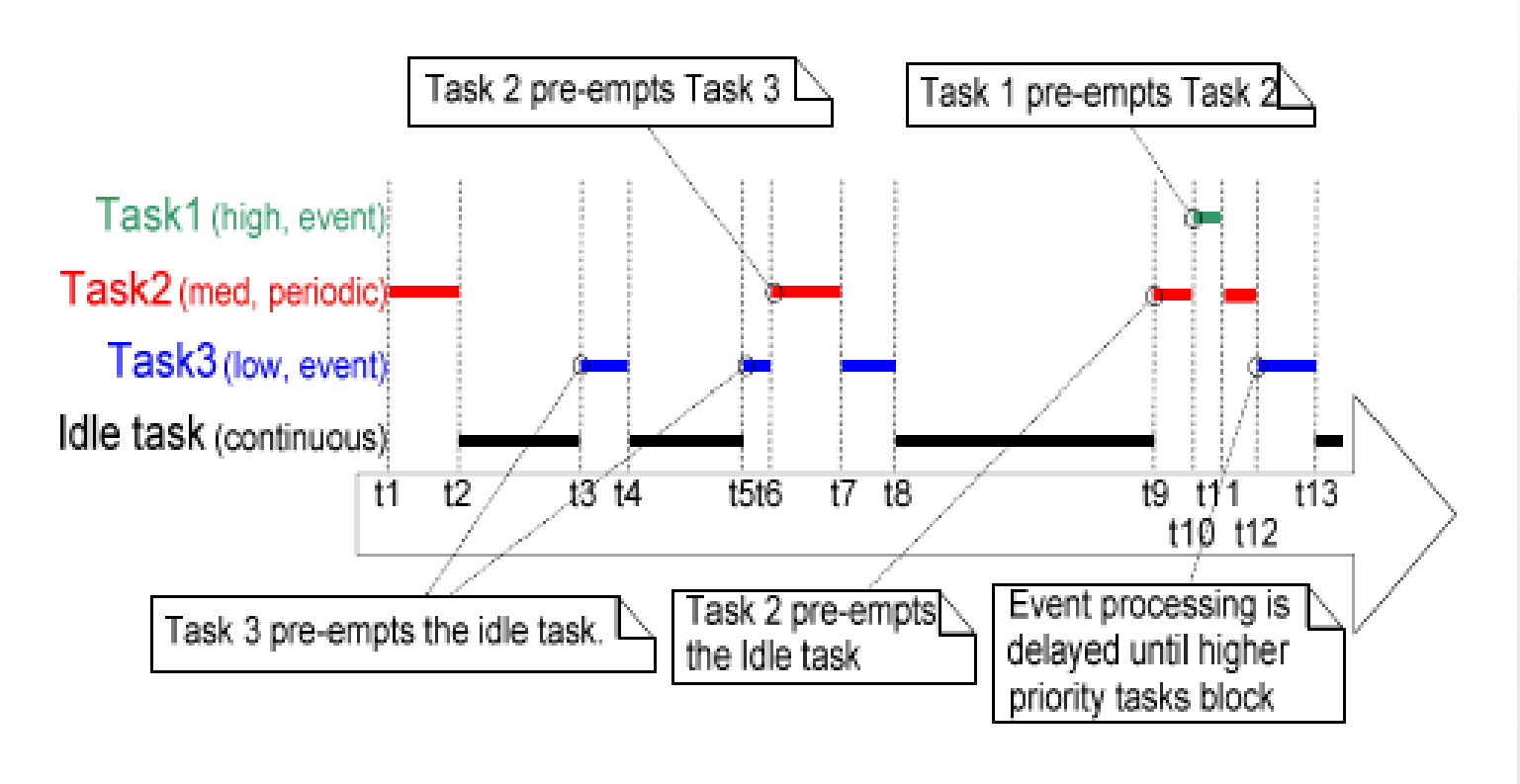
- En t8 :

  vControlTask termine un cycle et se suspend.

  ➢ *vKeyHandlerTask* est maintenant plus prioritaire et donc la frappe peut enfin être traitée.

Exemples de fonctionnement en multitâches (suite)

- Ex. de Préemption.



Tiré de "FreeRTOS User manual " [Richard Barry-2007].

Les événements dans un (RT)OS

- Une communication avec une autre tâche (activité)
- Une it (matérielle/logicielle) : e.g. l'arrivée d'un signal spécifique
- La fin d'un blocage (volontaire)
- En présence d'une IT : on interrompt la tâche en cours (notion *priorité*).
  - ➛ d'où la notion de **préemption**.

Exemples d'interruptions (it) :

- Un capteur a changé d'état, une touche appuyée (opérateur), ...
- La gestion d'une horloge (machine) est un bon ex. d'it (it timer, délai).
  - ➛ A l'arrivée d'un Top, on peut entreprendre plusieurs activités, ordonnancer et lancer des tâches, activer des tâches périodiques (*cron*)....
- **La gestion des activités nécessite un mécanisme de synchro.**
  - ➛ Par exemple, un bip lors de l'appui sur une touche
    - Ici, "alarme" et "lecteur" sont 2 tâches (processus)
  - ➛ Il existent des *schémas* :
    - producteur/consommateur, lecteur/rédacteur, ...

Les événements dans un (RT)OS (suite)

- Exemple : pseudo-code d'un gestionnaire de *timer*
  - Activé lors d'arrivée d'une It **tick**

```
TickISR() :
  incrementer compteur de tick
  si (compteur-tick rend une tache T PRETE)
  alors (selon la politique du OS) :
      placer T dans la file des prets
      (ou générer un evt. si T attend, cf. une condition)
      (ou commuter contexte pour T)
      (ou ....)
  retour d'ISR (différent d'un "return" ordinaire)
```

Contexte : une tache (e.g. Idle) s'exécute, l'It *tick* arrive, *"retour d'ISR"* permet de continuer où l'OS le dit (p.ex à la suite de la tache qui attend le timer)

- *"retour d'ISR"* procède à des actions (save contexte) avant de passer la main.
- Cette fonction peut être exécutée par un PIC dédié.

Ordonnancement de l'UC*

- But : maintenir un taux d'utilisation élevé de l'UC
- Idée première : recouvrement des E/S
- Quelques mesures :
  - Le **taux** d'utilisation de l'UC :
    - théorique : 0 à 100%
    - pratique : 40 à 95%
  - Le **débit**
    - nombre moyen de programmes utilisateurs traités par unité de temps
    - le taux augmente le débit
    - le débit ne tient pas compte de la taille des programmes
  - Optimisation de ces valeurs revient à optimiser :
    - temps de **traitement moyen** d'un système de tâches :
    - la moyenne des intervalles séparant la soumission et la fin des tâches.
    - temps de **traitement total** d'un ensemble de processus donné
    - temps de **réponse maxi** : soumission - réponse à une requête.

Ordonnancement de l'UC* (suite)

• Pour optimiser ces temps, il ne suffit pas de recouvrir les E/S par des calculs.

$\Rightarrow$ l'ordonnanceur doit choisir le processus qui améliore les critères.

• Les algorithmes d'ordonnancement pratiqués sont classés par :

➙ avec (sans) Temps Partagé ,

➙ avec (sans) réquisition,

➙ avec (sans) priorité,

➙ avec (sans) contrainte de précédence (et sans tenir compte des it').

N.B. : sans Temps Partagé : un processus détient l'UC jusqu'à sa fin ou bien jusqu'à demander une E/S.

➙ Variante Obéron : les tâches elles-même organisent et se passent l'UC.

Ordonnancement sans réquisition

Ordonnancement sans réquisition :

- Dans l'ordre d'arrivée : FIFO

  ➙ Méthode souple mais ne minimise pas le temps moyen.

- Dans l'ordre inverse des temps d'exécution :

  ➙ On peut démontrer que cette méthode **minimise** le temps moyen pour l'ensemble des algorithmes d'ordonnancements **sans réquisition**.

  ➙ Cette proposition n'est pas vraie si les tâches entrent au cours d'une assignation (à moins de refaire l'assignation à intervalle X donné).

  ➙ Méthode intéressante pour permettre de comparer, après coup, les performances des algorithmes réellement implantables, aux valeurs min qu'il est possible d'obtenir (avec des estimations).

Ordonnancement avec réquisition

- Nécessaire dans un système à temps partagé (ou TR)
- Un SE doit fournir une machine virtuelle;
- Les tâches longues ne doivent pas bloquer les autres;
- Question d'équité;

Méthode Tourniquet (Round Robin) :

- On utilise des It', commutations, Horloge;
- On choisit un intervalle **Quantum**

  Exemple : dans Vax/VMS, le quantum est multiple de 10ms (intervalle entre 2 tops d'horloge = 10ms); par défaut, quantum = 200ms

- File d'attente circulaire (ou double file);

Ordonnancement avec réquisition (suite)

- L'ordonnanceur choisit une tâche, le distributeur (dispatcheur) lance son exécution;
- La tâche peut termine avant le quantum ($\rightarrow$ remise à 0);
- Sauvegarde du contexte, cumul des temps, choix du prochain, commutation;

- Importance du Quantum :
  - si quantum trop grand : temps de réponse augmente;
  - si quantum trop petit : trop de commutations !
    - ➙ Une idée de CDC6000 : 10 mots d'état en mémoire

- NB : Round Robin avec une double file des prêts.

- Utilisation du **Plus court temps d'exécution restant : PCTER**

  Principe : à chaque réquisition, on calcule le temps d'exécution restant et on choisit la tâche dont le TER est minimum....

Ordonnancement avec priorité

Principe : égalité pour tous (équité)

Mais pour des raisons d'efficacité, les tâches systèmes sont plus prioritaires;

➙ d'autres devraient fournir des résultats avant certaine date.

➙ Solution : attribution de priorités;

- **2 méthodes :**
  - **Fixe** : problème de famine si priorité faible
  - **Variable** : re-calcul des priorités selon l'avancement des exécutions.

Ex : on peut ajouter +1 à la priorité des tâches en attente (et/ou enlever +1 de la priorité de ceux qui s'exécutent).

- Exemple OS2 : proche de VAX/VMS :

➙ augmentation si phase de calcul courte (avec des E/S).

➛ Truc : ajout fictif d'une lecture de temps en temps !

Ordonnancement avec priorité (suite)

Ordonnancement avec plusieurs files de priorité :

- Permet de réduire le temps d'attente d'une tâche sans modifier le quantum;
- Les files ont des N' de priorité fixes;
- Les tâches vont de file en file;
- Le temps de l'UC peut être partagé entre les files selon les priorités; par exemple, la file la plus prioritaire prendra 70% du temps de l'UC.
- Les files peuvent être gérées par des algos différents (adapté au type des tâches de la file);

Autre variante :

- Ordonner hiérarchiquement les files
- On n'exécute une file que si les file de priorité supérieure ont été traitées

$\rightarrow$ famine sauf si les priorités sont modifiées en permanence et les tâches changent de file;

Noyau : Gestionnaire de mémoire

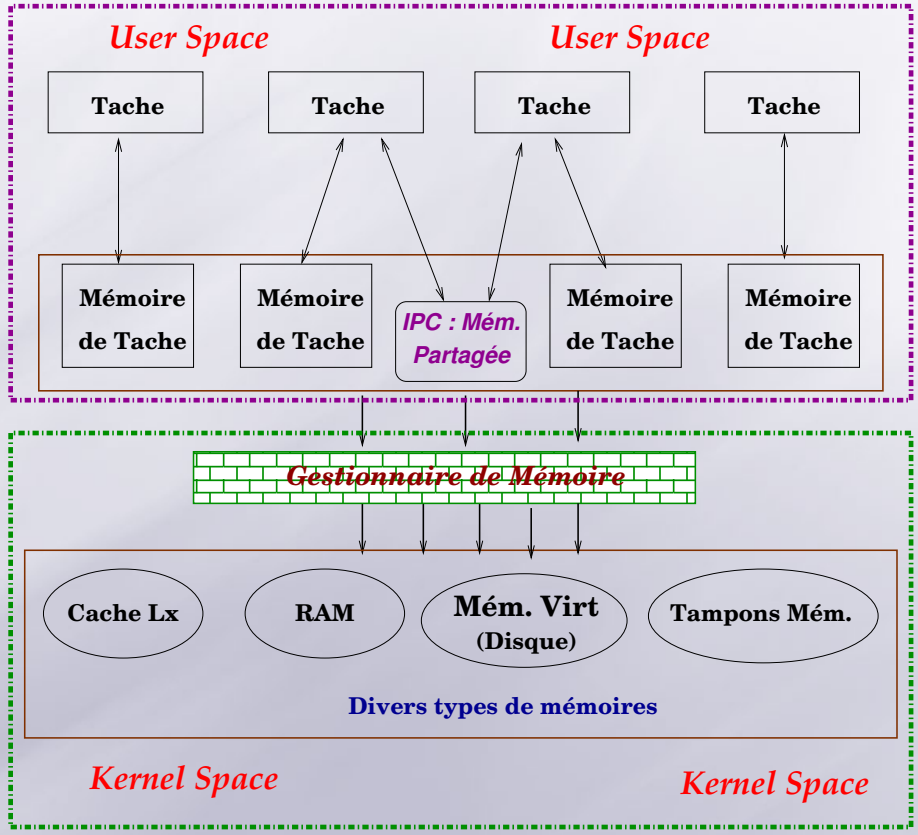


Figure 10 – espaces et le MMU

Noyau : Gestionnaire de mémoire (suite)

- Se charge d'allouer de la mémoire à des processus lorsqu'ils en ont besoin.
- Cette mémoire sera par défaut propre au processus demandeur
- **Mémoire Virtuelle** : abstraction (disque, RAM, Réseau), pagination
- Propriétés de la mémoire manipulée par tout processus :
  - est privée (protégée),
  - peut être étendue jusqu'aux capacités théoriques de la machine ;
  - un processus ne peut pas accéder à la mémoire d'un autre processus
    ➙ (sauf allocations et autorisations spécifiques).
- **MMU** : gère l'allocation demandée par le noyau.
  ➙ Prévient le noyau des violation
  ➙ Le noyau décide d'arrêter / tuer le processus fautif.

Noyau : Gestion du matériel

- Se fait par l'intermédiaire de **pilotes** de périphériques.
- Un pilote= un petit logiciel léger dédié à un matériel donné
  ➙ permet de communiquer avec ce matériel.
- Les pilotes sont généralement inclus dans l'espace noyau et communiquent avec l'espace utilisateur via SVC ,
  ➙ il faut commuter les contextes : on a intérêt à placer les pilotes dans le noyau (question de fluidité/latence).
- Certains périphériques lents (appareils photographiques numériques, outils sur liaison série, etc.) sont/peuvent être pilotés depuis l'espace utilisateur, le noyau intervenant au minimum.

☞ Il y a toujours eu des déplacement des fonctions dans ces 2 espaces.
  ➙ Cela dépend de ce qu'on veut (Perf., Taille, ...) !

Noyau & Gestion du matériel : exemple d'E/S

- Notion de base : *échange de signaux.*
  - ➙ Ex. imprimante : *prête, envoi données, fin, erreur out of paper, spool plein/vide, bourrage...*
- Le SE gère les communications entre les périphériques (malgré les standard différents).
  - ➙ Le SE passe par un Pilote (*Device Driver*, un par matériel)
- Le SE peut charger un module d'impression (+ un démon : e.g. *cupsd*)
- Le service impression est assuré par un logiciel :
  - ➙ Les vieux SE (DOS/CPM) s'en chargeaient eux même (zone SE)
- Le système d'ES : interface entre l'utilisateur et les périphériques (par commandes de haut niveau telles que *read, write, ...*).
  - ➙ Ces commandes provoquent un SVC (superviser call).
  - ➙ On peut sous traiter par un processeur *esclave*.

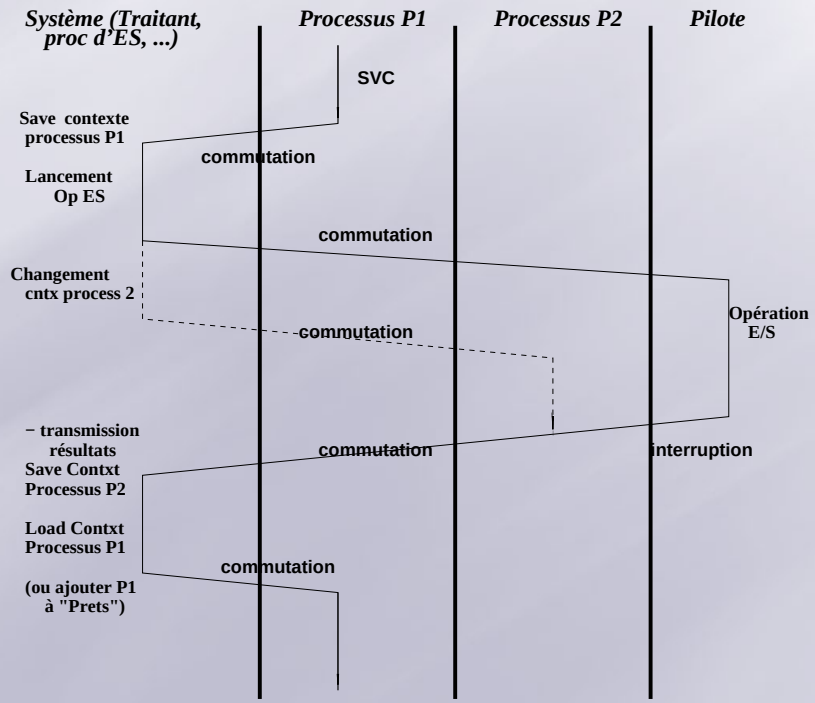
Noyau & Gestion du matériel : exemple d'E/S (suite)

Exemple : détails d'une demande d'impression :

- Le SE sous-traite à l'un de ses utilitaires (e.g. CUPS, LPR, ...)
  - ➛ Processus ou Démon ! (Le demandeur peut être mis en attente)
  - ➛ Paramètre du lancement : Printer + le fic. à imprimer
- LPR prépare une suite de *mots* (selon le type de la liaison)
  - ➛ Conversion éventuelle selon le langage de l'imprimante
- Le Pilote interroge l'état de l'imprimante (par réseau ou par câble)
  - ➛ Le pilote (driver) connaît les détails matériels
  - ➛ États : *prêt, libre, occupé, papier, cartouche, bourrage, fin,* ...
  - ➛ Le processus (e.g. LPR) n'a pas besoin de connaître les détails matériels, il s'adresse au Pilote.
  - ➛ Mais le Pilote et LPR exécutent le même protocole.
- Si tout va bien, commence un série de communications :
  - ➛ Envoi d'un mot (ou 1 *bit* si liaison série, un *paquet* si réseau, ...)
  - ➛ Attente de AR (ou un code d'erreur); peut être asynchrone
  - ➛ Refaire
- A la fin (ou en cas de pb) : Pilote ➛ LPR ➛ SE ➛ Demandeur. ../..

Noyau & Gestion du matériel : exemple d'E/S (suite)

- Pour clarifier et simplifier les développements, les E/S sont virtualisées (ou **synthétiques**)
  - ➙ Ex. : abstractions utilisant les primitives open, close, read et write pour manipuler toutes sortes de périphériques (schéma avec *recouvrement*).



TabMat