

# Introduction au TR, Embarqué la Robotique

Alexander Saidi  
ECL- LIRIS  
09-10

Novembre 2009

# Avant propos

## Petit Plan

- 2 séances où les sujets principaux abordés seront :
  - ◆ Introduction aux Systèmes d'Exploitation
  - ◆ Phases d'une compilation et la compilation croisée
  - ◆ Introduction aux Noyaux (Kernel) des Systèmes d'Exploitation
  - ◆ Embarqué et la Robotique
  - ◆ Stratégies de contrôle d'un Robot : un exemple concret
- Dans ce document, les sections signalées par (\*) peuvent être lues dans un 2e temps (contraintes de temps de présentation).

# Avant propos

## **Robotique : un domaine multi disciplinaires**

- Informatique (Algo, IA, IHM, ...)
- Électronique
- Mécanique

## **Ce par quoi le TR embarqué est concerné :**

- Système d'exploitation Embarqué :
  - Les éléments fondamentaux d'un SE auxquels l'embarqué est sensible.
    - ↳ les kernels (noyaux)
  - Système Temps Réel
- Robotique
  - Programmation Robot & stratégies : un exemple
- Compilation Croisée (notions de compilation),
  - Schéma d'un code exécutable

# Compilation Croisée

- Un compilateur croisé (*cross compiler*) est un programme capable de traduire un code source en code objet ayant un environnement d'exécution (architecture matérielle, système d'exploitation) différent de celui où la compilation est effectuée.
- Ces compilateurs sont principalement utilisés en informatique industrielle.

Un Exemple : la chine (ou un autre pays) fabrique tel modèle de voiture pour l'angleterre : les chinois roulent à droite (comme nous) et n'achètent pas ce modèle (qui convient aux anglais).

- De même que ces véhicules ne rouleront même pas en Chine, Le code généré par un compilateur croisé ne tournera pas in situ (sauf si les architectures sont compatibles).

# Compilation Croisée (suite)

- Les plateformes simples ont un schéma d'exécution simple.  
Par exemple, au démarrage, le compteur ordinal va à une adresse (A200) et exécute le code qui s'y trouve.
- Un exemple simple :  $N = N + 1$
- Le code Assembleur généré (pour un processeur embarqué basique) :

<i>A200</i>	Load	<b>A212</b>	Accu=N
<i>A204</i>	Add	#1	Accu=Accu+1
<i>A208</i>	Store	<b>A212</b>	N=Accu
<b>A212</b>	data		emplacement de N

N.B. : chacune des instruction est présentée sur 4 octes (puis décodée et exécutée)

- A props des différentes phases de compilation.

# Noyau

Un noyau de système d'exploitation (*kernel*) est la partie **fondamentale** de certains systèmes d'exploitation.

- Gère les ressources de l'ordinateur
- Permet aux différents composants matériels et logiciels de communiquer entre eux.
- En tant que partie du système d'exploitation, le noyau fournit des mécanismes d'abstraction du matériel
  - la mémoire,
  - processeur(s)
  - échanges d'informations entre logiciels et périphériques (matériels).
- Le noyau autorise aussi diverses **abstractions logicielles**
- Il facilite la **communication entre les processus**.
  - Bref : c'est le *coeur* du système d'exploitation.
- La séance précédente a donné un aperçu du noyau d'un SE général.

# Noyau (suite)

- Le noyau d'un système d'exploitation est lui-même un **logiciel**
- Son rôle central impose des **performances élevées**.
- Le noyau est la partie la plus **critique** d'un système d'exploitation
- Sa conception et sa programmation sont particulièrement délicates.
- Le noyau d'un système d'exploitation = un logiciel qui assure :
  - la communication entre les logiciels et le matériel
  - la gestion des divers logiciels (tâches) d'une machine
- lancement des programmes, ordonnancement, ...
  - la gestion du matériel (mémoire, processeur, périphes, stockage, ...).
  - assurer la sécurité de fonctionnement
  - compromis entre performance, sécurité et d'architecture des processeurs.

# Noyau (suite)

- Un noyau impose une partition virtuelle de la RAM physique en deux régions disjointes :
  - l'une étant réservée au noyau (l'espace noyau)
  - l'autre aux applications (l'espace utilisateur).
    - ➡ cf. les OS actuels : Linux, Windows, Mac OS X, etc.
- Le noyau est **prioritaire** sur l'utilisation des ressources matérielles, en particulier de la mémoire.
- Elle 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**.
- Le noyau offre ses fonctionnalités via des appels système (**SVC**).
- Il transmet ou interprète les informations du matériel via des **interruptions**
  - cf. les **entrées et sorties**.

# Processus et Noyau

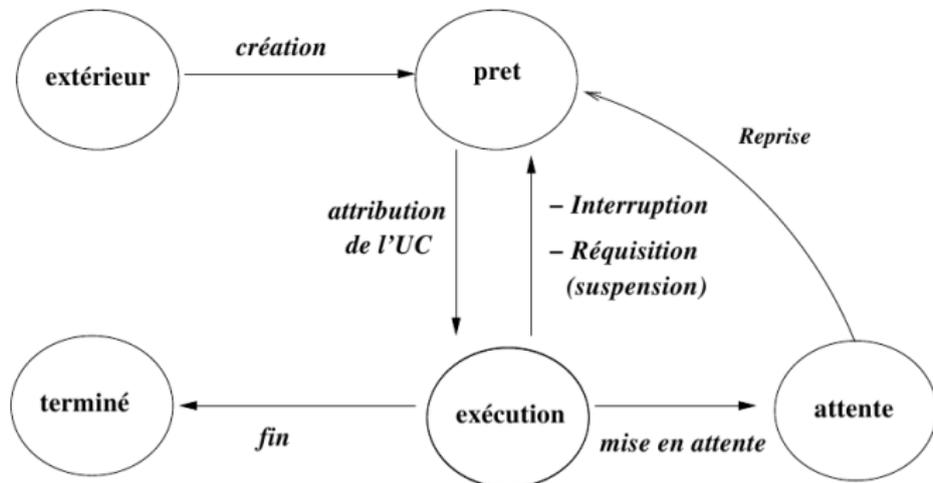
- Une abstraction principale : **processus** (ou tâche).
- Le noyau n'est pas en lui-même une tâche
  - Mais un ensemble de routines pouvant être appelées par les différents autres processus (exécution avec un certain niveau de privilèges).

ligne 1cm

- Un noyau gère des flots d'exécution des processus utilisateurs bloqués lorsqu'ils effectuent des appels systèmes.
- Un processus bloqué ne consomme pas de temps processeur,
  - il est réveillé par le processus système lorsque celui-ci se termine.
- Un processeur est capable d'exécuter un seul processus à la fois à l'instant  $t$ ,
  - un multiprocesseur est capable de gérer autant de processus qu'il a de processeurs

# Processus et Noyau (suite)

Rappel Diagramme (simplifié) d'états :



## Processus et Noyau (suite)

- Les noyaux multitâches permettent l'exécution de plusieurs processus sur un processeur,
  - ➔ en **partageant** le temps du processeur entre les processus.
- Pour une exécution (pseudo) parallèle, un noyau multitâche s'appuie sur les notions de :
  - commutation de contexte ;
  - ordonnancement ;
  - temps partagé.
- Pour les E/S : un traitement spécifique par l'ordonnanceur (attente, ...)

# Systèmes à noyau restreint

- Il existe des noyau (aux fonctionnalités restreintes)
  - **micro-noyaux**,
  - les systèmes **sans noyau** (MS-DOS, CP/M)
  - les **exo-noyaux**.
- Certains sont adaptés à des applications très ciblées
- La plupart d'entre eux est actuellement inadaptée pour une utilisation généraliste, dans des serveurs ou ordinateurs personnels.
  - Des variantes et des adaptations existent ../..

# Micro-Noyau

- On trouve dans les noyaux les fonctions des **micro-noyaux** :
  - un gestionnaire de mémoire
  - des ordonnanceurs,
  - des fonctions de communication inter-processus.
- Certains fournissent des fonctions moins fondamentales :
  - la gestion des systèmes de fichiers ;
  - plusieurs ordonnanceurs spécialisés (batch, temps réel, entrées/sorties, etc.) ;
  - des notions de processus étendues (eg. processus légers) ;
  - des supports réseaux (TCP/IP, PPP, pare-feu, etc.) ;
  - des services réseau (NFS, etc.).

## Micro-Noyau (suite)

- La plupart des noyaux fournissent également des pilotes pour le matériel.
- Un (micro) noyau peut fournir très peu de services.
- Par exemple , les fonctionnalités suivantes peuvent être implantées tant dans l'espace utilisateur que dans le noyau lui-même.
  - les pilotes matériels,
  - les fonctions réseaux et
  - systèmes de fichiers
  - ou les services
- Leur implantation à l'intérieur du noyau est faite dans **l'unique but** d'augmenter les performances.

## Micro-Noyau (suite)

- Selon la conception du noyau, la même fonction appelée depuis l'espace user ou l'espace noyau a un coût temporel très différent.
- Si cet appel de fonction est fréquent, on peut l'intégrer au noyau (performances).
- Ces techniques sont utilisées pour pallier des défauts des noyaux tels que les **latences** élevées.
- **Autant que possible**, il est préférable d'écrire un logiciel hors du noyau, dans l'espace utilisateur (perf. vs. sécurité).
- L'écriture en espace noyau suppose l'absence de mécanismes tels que la **protection de la mémoire**.
  - ➡ Il est donc plus complexe d'écrire un logiciel fonctionnant dans l'espace noyau que dans l'espace utilisateur, les bogues et failles de sécurité sont bien plus **dangereux**.

# Ordonnanceur

- Notion très liée au système multitâche.
- Son rôle : faire exécuter les tâches : save et **commutation** de contextes.
- 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.
- L'algorithme d'ordonnancement tient compte des priorités (1 ou n processeurs)
  - Dans le but d'utiliser efficacement les ressources de la machine.

# Ordonnanceur (suite)

## Types d'ordonnements :

- **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.

## Ordonnanceur (suite)

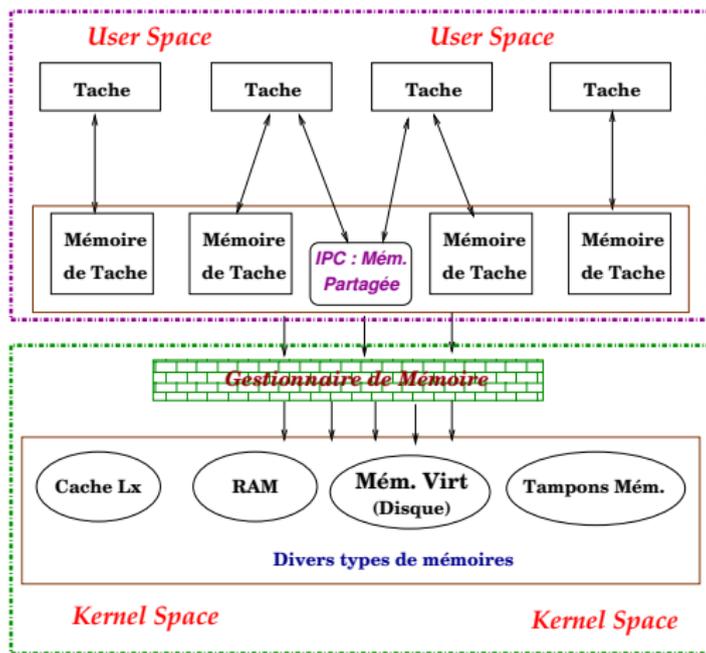


FIGURE: Différents espaces et le MMU

# Gestionnaire de mémoire

- Un sous-ensemble du système d'exploitation
- 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
- Tout processus doit manipuler une mémoire qui a les propriétés suivantes :
  - peut être étendue jusqu'aux capacités théoriques de la machine ;
  - la mémoire est privée (protégée),
  - 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 / tuper le processus fautif.

# Appels systèmes

- Les appels système 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 dans le programme appelant dans l'espace utilisateur.
  
- En plus d'un changement de mode d'exécution, l'appel système provoque au moins deux commutations de contextes :
  1. Contexte du processus appelant ;
  2. Contexte du noyau ;
  3. Voire, contexte du processus de reprise (pas forcément le premier).

# Appels systèmes (suite)

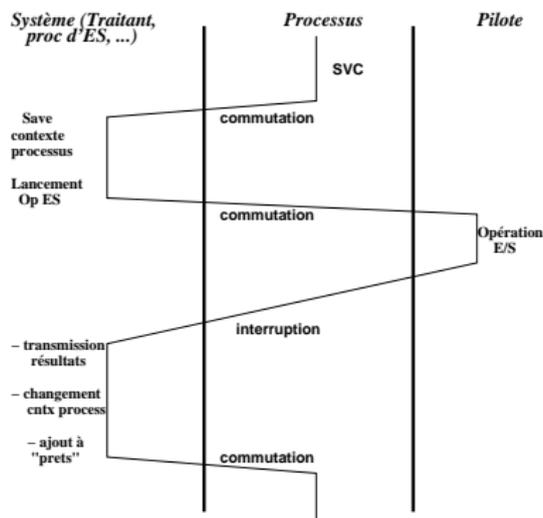
- Le coût d'un appel système est nettement plus élevé qu'un simple appel de fonction intra-processus :
  - un appel simple de fonction = quelques instructions primitives (chargement et exécution d'une zone mémoire),
  - le coût d'un appel système se compte en milliers ou dizaines de milliers d'instructions primitives, générant à la fois une charge et des délais d'exécution supplémentaires.
- Un SVC entraîne de nombreuses interactions de bas niveau (dans l'espace noyau), en particulier pour les **pilotes** de périphériques, pour les E/S plus généralement.
- Pour ces raisons, les fonctions qui sont utilisées de manière **intense** sont déplacées dans l'espace noyau.
- Les programmes utilisateurs font alors un nombre restreint d'appels système de haut niveau.

# 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).
- Il peut y avoir un très grand nombre d'accès à certains matériels (disques durs par exemple),
- 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).
- Mais Certains périphériques lents (appareils photographiques numériques, outils sur liaison série, etc.), sont/peuvent être pilotés depuis l'espace utilisateur, le noyau intervenant au minimum.

## Gestion du matériel (suite)

- Pour clarifier et simplifier les développements, les E/S sont virtualisées (ou **synthétique**)
  - ➔ Ex. UNIX, l'abstraction utilisant les primitives open, close, read et write pour manipuler toutes sortes de périphériques.



N.B. : Ce schéma ne traite pas le recouvrement.

# Différents types de noyaux

- Il existe toutes sortes de noyaux, plus ou moins spécialisés.
  - **spécifiques** à une architecture, souvent monotâches,
  - **généralistes** et souvent multitâches et multiutilisateurs.
- L'ensemble de ces noyaux peut être divisé en deux approches opposées d'architectures logicielles :
  - les noyaux **monolithiques** et les **micro-noyaux** .
- En général, les noyaux **monolithiques**, de conception ancienne, sont obsolètes et moins "propres" (difficiles à maintenir ).
- **Ex : noyau Linux** (sauvé par sa modularité)
- Dans un **micro-noyaux**, il faut **déplacer** l'essentiel des fonctions du noyau vers l'espace user (intéressante en théorie mais difficile en pratique).
- Les deux approches d'architectures de noyaux (micro-noyaux et les noyaux monolithiques) diamétralement différentes en termes de conception, se rejoignent en pratique
  - micro-noyaux hybrides et les noyaux monolithiques modulaires.

# Noyaux monolithiques non modulaires

- Certains systèmes d'exploitation, comme d'**anciennes** versions de Linux, certains **BSD** ou certains vieux Unix ont un noyau monolithique.
  - C'est-à-dire que **l'ensemble des fonctions du système et des pilotes** sont **regroupés** dans un seul bloc de code binaire.

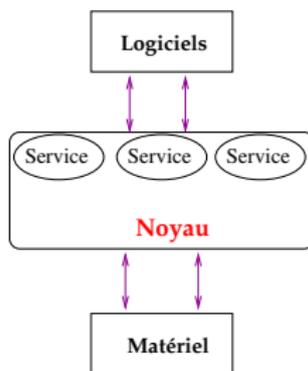


FIGURE: Architecture Noyau monolithique simplifié

# Noyaux monolithiques non modulaires (suite)

- Simplicité de concept, excellente vitesse d'exécution :
  - ➔ les noyaux monolithiques ont été les premiers à être développés.
- Mais avec le temps, très difficile à évoluer/maintenir.

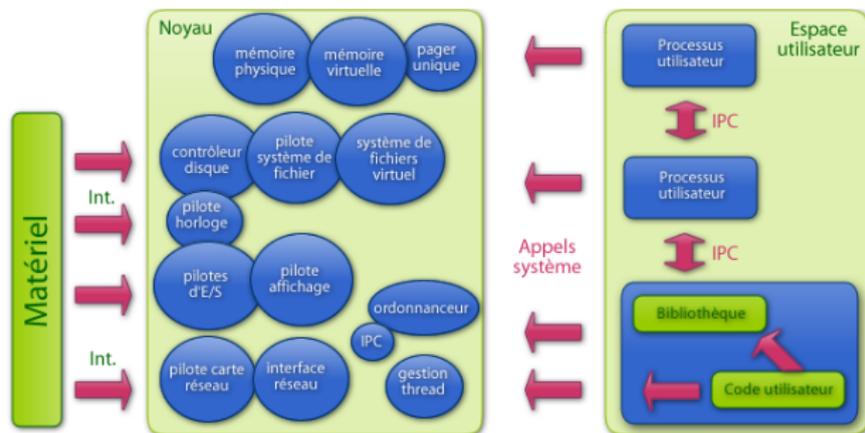


FIGURE: Architecture Noyau monolithique

# Noyaux monolithiques modulaires

- Réponse aux problèmes des monolithiques
  - ⇒ monolithiques **modulaires**.
  - seules les parties fondamentales du système sont regroupées dans un bloc de code unique (monolithique).
  - Les autres fonctions (e.g. les pilotes matériel), sont regroupées modules
  - séparés du code et du binaire du noyau.
- Utilisé dans un grand nbr de SE : Linux, des BSD ou Solaris.
- Exemple Linux : chargement à chaud des modules (pilotes).
  - augmente les possibilités de configuration.
  - Des systèmes de fichiers peuvent être chargés de manière indépendante,
  - un pilote de périphérique changé, etc.

# Systèmes à micro-noyaux

- Les limitations des noyaux monolithiques → les systèmes à **micro-noyaux**.

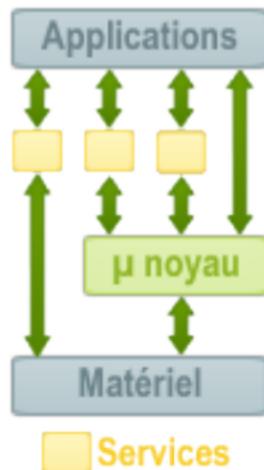


FIGURE: Micro Noyau simplifié



# Systèmes à micro-noyaux (suite)

## Bilan :

- + Eloignent les services "à risque" des parties critiques (le noyau minimal)
  - + permet de gagner en robustesse et en fiabilité,
  - + tout en facilitant la maintenance et l'évolutivité.
- Les mécanismes de communication (IPC) deviennent fondamentaux pour assurer le passage de messages entre les serveurs

## Avantages et inconvénients d'un système à micro-noyau

- Utilisation du mode **protégé** par les services qui accompagnent le micro-noyau.
  - ↳ les services (dans l'espace utilisateur) bénéficient de la **protection de la mémoire**.
- Indépendance des services = stabilité de l'ensemble
- Un des premiers micro-noyaux : **Mach**

# Systèmes à micro-noyaux (suite)

## Quelques inconvénients :

- L'utilisation de nombreux services dans l'espace utilisateur pose deux problèmes :
  1. La plupart des services sont à l'extérieur du noyau et génèrent un très grand nombre de SVC
  2. Les interfaces de communication entre les services (IPC) sont complexes et trop lourdes en temps de traitement.
- Néanmoins, le fait de placer les services en espace utilisateur (Ex. **L4**) :
  - Rapidité du traitement des IPC en simplifiant les communications au maximum
  - suppression de toute vérification des permissions (faites par les serveurs externes).

## Exemple d'associations micro noyaux - noyaux enrichis - SE

- ◆ A base du micro-noyau enrichi HURD : L4 et GNU Mach.
- ◆ A base du micro-noyau enrichi XNU : Darwin et MacOS

# Noyaux hybrides

**Noyaux hybrides** : combinaison des concepts des noyaux monolithiques + micro-noyaux (début 90s)

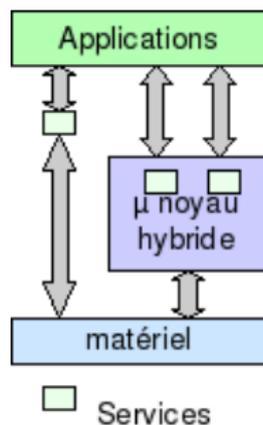


FIGURE: Noyau Hybride simplifié

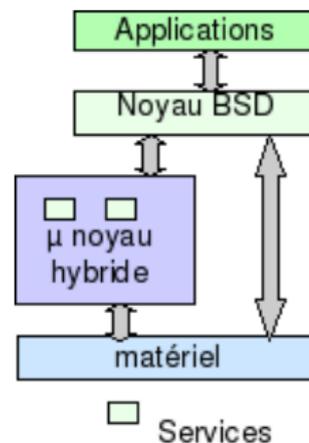


FIGURE: Noyau Hybride simplifié de XNU

## Noyaux hybrides (suite)

- Des Micro-noyaux : seules les fonctions fondamentales sont dans l'espace noyau +  
certaines fonctions non critiques mais gourmandes en SVC ont réintégrées l'espace noyau.
- Ce compromis permet d'améliorer les performances en conservant des propriétés des systèmes à micro-noyaux.
- Exemple : le noyau XNU de Mac OS X.
  - ➔ est basé sur le micro-noyau **Mach 3.0**
  - ➔ mais inclut du code du noyau monolithique **BSD** dans l'espace noyau.

N.B. : On appelle aussi *Noyau Hybride* d'autres types de noyaux :

- ➔ Les noyaux monolithiques sur micro-noyaux (TR ou non)
- Quelques exemples :
  - ➔ MkLinux (le noyau Linux sur Mach), Adeos,
  - ➔ **RTLinux** et RTAI.

# Méta-noyaux

- **Méta-noyau** : un ensemble de logiciels qui vise à appliquer la notion de noyau informatique au niveau d'un **réseau informatique**,
  - ➔ en créant une unique couche de gestion des périphériques au niveau d'un réseau.
- De cette manière, les logiciels peuvent être déployés et utilisés sur le réseau informatique comme s'il s'agissait d'une machine unique,
- L'ensemble des logiciels sur cette plate-forme se partagent les ressources de manière intégrée (comme sur un noyau simple).
- Un méta système doit permettre
  - ➔ la personnalisation,
  - ➔ la gestion des permissions
  - ➔ l'utilisation d'informations dépendant de la localisation.
- Cette notion rejoint les notions de **grappe de calcul**, de machine virtuelle, de serveur d'application et de CORBA.

# Noyaux temps réel

- Les noyaux temps réel sont fonctionnellement spécialisés.
- Des noyaux assez légers qui ont pour fonction de base stricte de garantir les temps d'exécution des tâches.

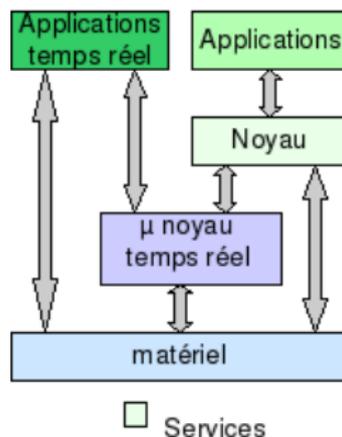


FIGURE: Micro Noyau TR hybride simplifié

## Noyaux temps réel (suite)

- Ce n'est pas une question de rapidité de traitement
  - ➔ La garantie des temps d'exécution en comparaison aux critères temporels de l'application industrielle
  - ➔ 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 l'**électronique embarquée**
  - ➔ conçus pour tourner sur des plates-formes matérielles limitées en taille, puissance ou autonomie.

### Exemples :

- Les noyaux TR peuvent adopter (en théorie) n'importe quelle architecture (ci-dessus)
- **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).

# Noyaux temps réel (suite)

- 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.

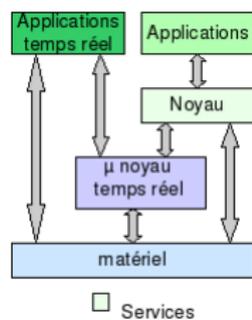


FIGURE: Micro Noyau TR hybride simplifié

- 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.

# Systèmes Embarqués

- De nos jours, les processus automatiques ont remplacé ce qui était fait à la main :
  - ◆ Processus : surveillance, tourner manivelles, tirer des cordes, chaîne et tapis de production, trier (écarter, séparer) les produits qui n'ont pas le bon calibre/couleur, ...
- On a maintenant des systèmes mécatroniques (vannes automatiques, réflexe auto, moteurs et pompes, robots, ...)
- Dans ces systèmes, l'Homme vérifie et fixe les consignes, met en marche/arrête, ...
- Ces systèmes limitent les risques d'accidents, réalisent des produits plus fiables, et précis, ..
  - ➔ Ils ne font pas grève! (Point de vue humain?)
- L'informatique est devenue **diffuse** (*ubiquitous computing/hardware*).

# Systèmes Embarqués (suite)

- Nous sommes envahis par les **Systèmes Embarqués** :
  - ◆ radio-réveil matin, machine à café programmable, 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 système embarqué ne possède pas d'entrée-sortie "std" (e.g. un vrai clavier/écran comme un ordinateur).

# Caractéristiques principales d'un système embarqué

- Est principalement numérique ;
- Met en oeuvre (généralement) un processeur (ou une forme simplifiée : DSP, PIC, FPGA ...).
- Exécute une application dédiée pour réaliser une tâche précise :
- ▶ N'embarque pas d'application scientifique ou grand public traditionnelle.
- Le clavier (une matrice) et l'écran (petit LCD) sont limités, voire inexistants ;
- Les architectures matérielles dédiées (ARM, X86, FreeScale, ...)
- Basse consommation, moins véloce qu'un PC std.
- L'**IHM** peut être très simple (cf. LED clignotant), ou bien très compliquée (cf. la cabine de pilotage d'avion).
- Des circuits numériques ou analogiques assurent (améliorent) les performances.

# Embarquée : un marché énorme et croissant

- **Ventes en 1999** : 1,3 milliards de processeurs 4 bits ; 1,4 milliards 8 bits, 375 millions 16 bits, 127 millions 32 bits et 3,2 millions 64 bits.
- **En 2004** : 14 milliards de processeurs de tous types (DSP compris)
- On compte une croissance d'au moins 6% par an sur ce marché.
- Seul 2% de tous processeurs est destiné au PC grand public ; le reste pour l'embarqué !
- Sur les 98% pour l'embarqué, 60% portent un OS embarqué (VxWorks, QNX, ...).
- Pour le reste, ce sont des systèmes "faits maison" très spécifiques :
  - souvent des adaptations de SE libre (e.g. UcLinux) pour l'embarqué
- **10%** des autres processeurs sont des 32 bits qui génèrent **31%** du chiffre d'affaire de cette tranche du marché (estimé à 48% pour 2008 ).

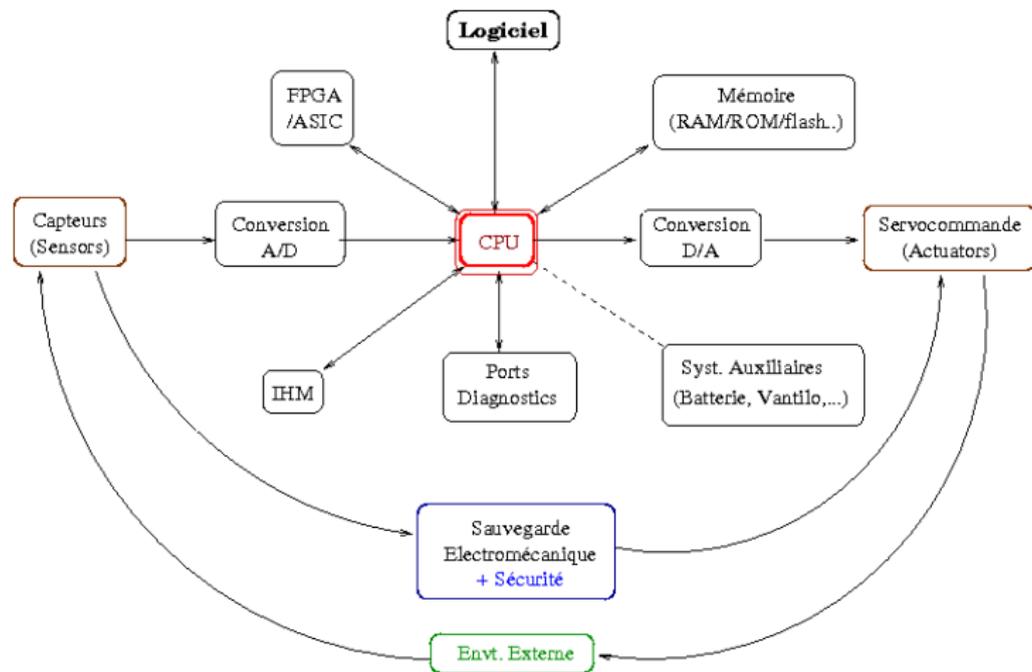
# Embarquée : un marché énorme et croissant (suite)

- On va migrer de plus en plus vers les processeurs 32 bits ; pour le moment un peu *forts* pour les systèmes embarqués
  - ➔ mais nécessaire si l'on fait tourner des systèmes tels que Linux embarqué évolué.
- Le prix moyen d'un processeur embarqué est de 6\$ contre 300 pour un processeur PC. standard.
  - ➔ Le marché de processeur PC est faible mais très lucratif!
- CA 2008 des processeurs famille X86 (Intel/AMD) seulement : 22 milliards d'euros (dont 30 % en Europe).

# Les domaines de l'embarqué

- **Jeux et calcul** (en général)
  - empaqueté dans un système embarqué (jeux vidéo, set top box, ...)
- **Contrôle système** : automobile, process chimique, nucléaire, systèmes de navigation
- **Traitement du signal** : radar, sonar, compression vidéo, Reconnaissances ...
- **Communication et réseaux** : téléphonie, internet, transmission d'information et commutation,
- ...

# Un système embarqué typique



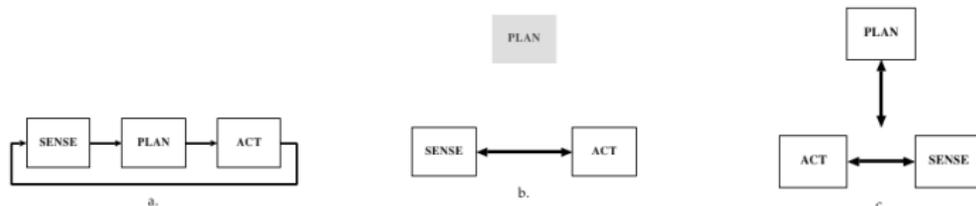
# Un système embarqué typique (suite)

- On trouve souvent en entrée des capteurs + convertisseurs A/D.
- En sortie, des actionneurs souvent analogiques couplés aux convertisseurs D/A.
- Souvent, un circuit FPGA (joue le rôle de co-processeur) épaulé le processeur.
  - ➔ FPGA : field programmable gate array (IC programmable "sur le champ, pour le prototypage d'IC industriel).
  - ➔ Dans un TP!, on remplace les capteurs en entrée par des interrupteurs, et la sortie par des LEDs....!
- CPU ou PIC? voir plus loin.
- Ce schéma général représente plusieurs paradigmes S P A

# Aperçu du paradigmes SPA

- Schéma de système embarqué général à tout automatisme dynamique :

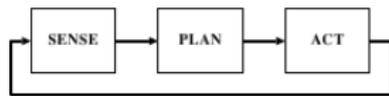
SPA : Sense, Plan, Act



- a) **SPA** : Hiérarchique
- b) **SA** : réactif
- c) **P,S-A** : Hybride délibératif/réactif.

## Aperçu du paradigmes SPA (suite)

- Le schéma Hiérarchique : de moins en moins utilisé (pb. Hyp. *monde fermé*, ...).



a.

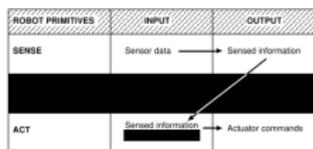
- Une autre vue du paradigme **SPA** Hiérarchique (Sense, Plan, Act)

ROBOT PRIMITIVES	INPUT	OUTPUT
SENSE	Sensor data	Sensed information
PLAN	Information (sensed and/or cognitive)	Directives
ACT	Directives	Actuator commands

- Lourd, gourmand en ressources et en calcul, rigide, ...

## Aperçu du paradigme SPA (suite)

- Le schéma *Réactif* (*Stimuli-Réponse*) : en vogue (surtout depuis 1988) grâce aux progrès en IA (*biologie* et *psychologie cognitive*, monde des insectes, etc),
- Rejet de la phase *plan* (P) direct ;
- Multiple schémas S-A, sous forme de processus concurrents et indépendants appelés **Comportements** (*Behaviors*) possiblement combinés (e.g. *avancer + tourner 45°*),...
- Une autre vue du paradigme **S-A** Réactif (Sense, Act)



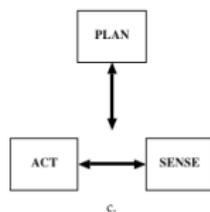
- Le schéma Réactif : rapide mais ne permet pas de modéliser un processus complexe en entier → schéma *Hybride* depuis 1992

## Aperçu du paradigmes SPA (suite)

- **Hybride** : dans la phase *Planification Délibérative*, le Robot cherche à planifier puis décomposer la *mission* en sous tâches
  - ➔ tâche = *Behavior* comme dans Réactif.
- Le schéma devient de fait **P,S-A** (figure (c))
- La composante S (capteurs, entrées) est aussi un mélange Hiérarchique-Réactif :
  - ➔ Ses données sont transmises à **A** et à **P** (pour une mise à jour du plan général).
  - ➔ Ex : la présence d'un obstacle concerne A et P (MAJ du map global)

# Aperçu du paradigmes SPA (suite)

- La Planification Délibérative (besoin calcul intensif) peut avoir lieu toutes les 5s alors que l'étape A réagit à S toutes les 20 ms.



ROBOT PRIMITIVES	INPUT	OUTPUT
PLAN	Information (sensed and/or cognitive)	Directives
SENSE-ACT (behaviors)	Sensor data	Actuator commands

Arrows in the table indicate data flow: from PLAN to SENSE-ACT, from SENSE-ACT to PLAN, from SENSE-ACT to INPUT, and from INPUT to OUTPUT.

# Environnement d'embarqué

- Contrairement à un PC (envt. protégé), un système embarqué peut évoluer dans un environnement hostile (voire agressif) :
  - Variations de température, vibrations et chocs, variation des alimentations, interférence RF, Corrosion, Eau, feu, radiations, ....
- On ne peut pas toujours contrôler l'environnement.
  - Parfois, il n'est même pas contrôlable !
  - Il faut en tenir compte dès la conception.
  - Par ex, on doit tenir compte des variations des caractéristiques électroniques des composantes en fonction de la température ou des radiations :
    - ce qui n'est pas le cas dans un PC.
- Les systèmes embarqués récents sont très communicants (possible via les processeurs puissants).

# Environnement d'embarqué (suite)

- La connectivité de ces systèmes permet le contrôle à distance des processus.
  - ➔ C'est la forme évoluée du contrôle par RS232, RS485, bus de terrain dédié.
- Souvent, un navigateur WEB permet ce contrôle (pas besoin d'IHM sophistiquée)
  - ➔ Ex. : les chauffagistes proposent de l'entretien à distance (par le WEB)!
- La communication WIFI : la norme IEEE 802.15 comme *Zigbee* pour l'embarqué,
  - ➔ en particulier en *Domotique* (couplé à un réseau de capteurs sans fil).
- Attention à la sécurité des systèmes (puisque connectés à l'Internet) si l'on veut que notre Chaudière ne se plante pas!!

# Les Contraintes de temps dans un système embarqué

- Système Embarqué → système Temps Réel (souvent)  
→ **contraintes temporelles** .

## Dans un RTOS

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

- Après l'acquisition/traitement, on doit avoir le temps d'utiliser cette information avant qu'elle ne soit rafraîchie.
- On doit donc garantir un temps max d'exécution (pas un temps moyen).
- D'où la notion de *complexité* (temps, espace) dans les algorithmes (cf. exemple de tri) !

# Eléments de Conception d'Embarqué

- On aura tendance à surdimensionner (si méconnaissance des événements/it).
- Le concepteur est **Pluridisciplinaire** : info, électronique, réseaux, sécu.
  - En plus, savoir optimiser les coûts ....!
- Le 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, conso élect, coût, tps de dév  
(concurrence!)

# Systèmes Embarqués Libres

- Linux de plus en plus utilisé dans l'embarqué :
  - Libre et dispo, stable et efficace, pas de royalties, ouvert, diff distributions dispo, communauté rapide et réactive (en cas de besoin d'aide), connectivité IP std...
  - 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'accomode de l'absence de MMU (ex. *uClinux*)
- Attention MMU : gain de performance mais code à surveiller.

# Temps Réel

- Un système Temps Réel (**RTOS**) : le **coeur** d'un *Système Embarqué*

## Définition

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

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

*un OS est Temps Réel si l'information après acquisition et traitement reste encore pertinente.*

➡ Ce temps de prise en compte d'information (e.g. une It°) doit rester inférieur à la période de rafraîchissement de l'information.

# Temps Réel (suite)

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

**Exemple :** ► Un système de navigation d'un drone : 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 m<sup>3</sup> 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 un stimuli extérieur*.
  - Il peut être embarqué (dans une voiture, avion, Robot, ...)
  - Permet l'automatisation d'un processus

# 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 ; logique câblé éventuelle).
    - Le non respect de ces contraintes peut même entraîner 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)

- Les contraintes temporelles oriente le choix d'un processeurs.

## Choix d'un Processeur :

- 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 dizaine de microsecondes (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 de 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 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 tâche 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.

# Prévisibilité

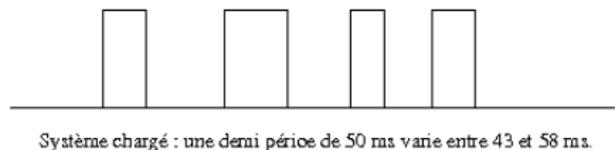
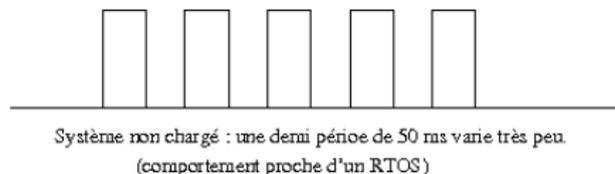
- Si l'on ne peut pas garantir les délais lorsqu'un système est chargé :

```
#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);
}
```

# 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  $\pm 0.1$ ).

# Fonctionnement d'un RTOS

## Rappel :

- **Dans un RTOS** : on doit pouvoir acquitter une interruption (**it**), traiter l'information et signaler 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!).
  - ➔ Stocker ? (Pooling)
- Cela dépend du processus extérieur qui impose ses Contraintes (et non le contraire).

# Approches en conception des RTOS

**1- Synchrones, Déterministes** : 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.

**3- Approche Objet** : plus récente.

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

**4- Approche Coopérative** : Peu utilisée (*Oberon*).



## Quelle est la forme de ces événements ?

- 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 **péremption**.

### Exemples d'interruptions (it) :

- Un capteur a changé d'état, une touche appuyée (opérateur), ...
- La gestion d'une horloge (de la 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, ...

# Mécanismes de Synchronisation

- Il existe plusieurs solutions pour la synchronisation.

▶ Solutions matérielles :

- ↳ Utilisation d'une instruction **insécable**
  - ↳ (ex. 68K : **Test And Set**)
- ↳ Verrouillage de bus
- ↳ Masquage et interruption
- ↳ **Sémaphores**

▶ Solutions logicielles :

- ↳ Code et algorithmes (compliqué)

N.B. : Parfois, les sémaphores sont considérés comme une solution logicielle bien qu'ils soient implantés par des solutions matérielles.

# Sémaphores

- A considérer comme un distributeur de tickets dans un parking avec un nombre limité de places.
  - ➡ Cas d'une place (une cabine téléphonique)
  - ➡ Cas de N places (un parking)

```
struct Semaphore
{ valeur : entier;
  attente : file d'attente;
}
```

```
Fonction V(Sem : Semaphore)
{Si Sem.val < 0 Alors Débloquer_suivant(Sem.attente);
 Finsi
 Sem.val ++;
}

Fonction P(Sem : Semaphore)
{Sem.val --;
 Si Sem.val < 0 Alors Attendre(Sem.attente);
 Finsi
}
```

# Sémaphores (suite)

- Un sémaphore **binaire** (0 ticket) peut servir à synchroniser deux tâches.
  - La valeur initial = 0
  - Tant que l'un n'aura pas produit un appel à V, l'autre ne passera pas.

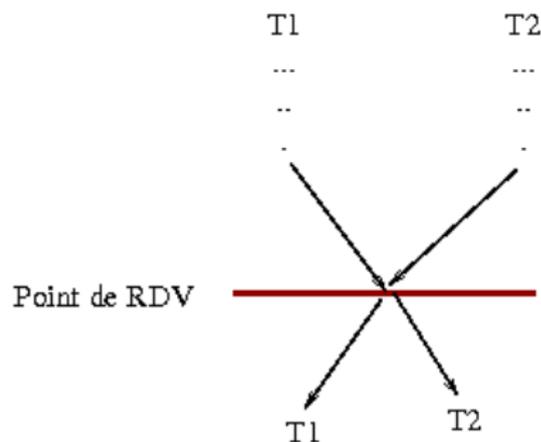
## Exemple code :

```
Bool First=True;
Semaphore S init 0;
```

### Code RDV de $T_i$ :

```
if (First)
  First=False; P(S);
else
  First=True; V(S);
```

➔ Protection de *First*?



# Sémaphores pour partage de ressources

- Une ressource peut être **partageable** ou non, à accès **simultané** possible,...
- **Exemples** de ressources ne pouvant être utilisées simultanément
  - 1 l'accès à une imprimante (non gérée par une tâche *IMP\_manager*),
  - 2 l'accès à un driver matériel,
  - 3 l'accès à une variable commune ou autres mécanismes (ressource)
- **Solutions** :
  - ➔ Opération insécable (TAS, l'écriture d'un booléen) et **exclusion mutuelle**

# Gestion d'accès en exclusion mutuelle

- **Mutex** (sémaphore d'exclusion mutuelle) avec une valeur init = 1.
  - Avant d'accéder à la ressource, la tâche prend un ticket qu'elle rendra à la fin.
  - Les attentes doivent être courtes pour permettre de satisfaire les contraintes temporelles.
- N.B. : la solution avec TAS est plus rapide (si un seul ticket suffit)
- **TAS** : instruction élémentaire (insécable, atomique, exécutée en 1 cycle) pour lire et écrire un mot mémoire.
  - Deux opérandes : un registre  $R$  et un mot mémoire  $B$  .
  - On copie  $B$  dans  $R$  et on place 1 dans  $B$ .

```

Procédure TAS (var a,b : entier)
  Début          a := b;      b :=1;
  Fin TAS;
  
```

## Gestion d'accès en exclusion mutuelle (suite)

- **TAS** résout le problème de **Section Critique** (SC) de la manière suivante :
  - ↳ Les processus partagent une variable *Verrou*.
  - ↳ Chaque processus  $P_i$  possède une variable locale  $Test_i$ .
- Code de chaque processus  $P_i$  :

```

Testi : entier ;
Répéter
    <SR>
    TAS(testi, Verrou) ;
    Tantque Testi=1 faire
        TAS(testi, Verrou) ;
    finTq
    < SC >
    Verrou := 0 ;
Jusqu'à Faux ;

```

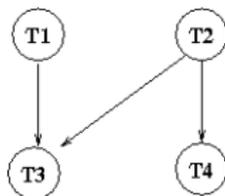
- Le premier processus qui exécute TAS trouve  $Verrou = 0$  et entre en SC.
- Les autres trouveront  $Verrou = 1$  et attendent.

# Gestion d'accès en exclusion mutuelle (suite)

- Cette solution garantie *l'exclusion mutuelle* mais pas l'attente bornée (assurée par d'autres moyens).
- N.B. : IBM360 était la première machine à proposer TAS (TST).  
Par la suite, Motorola a implanté TAS puis Intel iAPX86 a proposé XCHG qui échange le contenu d'un registre avec un mot mémoire.
  - Code de XCHG :
 
$$\begin{aligned} & \text{Reg} = 1; \quad \text{XCHG } \text{Reg}, \text{Verrou} \\ \Rightarrow & \text{Reg} := \text{Verrou} \text{ et } \text{Verrou} := \text{Reg} = 1 \quad \simeq \text{TAS.} \end{aligned}$$
- NB : TAS et XCHG réalisés avec verrouillage du bus.
- Les sémaphores sont utilisés dans différentes situations de synchronisation et d'exclusion.

# Exemple d'utilisation de sémaphores pour la synchronisation

- Soit le graphe de précédences suivant :



- Une solution qui minimise le nombre de sémaphores :

```

S : Sémaphore init 0;
parbegin
  begin T1 ; P(S) ; T3 ; end ;
  begin T2 ; V(S) ; T4 ; end ;
parend
  
```

- (Montrer) un exemple de partage de l'écran.  
 ➔ Si chacun écrit, les messages sont mélangés.

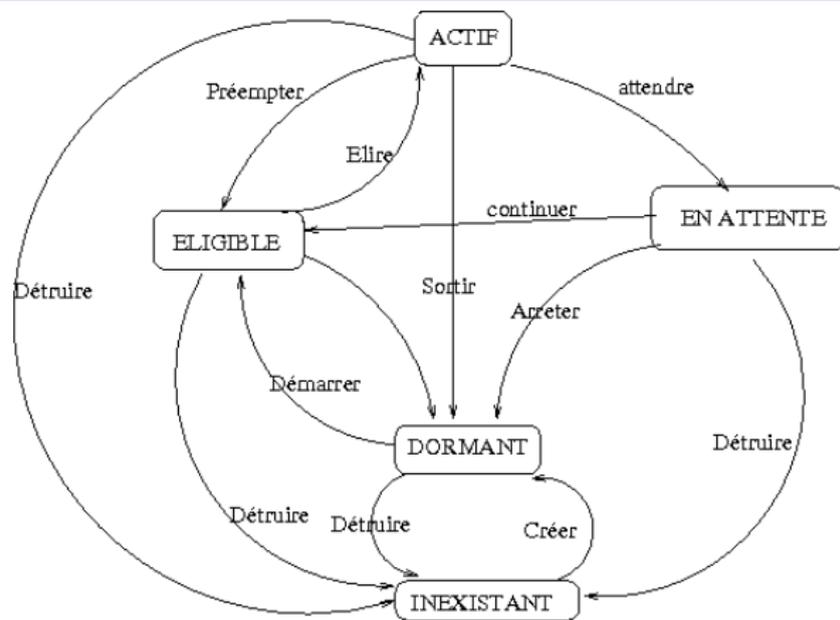
# Communication entre tâches

- Les tâches se synchronisent ; mais elles peuvent également communiquer.
  - Via un tampon (de taille 1 à N) = Boite aux lettres.
  - Synchro de R/W en E/S, file d'attente éventuelles...
  - Un expéditeur envoie des messages, un seul lecteur les lit (e.g. Spooler) en FIFO.
- Si la communication ne nécessite pas de tampon/file, on peut utiliser un drapeau (une var globale) protégée par un Sémaphore.
  - Certains RTOS (ou langages comme ADA ou Java) proposent des variables protégées, avec accès en exclusion mutuelle.
  - Exemples divers, mécanismes de passage de messages (peuvent servir à la synchro), Client-Serveur, ....

# Le rôle du noyau de RTOS

- Les mécanismes ci-dessus sont souvent intégrés à un RTOS : son noyau propose
  - des mécanismes de synchro, passage de messages, sections critiques, ...
  - gestion de temps, des It, et des ressources systèmes (RAM, Disk, KB, affichage, ...)
- Coté matérielle, beaucoup de RTOS utilisent une *couche matérielle abstraite* :
  - **HAL** (Hardware Abstraction Layer) permet de ne pas se préoccuper des spécificités matérielles de la plateforme (eg. la mémoire et sa gestion)
- Un RTOS assure surtout l'**ordonnancement des tâches** (selon des priorités).
  - Un ordonnanceur (dit aussi *scheduleur*) est un élément clef d'un RTOS.
  - détermine une politique et permet l'exécution des tâches.
- Les tâches reçoivent un état : *active, en attente, éligible, inexistant, ...*

# Le rôle du noyau de RTOS (suite)



- Le processeur est une ressource **non partageable** : géré par l'ordonnanceur.

# Le rôle du noyau de RTOS (suite)

## Taches : le principe commune à tout RTOS

- Une tâche : Inexistant  $\Rightarrow$  Créée  $\Rightarrow$  Dormant
  - ➔ Elle sera Démarrée par un code différent.
- Certains systèmes ajoutent un état de *suspension*.
- L'ordonnanceur choisit une tâche selon des règles et stratégies strictes mais diverses.
- **Exemples de stratégies** (orientée Embarqué) :
  - ➔ Si Contraintes fortes de temps  $\Rightarrow$  **EDF** (Earliest Deadline First).
  - ➔ **LLF** (Least Laxity First) : choix de la tâche ayant la plus petite *laxité* (i.e. au plus petit temps de marge entre la fin de la tâche et le Dealine).
- Mais en général, à ces stratégies *ad-hoc*, on préfère une gestion par **priorité** (qui peut tenir compte des stratégies ci-dessus).

# Gestion des Priorités (des tâches)

Stratégie plus homogène commune aux RTOS :

- La priorité : **fixe** ou **dynamiques** (change au cours de la vie d'une tâche).
  - ↳ Le **Rate Monotonic** est la stratégie à priorité fixe la plus utilisée.
- En général, on fixe les priorités dès la conception et inversement proportionnelle à la durée d'activation.
- Gestion des priorités dynamiques : files de priorités
- Ordonnanceur : **round robin**
- Il reste quand même des cas particuliers qu'il faut traiter/prévoir/débloquer.

## Gestion des Priorités (des tâches) (suite)

### Problème d'inversion des priorités :

- Un problème courant : inversion des priorités
- Une tâche de basse priorité peut bloquer une ressource alors qu'une tâche de grande priorité en a besoin et reste donc bloquée.  
Cela provoque une situation de blocage =  
le problème d'*inversion des priorités*  
    ➡ une tâche de forte priorité est soumise à l'exécution d'une autre de priorité basse.
- Arrive fréquemment (cf. une distribution de billes/pages RAM)
- Arrive aussi dans les systèmes embarqués à base de *Behavior*, avec accès concurrent à une ressource (Notion d'ensembles de sorties disjoints (des agents) (voir + loin))

# Gestion des Priorités (des tâches) (suite)

## Deux solutions à ce problème :

- 1- "forçage de priorité" (après un délai) :
  - ➡ on attribut une grande priorité à la tâche bloquante.
- 2- "Héritage de priorité" :
  - ➡ la plus forte priorité des taches bloquée est donnée à la tâche bloquante.
- Ces stratégies sont en général codés dans le RTOS (i.e. on évite de coder soi même la gestion de ressource/sémaphore et on s'en remet au RTOS.)
- De même , il faut utiliser les APIs proposés (au lieu des siens) pour accéder aux ports série/parallèle, .... Il faut aussi lire la doc!

## Question : quel RTOS utiliser ?

- Dans certains domaines, les normes imposées déterminent le choix
  - ➡ DO-178B pour l'aéronautique, OSEK pour les voitures.
- Pour le cas général, VxWorks est connu et utilisé.
  - ➡ Il y en a plein d'autres.

# Processus et threads

Cf. Courses. (../Hippique/chevaux-screen)

# Temps partagé & Temps Réel (sous Linux)

- Système **Temps Réel** vs. Système **Temps partagé**
- Temps partagé (comme Linux ou même 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 priorité pour un partage équitable (**quantum** de temps ou *tick*).
  - La commande *nice* (Unix) permet d'intervenir sur les priorités.
  - Les tâches doivent pouvoir accéder aux différentes ressources partagées.

# Temps partagé & Temps Réel (sous Linux) (suite)

- Le temps partagé (Un\*x, Windows) entraîne 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 I/O de manière non optimisée (pas grave!) : le temps de latence (temps entre la réception d'un I/O et son traitement) n'est pas garanti par l'OS.
  - L'utilisation de la mémoire virtuelle peut entraîner des fluctuations dans les temps d'exéc des tâches....
- **Bref : Il y a du chemin pour passer du Temps Partagé au Temps Réel.**

# Quelques RTOS Posix

- VxWorks et pSOS sur beaucoup de plateformes,
- QNX
- microC/OS : version enseignement gratuite
- Window CE!
- LynxOS
- Nucleus
- Ecos
- microLinux
- Et le projet temps réel dur XENOMAI 2 sous linux.
  - Permet de simuler plusieurs RTOS.

# Programmation Robot

- Progrès en Robotique sur 2 fronts
  - Matériel
    - ➡ Microcontrôleurs
    - ➡ Capteurs
  - Logiciel (soft)
    - ➡ Contrôle à base de réflex (**Behavior-based** : nécessite peu de ressources de calcul)
- Deux approches principales :
  - **centralisée** (classique)
  - **décentralisée**.
- Revoir les schémas SPA.

# Approche centralisée (planification)

- Toutes les données des capteurs arrivent au même centre de contrôle :  
 ➔ Beaucoup de calculs :



**FIGURE:** les modules fonctionnels transforment les données capteurs en une série d'actions.

- Le programme (soft) du robot est décomposé en 1 série de fonc.
- Les bruits/conflits dans les données capteurs sont résolus ;  
 ➤ un modèle (éventuellement incomplet) du monde réel est construit par le programmeur et par avance.
- La représentation du monde contient les détails géométriques des objets du monde du robot (avec leur position et leur orientation).
- **Le Robot reçoit un but, planifie ses actions pour atteindre le but.**  
 ➤ Le robot traduit ensuite son plan en une série d'actions en envoyant les commandes aux servomoteurs.
- Le robot peut utiliser les capteurs dans les actions :  
 ➔ Ex : *avancer le long de l'axe X jusqu'à l'observation de l'adversaire.*

## Approche centralisée (planification) (suite)

- HANDEY = un robot fonctionnant sur le principe modélisation / planification.

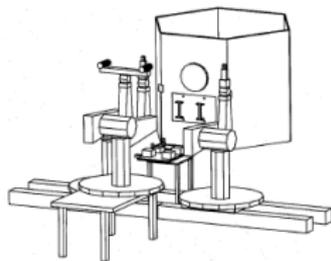


FIGURE: HANDEY (MIT) dans l'espace (manipulation par pince des satellites en orbite).

- Principe **piquer/placer** : modèle du monde + position d'une pièce et son orientation + position et orientation finale. ..//..

# Approche centralisée (planification) (suite)

## ... Handey (suite) ...

- Le robot utilise un scanner laser pour repérer les objets et les incorporer dans son modèle géométrique du monde.
- Crée un plan efficace pour piquer, se déplacer et poser une pièce (par une pince).
- La compatibilité de la pince avec les objets et positions est vérifiée
  - plusieurs actions sur les objets pour bien les appréhender par la pince (à 2 doigts).
- Handey peut disposer d'un robot **esclave** pour l'aider!
  - La coopération est possible entre les deux (l'un est chef!)
- **Une fois tout planifié**, le Robot envoie une série de commandes aux servomoteurs.
  - Puis vérifie les résultats (conformes au plan ?)

# Approche centralisée (planification) (suite)

## Avantages :

- La planification par avance permet l'optimisation et la correction (incohérence détectables).
  - On peut donc garantir des résultats.
- Sur un robot mobile, cela permet d'**éviter des impasses** avec un piège (trappe sans pouvoir **revenir en arrière!**).
- L'approche centralisée produit de meilleurs résultats (si ressources disponibles) que les approches décentralisées
  - vs. l'approche décentralisée : informations locales et partielles.

# Approche centralisée (planification) (suite)

## **Inconvénients** de l'approche centralisée :

- Temps de calcul, ressources nécessaires, Pb. de coût.
  - Inadaptation aux robots évoluant dans un environnement changeant.
  - Le plan embarqué doit être précis.
  - Nécessite aussi des capteurs très précis + une calibration précise.
- 
- Les capteurs sont peu précis, donnent des résultats parfois différents sur un même objet. La surface d'un objet peut perturber les mesures renvoyées.
    - Il faut multiplier le nombre de capteurs et traiter tout.
    - Besoin d'un codage de haut niveau (cf. Handey, en Lisp) + temps de calcul (processeur)
  
  - Les robots sont peu autonomes dans ce modèle.
    - Le monde alentour ne doit pas changer
      - surtout entre les captures d'info et l'action !.

# Approche décentralisée à base de réflexe (cybernétique)

- Plus récent ; Vient du MIT (R. Brooks, Labo IA)
- Idée : combiner le contrôle TR distribué avec le réflexe à base de capteur.



- Les réflexes (**Behavior**) sont les couches du **contrôle** du système.
  - ➔ Un module d'**arbitrage** permet de résoudre les conflits et mettre en oeuvre un scénario donné.

## Attention

- Une réaction (réflexe) ne contrôle pas (n'appelle pas) une autre.
- Les Réflexes sont exécutées en parallèle, mais un réflexe d'un niveau supérieur peut suspendre une d'un niveau inférieur.

# Approche décentralisée à base de réflexe (cybernétique) (suite)

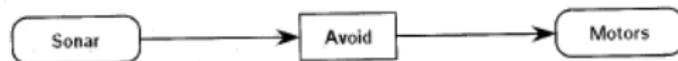
## Approche décentralisée réactive (suite) ..

➡ Si les Réflexes du niveau supérieur sont désactivés (par exemple, coupure liaison capteur), les réflexes du niveau inférieur reprennent la main.

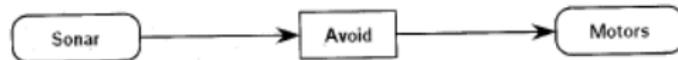
- Les capteurs interagissent à travers les Réflexes.
- Il n'y a pas de modèle global du monde réel contrôlant l'ensemble, ni de structure de données unifiée.

# R1 : un Robot simple

- Un robot muni d'un anneau de capteurs sonars (balayage à 360°), un détecteur IR, un processeur basique (simple) + un peu de RAM.
- On lui assigne le but : **éviter de se cogner dans les obstacles.**
- Un diagramme :



- **Sonar** est un module qui opère sur les capteurs sonars qui s'occupe en permanence de la distance mesurée.
- **Motor** est un module qui envoie du courant au moteurs en réponse de la commande qu'il reçoit.
- Entre les deux, le module **Avoid** s'appuie sur les données sonars et calcule en permanence des commandes envoyées au Motor.

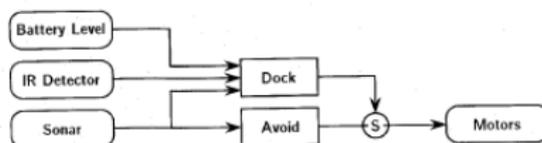


# R1 : un Robot simple (suite)

- **Le but** de ce robot est de rester à distance (loin) de tous les obstacles (repérables par les sonars)... par réflexe / réaction !
- Avoid contient un code qui implante un comportement (réflexe) simple et réflexif.
- Si la donnée Sonar de tête est trop "courte", Avoid arrête l'avancée du Robot.
- Si, hormis les mesures arrières (à contre sens), un capteur sonar mesure une distance la plus proche, Avoid fait tourner le Robot pour pointer dans cette direction.
- Quand le sonar arrière mesure la distance la plus proche, les moteurs reçoivent la commande d'avancer (de s'éloigner).
- Si les données sonars dépassent un certain seuil, *Avoid* ne fait rien (n'envoie pas de commande).
- L'ensemble de ces opérations (simples) est traitable par un petit MC68HC11A0 capable d'exécuter tout le code très rapidement (plusieurs fois par seconde).
- Le réflexe de **cette stratégie** permet au robot de rester à distance (loin) de tous les obstacles (repérables par les sonars).

## R2 : Recharge de batterie

- On ajoute un deuxième module **Dock** dont le propos est de mener le robot à une base de charge, quand les batteries du robot sont faibles.
- **Dock** récupère des données **sonars**, **IR** et **batterie**.

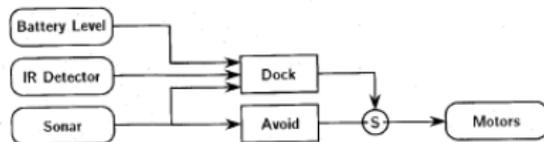


**FIGURE:** Diagramme à base de la réflexe du robot qui va en stand de charge

- **Le stand de charge** possède une balise IR. Quand active, le module Dock calcule les commandes pour mener le robot à ce stand tout en évitant les autres obstacles.
  - ➔ Avoid et Dock doivent "se parler" pour le contrôle du moteur.
- Le noeud (S) du diagramme (suppressor) permet le passage de messages venant du câble d'origine directement vers la sortie.

## R2 : Recharge de batterie (suite)

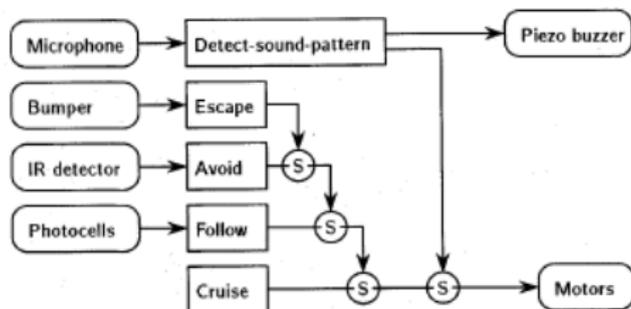
- Sur (S), un message arrivant de la liaison avec une flèche (Dock dominant) supprime (sans sauvegarde) celui venant du câble inférieur :
  - ➔ seul le message dominant va sur la sortie de (S). Si pas de message du câble dominant, celui du câble inférieur passe.



- **Tant que la batterie n'est pas faible**, le Robot évite les obstacles.
- Lorsque la batterie baisse, Dock s'active et conduit le Robot vers le stand, en répondant à la balise et en supprimant l'effet d'Avoid
  - ➔ Le Dock *subsume* Avoid et produit une compétence plus élevée.
- Un autre intérêt de cette approche réactive est la possibilité d'améliorer le système,
  - ➔ **sans perturber** la compétence précédente ni l'alourdir l'ensemble.

# R3 : un Robot plus évolué

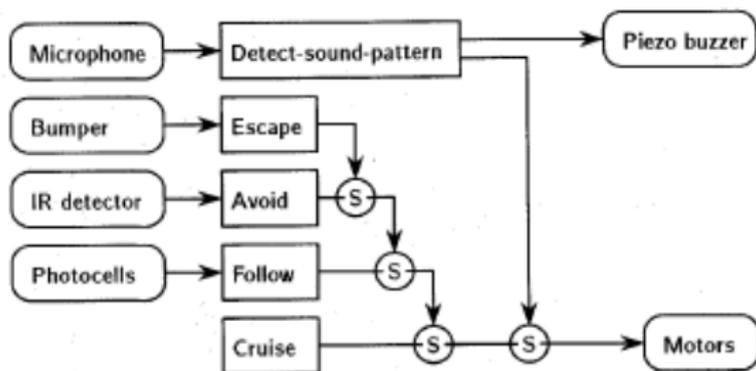
- On étend les compétences de R2 en lui assignant :
  - 3 capteurs de collision placés sur un anneau souple (un *rideau* de capteurs) ;
  - 2 capteurs infra rouge de proximité
  - 2 cellules photoélectriques
  - 1 microphone
- La figure suivante donne un aperçu des compétences de R3 :



## R3 : un Robot plus évolué (suite)

- Rappel : les réflexes s'exécutent en parallèle.
- Chaque module examine ses entrées en permanence et calcule une commande en sortie.
- Le module "Cruise" fait déplacer le Robot en permanence.
  - Ce module demande aux moteurs de **toujours avancer**.
- Le module "Follow" fait suivre une lampe (vers la lumière).
  - Lorsqu'il y a une différence d'intensité entre les deux cellules, il commande les moteurs de tourner vers la source la plus forte.
- Avoid évitera un obstacle détecté par les IR.
  - Lorsque Avoid détecte une obstruction sur le côté gauche du Robot, il commandera à tourner à droite, et inversement pour le côté droit.
  - Un obstacle en face cause Avoid de stopper R3 ; tourner de 90° à gauche ou à droite

## R3 : un Robot plus évolué (suite)



- On constate que "Avoid" prime sur "Follow" et "Cruise" lorsque les capteurs IR détectent une collision imminente.
- Le module "Escape" aide à éviter les obstacles si les IRs sont aveuglés par un obstacle.
- Les commandes depuis Follow et Cruise sont en concurrence mais c'est Follow qui l'emporte. Ce qui veut dire que R3 avance tout droit sauf si une source de lumière est à suivre.

## R3 : un Robot plus évolué (suite)

- Si jamais les capteurs IR ne fonctionnent pas (pour certains objets), le module "Escape" sera activé lorsqu'une collision est détectée (par les détecteurs de collision). Escape commandera alors le robot de s'éloigner de l'obstacle.
- La disposition des noeuds (S) place **Escape au niveau le plus élevé** : les cmds. de Escape sont les plus prios. en matière de déplacement du Robot.
- Enfin, le module "Detect-Sound-pattern" fait jouer un son à R3 lorsque l'on frappe dans les mains avec une certaine fréquence suivi d'une pause.
  - Une commande est alors envoyée à un buzzer puis une commande d'arrêt est envoyée aux moteurs.

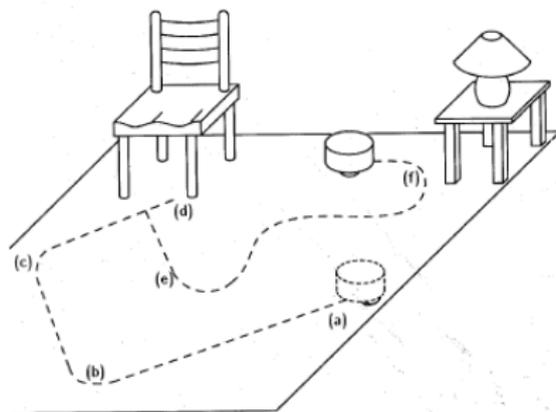
### **Effet et comportement global :**

- Le robot R3 avance d'abord rapidement, cherchant la source de lumière la plus forte.

Avancant vers la lumière, il évitera les obstacles mais s'il y a une collision avec un objet, R3 changera de direction et s'éloignera.

## R3 : un Robot plus évolué (suite)

Quand on frappera dans les mains d'une certaine manière, R3 s'arrête, joue un son puis se remet en route.



# Implantation

**Propos** : implanter plusieurs réflexes (comportements) en parallèle qui s'exécuteront sur des petits processeurs (PIC).

- **Technique Multi-tâche** : une boucle générale qui donne la main à tour de rôle aux réflexes.

- Donnera l'impression que les réflexes tournent simultanément.

- Quelques notions simplifiées nécessaires à reprendre :

- ➔ processus, scheduler et AEFs.

- Exemple : un robot qui émet périodiquement une lumière (LED) :

- La fonction *sleep(n)* permet faire une pause de *n* secondes.

- Le code sera de la forme :

```
void multi_flash()
{while (1) {                // boucle infinie
    flash_leds();          // faire clignoter
    sleep(1);              // pause 1 seconde
}
}
```

# Implantation (suite)

Un tel code dans un programme simple (sur un système simple) mobilisera **tout le processeur**.

- ➔ Solution : On enferme ce code dans un processus
- ➔ sera une entité parmi d'autres.W
- ➔ Son code mobilisera toutes les ressources de ce programme mais celui-ci sera un processus parmi d'autres.

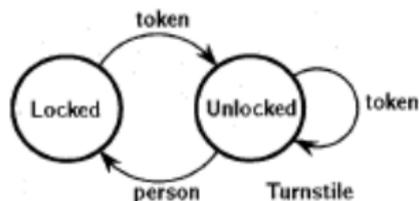
- **Plus intéressant** : le code de R3 peut disposer de plusieurs processus pour faire fonctionner le robot.
- Un processus est un programme (son code) indépendant qui fonctionne en parallèle avec d'autres processus (programmes).
- Un processus a un code, des ressources/fichiers/mémoire/....
- Pour faire tourner un ensemble de processus, on utilise un **scheduleur**.

# Implantation (suite)

- Le scheduler est un programme maître (partie du OS)
  - donnera à tour de rôle la main (l'UC) à un processus pour un certain temps (quelque millisecondes) selon une certaine stratégie d'élection.
- Pour reprendre la main (l'UC) d'un processus, un scheduler procède à une *préemption* (**multi tâche Préemptif**).
  - Une version plus simple : **multi tâche coopératif** (e.g. Oberon) :
    - ➔ Dans cette variante, c'est le processus qui décide de rendre l'UC (au scheduler s'il y en a, sinon, à un autre processus).
  - Mais alors, il faut retrouver où on était avant de donner la main au scheduler / à un autre processus
  - Brooks avait proposé d'assigner à chaque processus un MEF (machine d'état fini).

# Implantation (suite)

- En l'absence d'un scheduleur, on peut implanter un schéma de controle de reflexes par une MEF.
  - Même en présence d'un scheduleur, cela nous aide à bien comprendre les reflexes.
- Une MEF est une abstraction d'un calcul composée d'états.
  - La machine change d'état en présence d'un événement (une entrée).
- Un exemple : tourniquet

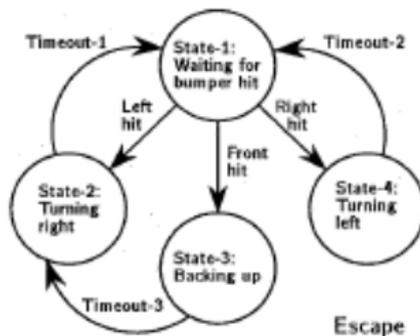


- On note  $\sigma : S \times \alpha \rightarrow S'$

# Formalisme de contrôle des réflexes

Coopératif (sans préemption)

- Exemple : quand R3 se cogne dans un objet → réflexe Escape.
  - Escape utilisera un capteur de collision.
- La figure suivante donne la MEF.



On constate : s'il y a une collision de face, on recule puis tourne à droite.

➤ Les time-out laissent la main aux moteurs pour tourner dans le sens indiqué.

# Formalisme de contrôle des réflexes (suite)

Coopératif (sans préemption)

- Le pseudo-code correspondant :

Escape

Outputs: (Motor-command)

State-1: If Bumper-Hit = Nil

Release

else if Bumper-Hit = LEFT

Switch to State-2

else if Bumper-Hit = RIGHT

Switch to State-4

else Switch to State-3

State-2: If time-in-this-state > timeout-1

Switch to State-1

else

motor-command = turn-right

Release

State-3: If time-in-this-state > timeout-3

Switch to State-2

else

motor-command = back up

Release

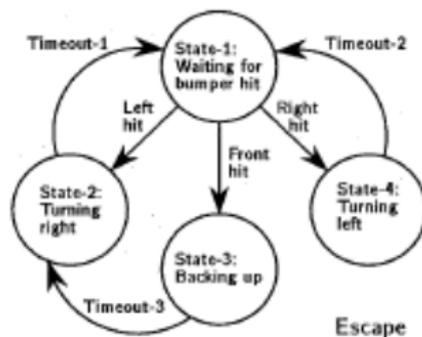
State-4: If time-in-this-state > timeout-2

Switch to State-1

else

motor-command = turn-left

Release



# Formalisme de contrôle des réflexes (suite)

Coopératif (sans préemption)

## Commentaires sur le code :

- Le code ci-dessus est appelé par le scheduler dans l'état 1.
- On utilise un driver **Bumper-Hit** pour connaître le type de collision.
  - Si pas de collision, **Release** rend la main au scheduler.
  - ➡ Cad. : lorsque la main est redonné à ce code, on sera encore dans l'état-1.
- Dans l'état-1, si collision à gauche, on passe à l'état-2 et exécute le code de cet état.
  - Si l'état-2 a gardé le contrôle (de l'UC) plus longtemps que Timeout-1, alors le contrôle est passé à l'état-1. Sinon, on émet une commande moteur (à droite) et le contrôle est passé au scheduler.
- Même principe pour les états 3 et 4 avec des timeout et commandes moteurs différentes.
- A l'état-3, on émet une commande pour reculer pendant Timeout-3 puis passe à l'état-2.

# Formalisme de contrôle des réflexes (suite)

Coopératif (sans préemption)

Réflex xx

Entrées : I1, I2, ..., In

Sorties : O1, O2, ..., Om

Etat-1 : {le code de l'état 1}

Etat-2 : {le code de l'état 2}

.....

Etat-n : {le code de l'état n}

- Implantation réel : le code d'un état doit être court pour libérer l'UC rapidement (**puisque pas de préemption**) car chaque état/MEF garde et peut bloquer l'ensemble.
- N.B. : On peut traduire les MEFs (et les AEFs d'une manière générale) par un code simple et systématique.

NB : En général, les MEFs sont séquentielles sans la notion de Release et Timeout, mais on peut leur envisager ce types d'extensions.

# Formalisme de contrôle des réflexes (suite)

Coopératif (sans préemption)

- Schéma utilisant MEF coopératif : le code du scheduleur sera le suivant :

Shéduleur Coopératif

Call Réflexe-1

Call Réflexe-2

.....

Call Réflexe-n

Call Arbitre

- Pour un schéma Coopératif, le scheduleur **boucle à l'infini** et appelle les Réflexes un par un.
- Le réflexe actif calcule pendant un certain temps puis rend le contrôle au scheduleur.
- Une fois par itération (au moins), le scheduleur appelle **Arbitre** pour faire circuler (transmettre) les commandes et résoudre les conflits (cf. noeud S).

# Ctrl Coopératif des réflexes (non préemptif)

## L'arbitre

- La connexion entre les modules réflexes (behavior) dans un réseau de réflexes est spécifié par le diagramme des réflexes.
- Pour connecter la MEF Escape à une MEF de moteur (controle les moteurs) :

**Connecter** (les entrées et les sorties) :

Sorties : **Escape**, commande-moteur

*émise par Escape*

Entrées : **Moteur**, commande-moteur

*vers moteurs*

- I.e. : connecter les commandes en sortie de Escape (qui sont des commandes moteurs) aux entrées des moteurs (leur commandes en entrée = commande-moteur).
  - La fonction Arbitre transmet la commande émise par Escape aux moteurs.
  - La même commande émise peut être transmise en entrées de plusieurs autres.
- On implante le noeud S (supress) en **ordonnant** ces connexions.

# Ctrl Coopératif des réflexes (non préemptif) (suite)

L'arbitre

## Exemple :

### Connecter :

Sorties : **Réflex-1**, Sortie-R1

Entrées : **Réflex-2**, Entrée-R2

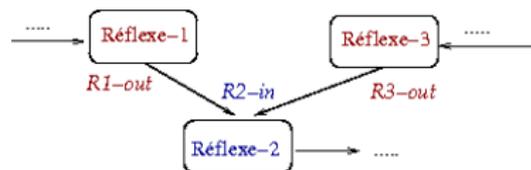
*émission par Réflexe-1  
vers Réflexe-2*

### Connecter :

Sorties : **Réflex-3**, Sortie-R3

Entrées : **Réflex-2**, Entrée-R2

→ Figure : si dans la même itération du scheduler, *Réflexe-1* calcule une valeur **R1-out** et *Réflexe-3* la valeur **R3-out**, l'arbitrage peut s'assurer que seule **R3-out** atteint **R2-in** (traduction neud **S** par un simple tri/ordre).



# Ctrl Coopératif des réflexes (non préemptif) (suite)

## L'arbitre

- Une cmd. exclue une autre ; une commande peut s'ajouter à une autre,
  - ↳ Exemple : reculer + tourner ou même reculer + avancer ...
    - Dans ce cas, l'arbitrage prendra une *décision* en éliminant toutes les commandes à destination d'un même réflexe.

# Ctrl Coopératif des réflexes (non préemptif) (suite)

L'arbitre

## Introduction de la notion de temps :

➤ si un passage du scheduler prend un cycle (un tick ou une unité) du système, alors un message (commande émise) peut supprimer (noeud S implicite) un 2e si le 2e est arrivé pendant le tick du premier.

➤ Une bonne implantation rendra cette caractéristique explicite. On peut avoir d'autres unités qu'un tick.

• Il existe d'autres types de mécanismes d'arbitrage que le principe de suppression (noeud S).

➤ Le principe du *subsumption* (Brook) : noeud d'**inhibition**.

➤ Un noeud d'inhibition fonctionne comme un switch et un message arrivé par une connexion dominante ne remplace pas les messages des connexions inférieures. Mais plutôt, ils empêchent le noeud d'inhibition de transmettre le message venant des connexions inférieures (au lieu de transmettre puis filtrer par S).

# Système à Contrôle Préemptif

- On a vu le principe coopératif à l'aide des MEFs :
  - Intéressant pour comprendre.
  - Surtout **utile** si l'on ne dispose pas de vrai multi-tâche.
  - La difficulté : l'organisation et le passage de contrôle d'un réflexe à une autre dans la boucle du scheduler ; ce qui rend la programmation plus difficile.
- N.B. : les cartes embarquées n'ont pas forcément un dispositif multi-tâche.
  - ➡ Le contrôle Coopératif peut alors être une solution.

**Suite** : le principe préemptif, disponible dans la plupart des systèmes.

- Le scheduler est intégré dans l'OS. Les réflexes sont plus faciles à coder
- ➡ n'ont pas besoin de la partie qui gère l'abandon et la reprise de contrôle.

# Contrôle Multi-Tâche Préemptif...

- On reprend l'exemple de LEDs clignotantes.
- L'OS permet de lancer des **processus** qui exécutent le programme (LED) :

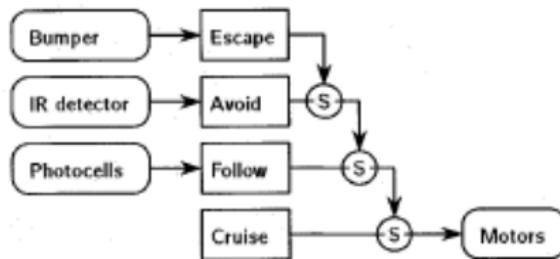
```
void multi_flash()
{while (1) {                // boucle infinie
    flash_leds();          // faire clignoter
    sleep(1);              // pause 1 seconde
    }
}
```

```
// Lancement du processus
int id = lancer_processus(multi_flash());
```

- Après ce lancement, on peut exécuter en parallèle d'autres commandes et processus, pendant que les LEDs clignotent.
- Pour supprimer ce processus par l'OS ou par un autre processus (sous conditions) : `kill_processus(id);`

# Implantation Préemptif de R3

- On implante R3 ( **légèrement modifié** ) :



Le principe de fonctionnement :

- 3 bumpers (avant gauche et droit, arrière)
- Capteurs IR (de proximité) pour éviter les obstacles (ou un robot adverse)
- Les Cellules photos pour chercher et suivre une lumière

# Implantation Préemptif de R3 (suite)

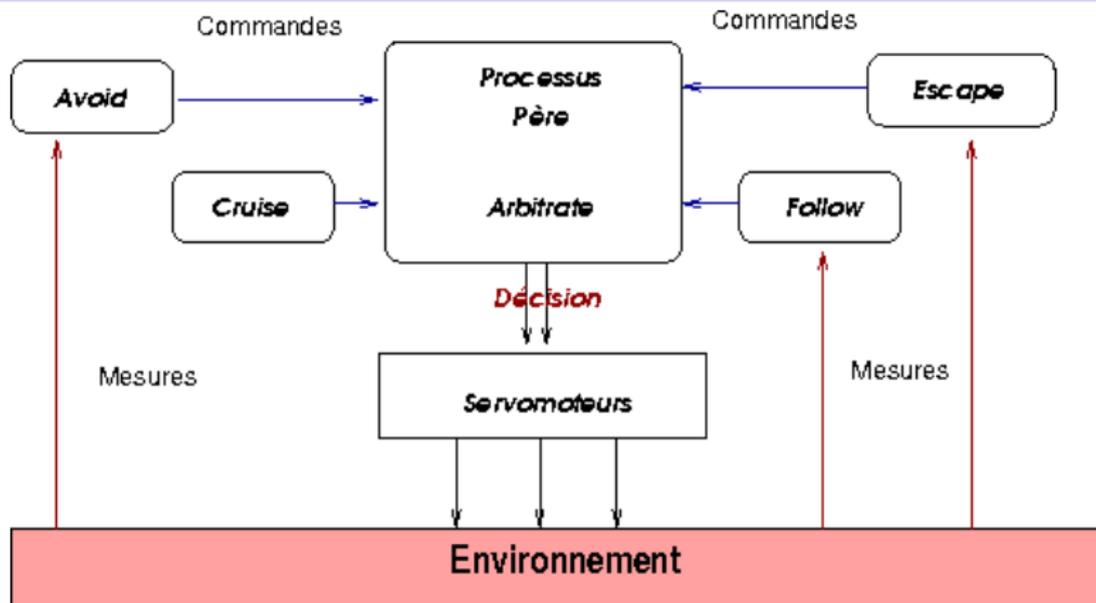


FIGURE: Les 6 processus parallèles

## Implantation Préemptif de R3 (suite)

### Code de Cruise :

- Rappel du but du réflex **Cruise** : faire avancer R3 en permanence.

```
int cruise_commande;           // la commande pour les moteurs
int cruise_output_flag;       // drapeau de signalisation

void cruise()
{while (1) {
    cruise_commande = FORWARD; // avancer (pour les moteurs)
    cruise_output_flag = True;  // une commande a été produite
  }
}
```

- Cruise se contente d'émettre FORWARD (pour les moteurs) et activer cette commande (pour sa prise en compte).

# Implantation Préemptif de R3 (suite)

**Code de Follow** (suit une lumière) :

```
int follow_commande;           // la commande émise
int follow_output_flag;       // drapeau de signalisation
void follow()
{ int left_photo, right_photo, delta;    // les cellules photoélectrique
  while (1) {
    left_photo = analog(1);             // lire le canal A/D 1
    right_photo = analog(0);           // lire le canal A/D 0
    delta = right_photo - left_photo;
    if (abs(delta) > seuil) {          // écart significatif
      if (delta > 0) follow_commande = TURN_LEFT;
      else follow_commande = TURN_RIGHT;
      follow_output_flag = True;       // activation demandée
    }
    else follow_output_flag = False;   // Désactiver (pas de différence)
  }
}
```

# Implantation Prémptif de R3 (suite)

## Commentaires :

- Si Follow détecte une différence entre les cellules gauche et droite, et si cette différence dépasse un certain seuil, il émet les commandes permettant de diriger R3 dans la direction ad-hoc. Sinon, Follow n'émet pas de commande.
- Rappelons que la commande émise aux moteurs est décidée par *Arbitrate*.

# Implantation Prémptif de R3 (suite)

**Code de Avoid** (lit les deux capteurs de proximité IR) :

```
int avoid_commande;           // la commande
int avoid_output_flag;       // drapeau de signalisation

void avoid()
{ int val_irs;                // valeur des IRs
  while (1) {
    val_irs = ir_detect();
    if (val_irs == 0b00)      // on ne voit rien
      avoid_output_flag = False; // Désactiver
    else {
      switch (val_irs) {
        case 0b11 :           // les 2 IRs "voient" qq chose
          avoid_commande = ARC_LEFT; // virage à gauche
        case 0b10 :           // IR gauche "voit" qq chose
          avoid_commande = ARC_RIGHT; // virage à droite
        case 0b01 :           // IR droit "voit" qq chose
          avoid_commande = ARC_LEFT; // virage à gauche
      }
    }
  }
}
```

## Implantation Préemptif de R3 (suite)

```
        default : ;
    }
    avoid_output_flag = True;
}
}
```

- Si le capteur de gauche voit quelque chose, on émet une commande pour tourner à droite. Dans les autres cas, on tourne à gauche.
- Une version aérée (pour comprendre) du même code :../..

# Implantation Préemptif de R3 (suite)

```
int avoid_commande;           // la commande
int avoid_output_flag;       // drapeau de signalisation

void avoid()
{ int val_irs;                // valeur des IRs
  while (1) {
    val_irs = ir_detect();
    if (val_irs == 0b00)      // on ne voit rien
      avoid_output_flag = False; // Désactiver
    else {
      avoid_output_flag = True;
      if (val_irs == 0b10)    // IR gauche "voit" qq chose
        avoid_commande = ARC_RIGHT; // virage à droite
      else                    // si les 2 ou de droite voit
        avoid_commande = ARC_LEFT; // virage à gauche (par défaut)
    }
  }
}
```

# Implantation Préemptif de R3 (suite)

**Code de Escape** (évite les collisions avec les obstacles détectée par la ceinture de détecteurs circulaire) :

```
int escape_commande;           // la commande
int escape_output_flag;       // drapeau de signalisation

void escape()
{while (1) {
    bump_check();              // état des détecteurs
    if (bump_left && bump_right) { // choc frontal
        escape_output_flag = True;
        escape_commande = BACKWARD; // Reculer un peu
        usleep(200);
        escape_commande = TURN_LEFT; // puis à gauche
        usleep(400);
    }
    else if (bump_left) { // choc à gauche
        escape_output_flag = True;
        escape_commande = TURN_RIGHT; // tourner à droite un peu
    }
}
```

# Implantation Préemptif de R3 (suite)

```
    usleep(400);
}
else if (bump_right) {                                // choc à droite
    escape_output_flag = True;
    escape_commande = TURN_LEFT;                     // tourner à droite un peu
    usleep(400);
}
else if (bump_back) {                                 // choc à l'arrière
    escape_output_flag = True;
    escape_commande = TURN_LEFT;                     // tourner à droite un peu
    usleep(200);
}
else escape_output_flag = False;                       // Désactivé
}
}
```

- Si choc à l'arrière, on tourne à gauche.
- N.B. : l'appel **bump\_check()** ; permet de connaître l'état des Bumpers.

# Implantation Prémptif de R3 (suite)

## Code du Servomoteur

- Habituellement, les servomoteurs n'ont pas un code important.
  - Mais il est possible d'avoir des servomoteurs sophistiqués (englobant l'odométrie).
- Sur notre Robot, le driver Moteur du Robot est un simple code qui en permanence lit la valeur d'une variable **motor\_input** et transmet une valeur adéquate aux ports connectés aux moteurs (servo moteurs).
- **Dans la simulation**, un processus est affecté au servomoteur permettant de simuler l'avancement du Robot R3.
- De plus, un processus indépendant (**Affichage**) est chargé de relever en permanence la position du Robot ainsi que les états de divers capteurs/Bumpers.
  - Ces données sont affichées sur un écran.

# Implantation Préemptif de R3 (suite)

## Capture d'écran du simulateur (

The screenshot shows a terminal window with a grid-based simulation on the left and a control panel on the right. The grid contains several 'x' characters representing objects or robots. The control panel on the right has several sections with labels and values.

```

Robot : Attachement des Zones fait
>>>Le semaphore 588129 existe deja. Sa valeur actuelle est 1
>>>Le semaphore 720859 existe deja. Sa valeur actuelle est 1
Robot : Attachement des Zones fait
>>>Le semaphore 588129 existe deja. Sa valeur actuelle est 1
>>>Le semaphore 720859 existe deja. Sa valeur actuelle est 1
Robot : Attachement des Zones fait
>>>Le semaphore 588129 existe deja. Sa valeur actuelle est 1
>>>Le semaphore 720859 existe deja. Sa valeur actuelle est 1
Robot : Attachement des Zones fait
LE PERE va ATTENDRE LES FILS.
>>>Le semaphore 588129 existe deja. Sa valeur actuelle est 1
>>>Le semaphore 720859 existe deja. Sa valeur actuelle est 1
Robot : Attachement des Zones fait
  
```

Control Panel (Right):

```

-----
| OUISE
| FORWARD
|-----
| PRETS=FULLM
| / (R= S)
| NDIR_cad
|-----
| IS (-d=nd)
| ARC_LEFT (R= 1)
|-----
| Buteurs G/R/B
| ---
|-----
| Escape
| NDIR_cad
|-----
| Arbitre (Hoteur)
| ARC_LEFT
|-----
| Pos A. Bir
| 14/12/90
  
```

# Implantation Préemptif de R3 (suite)

## Le Processus Père : la fonction main()

- Chaque réflexe codé ci-dessus émet une ou plusieurs commandes.
  - Ces commandes sont adressées au Robot.
- La fonction **Main()** contient le code du **processus père**. Il est lancée au départ pour activer les réflexes sous forme de processus  **fils**  qui s'exécuteront en parallèle.
  - main() peut être mise en ROM ou en RAM sauvegardée par batterie.

```
int main()
{lancer_processus(driver_motor());           // lancer pilote moteur
  lancer_processus(cruise());
  lancer_processus(follow());
  lancer_processus(avoid());
  lancer_processus(escape());
  lancer_processus(arbitrate());
```

- ```
}
```
- On ajoutera à ces processus les deux autres :
    - Servomoteur
    - Affichage

## Implantation Prémptif de R3 (suite)

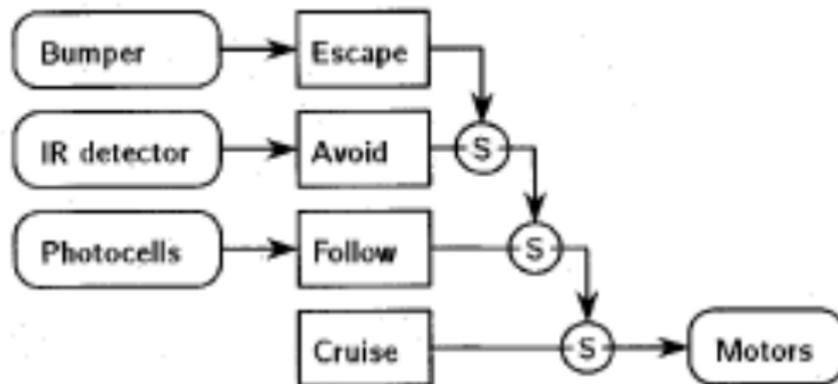
### La fonction `Arbitrate()`

- Le code du processus d'arbitrage permet d'établir la stratégie de résolution des conflits.
- Cette fonction implante un principe de *passage de messages* entre les processus.
- Une fois les réflexes codés, le diagramme de conception précise comment les sorties et les entrées doivent être connectés.
  - La fonction implante donc une liste ordonnée des actions de connexion.
  - Lorsque plusieurs commandes sont adressées à une entrée de réflexe (souvent le moteur ici), on donnera priorité à la dernière qui subsume les précédentes.

# Implantation Préemptif de R3 (suite)

## Suite (arbitrate) ...

- Rappel du diagramme (dit *cablé*) de conception de R3 :

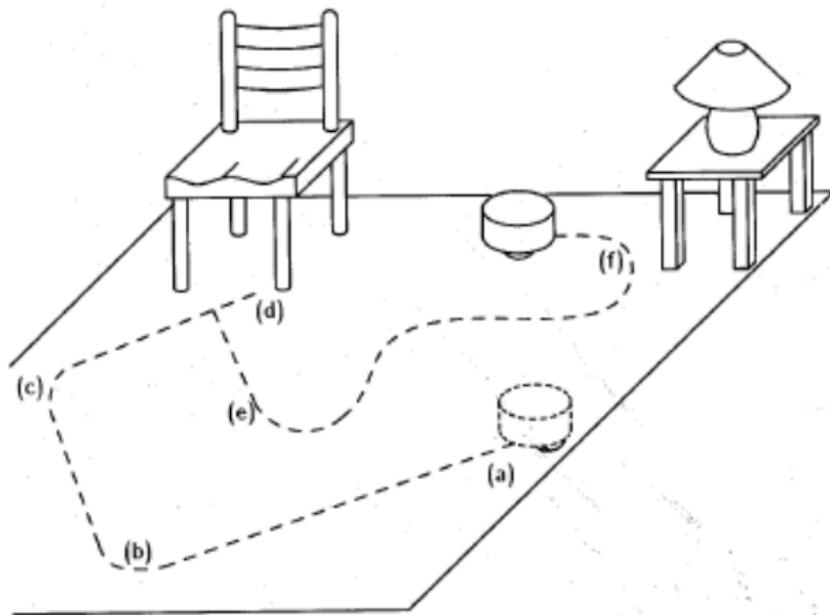


# Implantation Prémptif de R3 (suite)

```
void arbitrate()
{while (1) {
    if (cruise_output_flag == True)
        motor_input = cruise_commande;           // connecter les sorties des
    if (follow_output_flag == True)               // réflexes aux entrées du
        motor_input = follow_commande;           // moteur
    if (avoid_output_flag == True)
        motor_input = avoid_commande;
    if (escape_output_flag == True)
        motor_input = escape_commande;
    sleep(tick);                                  // attendre 1 tich (fixé d'avance)
}
}
```

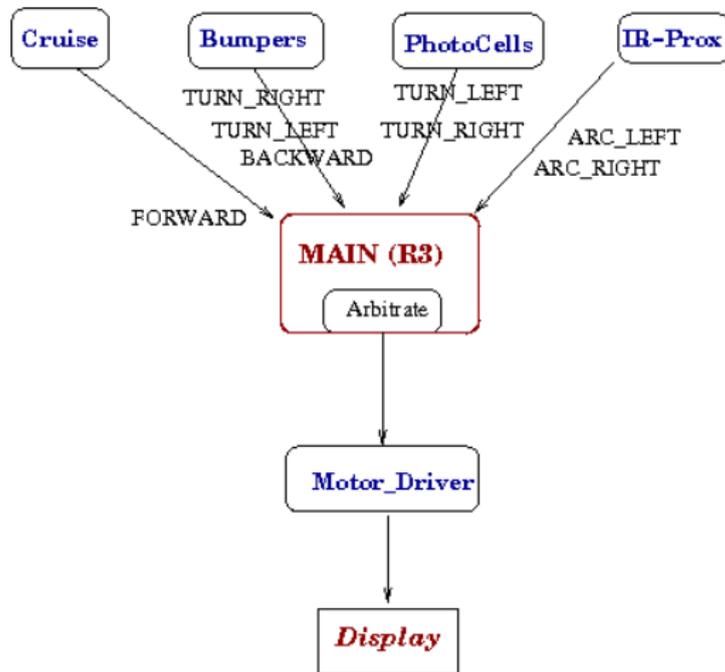
# Implantation Prémptif de R3 (suite)

Exemple du milieu :



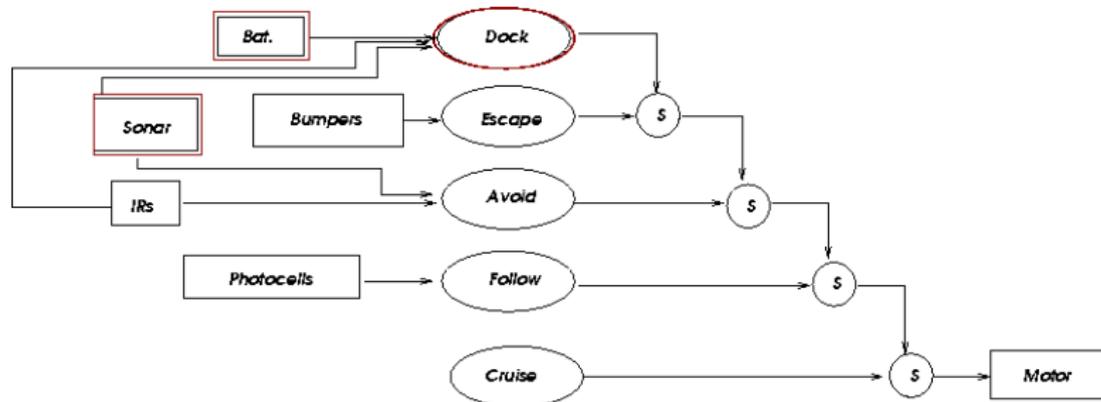
# Implantation Préemptif de R3 (suite)

## Schéma Simulation



# A réaliser

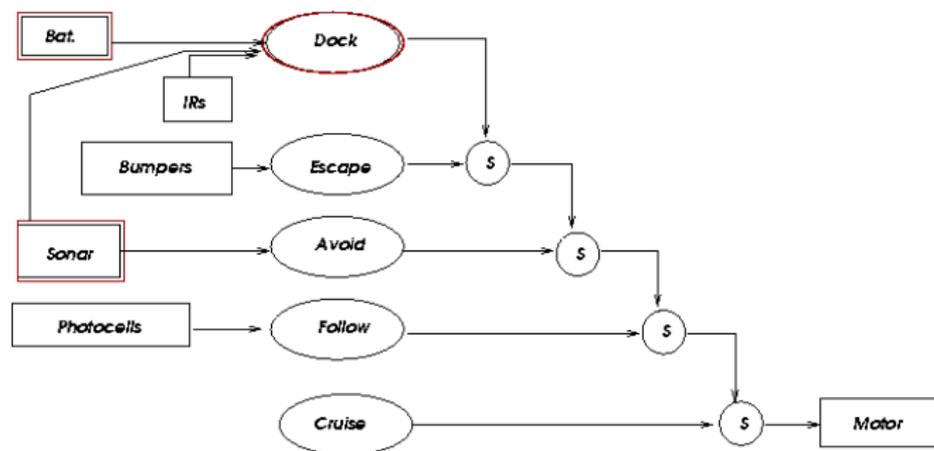
- Ajouter le module Duck : si le niveau de batterie devient bas, aller au stand de recharge. La commande deviendra prioritaire sur toutes les autres.
- Le stand est signalé par une balise IR.



# A réaliser (suite)

Un autre schéma :

► ici, **Avoid** est alimenté par Sonar et **IRs** sont seulement utilisés par **Dock**.



# A réaliser (suite)

## A propos du Sonar

- Rappel : IR permet de détecter une proximité :
  - ➔ Quelque chose est/n'est pas là..
- Sonar : dispositif sonore, permet d'obtenir une distance.
  - On mesure le *temps de vol* (time of flight) entre le ping et son écho.
  - On connaît la vitesse du son ➔ on peut mesurer la distance.
- Un exemple de Sonar :

- Pour mesurer une distance, on émet un bref signal (de l'ordre de 400 volt) et on attend l'écho

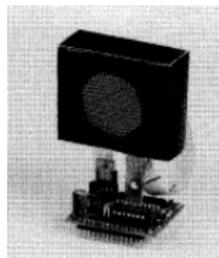


FIGURE: fff

# A réaliser (suite)

```
void init_sonar() {  
  // initialisation : on écrit souvent un bit d'effacement  
}  
void ping() // initialisation d'un ping  
{commencer_ping(); // on écrit souvent un bit  
  usleep(30000); // attente de l'écho  
  effacer_bit_echo();  
}  
// Déterminer si un écho est reçu. Si oui, calculer sa longueur  
// En cas d'écho, on dispose de deux dates (temps) tick1 et tick2  
float distance()  
{if (pas_echo()) return -1;  
  // calculer le temps et convertir en longueur  
  return (tick2 - tick1)/2*0.33; // Distance en millimètre (aller-retour)  
// Vitesse du son dans l'air : 330 m/s = 0.33 : millimètre par microsecond  
}
```

# Quelques Projets

- Quelques projets Robotiques intéressants.
  - ① Lewis & Clark : on s'évite!
  - ② Autres : à continuer ...

# Quelques Projets (suite)

## Lewis & Clark :

- But : "bouger et ne pas se figer!".

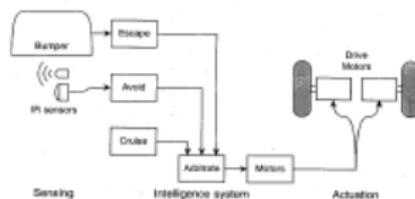


FIGURE: Le diagramme **SPA réactif** pour Lewis & Clark. Priorité à Escape.

- On lance un certain nombre de Robots (e.g. 5) dont le comportement est identique selon le schéma ci-dessus.
  - Cruise = **aller vers**,
  - Escape et Avoid : **déguerpir**.
- On souhaite programmer (simulation) ce ensemble.

# Déplacement Robot

- A\*
- Monte Carlo
- ... etc