Projets Robots (PE47-PE51)

Ecole Centrale de Lyon

2007-2010

Alexander Saidi

21 février 2008

Robotique : un domaine multi disciplinaires

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

PLAN :

- Système Embarqué
- Système Temps Réel
- Robotique
  - ➤ Programmation Robot & stratégies

1.1 Systèmes Embarqués

- De nos jours, les processus automatiques ont remplacé ce qui était fait à la main :

  ➤ 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 mintenant des systèmes <u>mécatroniques</u> (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*).

• 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).

1.1.1 Caractéristiques principales d'un système embarqué

- Est principalement numérique ;
- Met en oeuvre (généralement) un processeur (ou une forme simplifiée : DSP, PIC, ...).
- 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.

1.1.2 *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 des systèmes libres (e.g. Linux) pour l'embarqué
  - ➥ Ex. uCLinux, ...

../..

… Le marché d'embarqué (suite) …

- 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).

- 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.

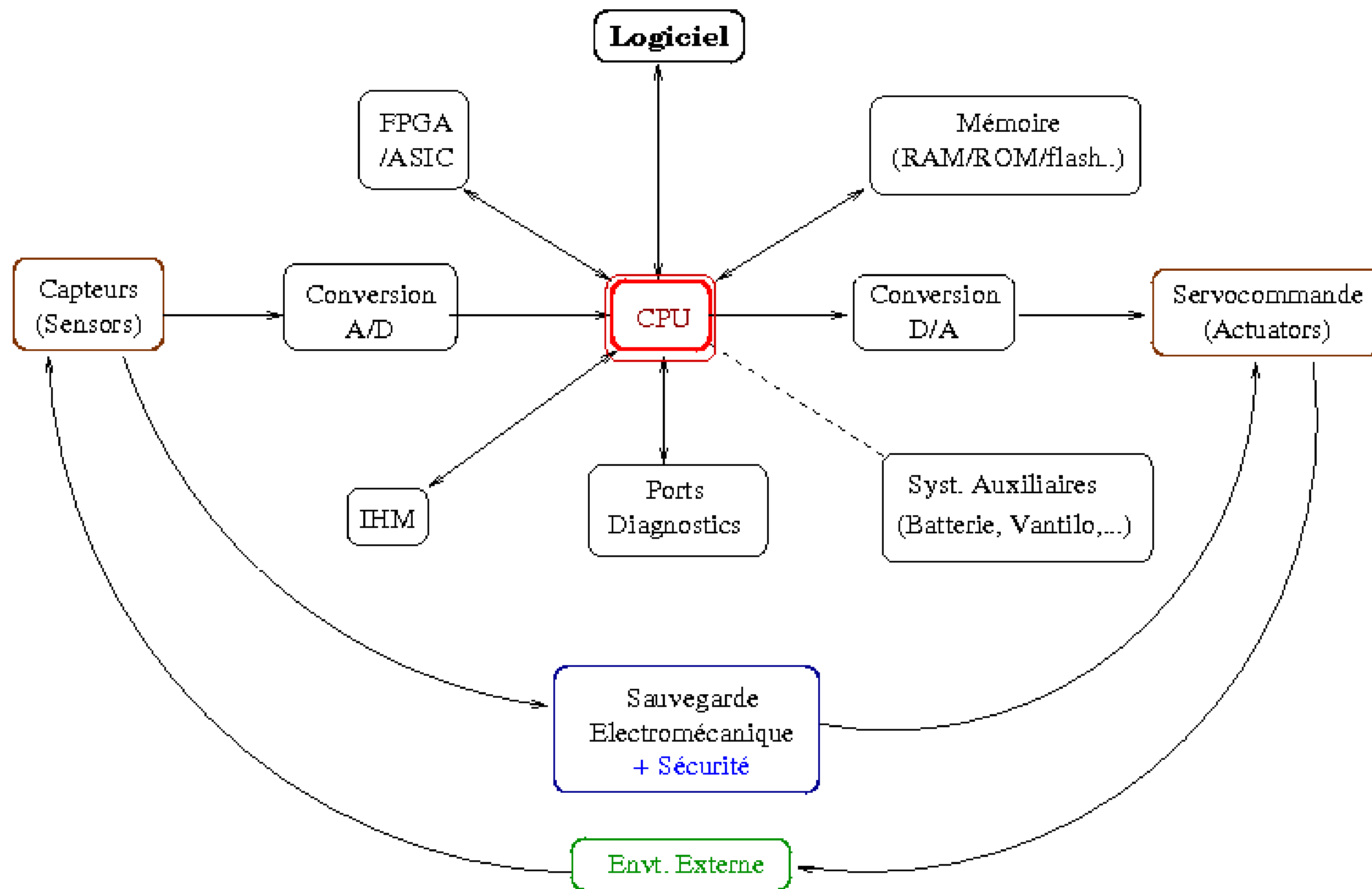
- Le prix moyen d'un processeur embarqué est de 6$ contre 300 pour un processeur PC. standard.

  ➥ Le marché de processeur PC est faible mais très lucratif !

1.1.3 *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, ...
- . . .

Schéma d'un système embarqué typique ../..


Logiciel
FPGA
/ASIC
Mémoire
(RAM/ROM/flash..)
Capteurs
(Sensors)
Conversion
A/D
CPU
Conversion
D/A
Servocommande
(Actuators)
IHM
Ports
Diagnostics
Syst. Auxiliaires
(Batterie, Vantilo,...)
Sauvegarde
Electromécanique
+ Sécurité
Envt. Externe

1.1.4 *Un système embarqué typique (figure)*

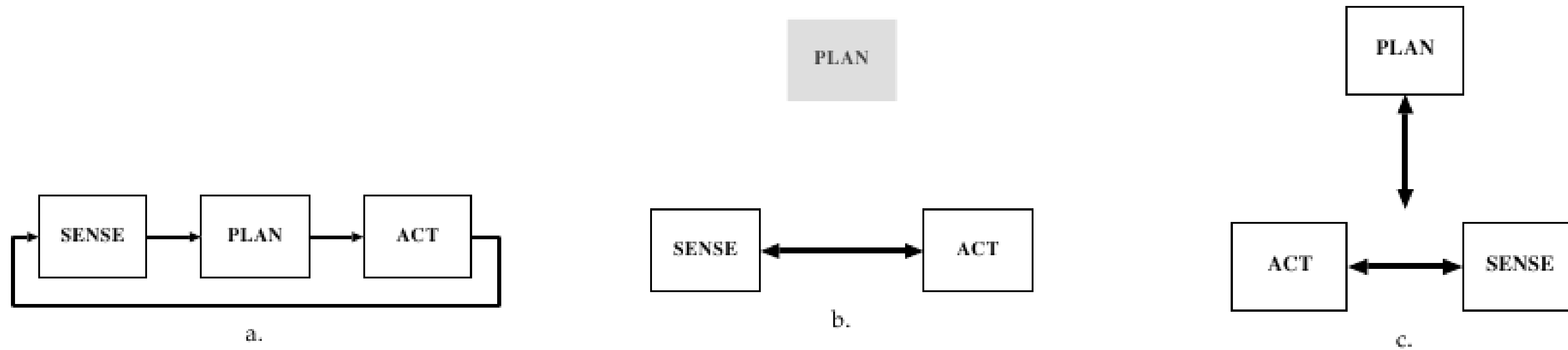
- On trouve souvent en entrée des capteurs + convertisseurs A/D.
- En sortie, des actionneurs souvent analogiques couplés aux convertisseurs D/A.
- Souvent, un circuit FPGA (joue le rôle de co-processeur) épaule le processeur.

➥ 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 ../..

1.1.5 *Aperçu des paradigmes* **SPA** *(Sense, Plan, Act)*

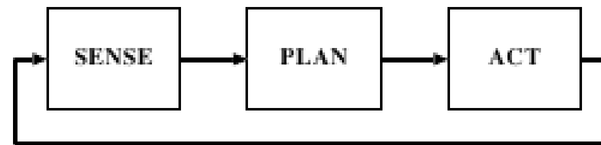


a) **SPA** : Hiérarchique

b) **SA** : réactif

c) **P,S-A** : Hybride délibératif/réactif.

• 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	[illegible] directives	Actuator commands

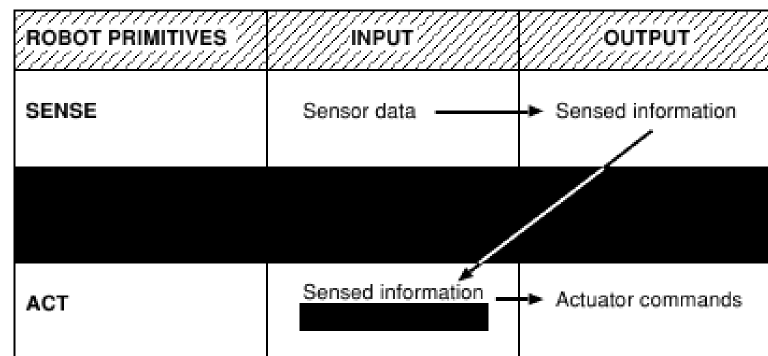
• Lourd, gourmand en ressources et en calcul, rigide, ...

• 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 ../..

• **<u>Hybride</u>** : dans la phase *Planification Délibérative*, le Robot cherche à planifier puis décomposer la *mission* en sous tâches ➝ tâche = *Behavior* comme dans Réactif.

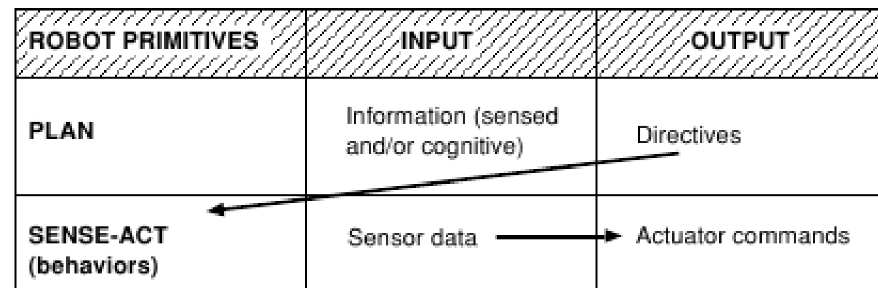
• Le schéma devient de fait **P,S-A** (figure (c))

• 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).

➥ <u>Ex</u> : la présence d'un obstacle concerne A et P (MAJ du map global)

• 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

1.1.6 *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 <u>dès la conception</u>.

➥ 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.

../..

.... Environnement d'embarqué (suite) :

• Les systèmes embarqués récents sont très communicants (possible via les processeurs puissants).

• 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 ! !

1.1.7 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* dans les algorithmes (cf. exemple de tri) !

1.1.8 *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 !)

1.1.9 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.

1.2 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 taches dans des limites temporelles bien précises.*

- Ou encore (rappel) : *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 rafrîchissement de l'information.

- 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 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 un stimuli extérieur.*
  - ➥ Il peut être embarqué (dans une voiture, un avion, un Robot, ...)
  - ➥ Permet l'automatisation d'un processus

1.2.1 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 par rapport à 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 C° peut même entrainter 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.

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).

1.2.2 Gestions du temps dans 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.

1.2.3 Les critères des RTOS

- 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) :

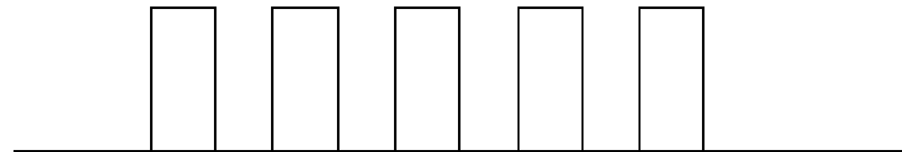
➥ Quelque soient les contraintes temporelles (échelonnées entre quelques micro-secondes à quelques secondes), <u>on doit pouvoir prédire le comportement</u>.

1.2.3.1 Prévisibilité

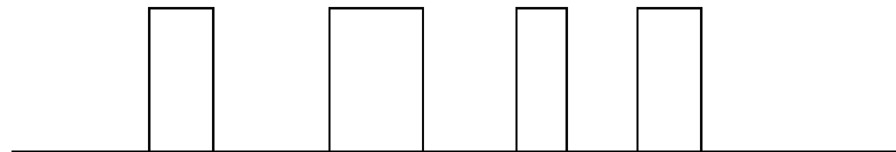
- Si l'on ne peut pas garantir les délais lorsqu'un système est chargé :

```
#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);
}
```



Système non chargé : une demi périoe de 50 ms varie très peu.
(comportement proche d'un RTOS)



Système chargé : une demi périoe de 50 ms varie entre 43 et 58 ms.

- 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).

1.2.4 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).

1.2.5 3 Approches en conception des RTOS

1- Synchrone, Déterministe : la plus courante.

➥ Une boucle infinie, exécution des instructions l'une après l'autre, aucun mécanism 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*).

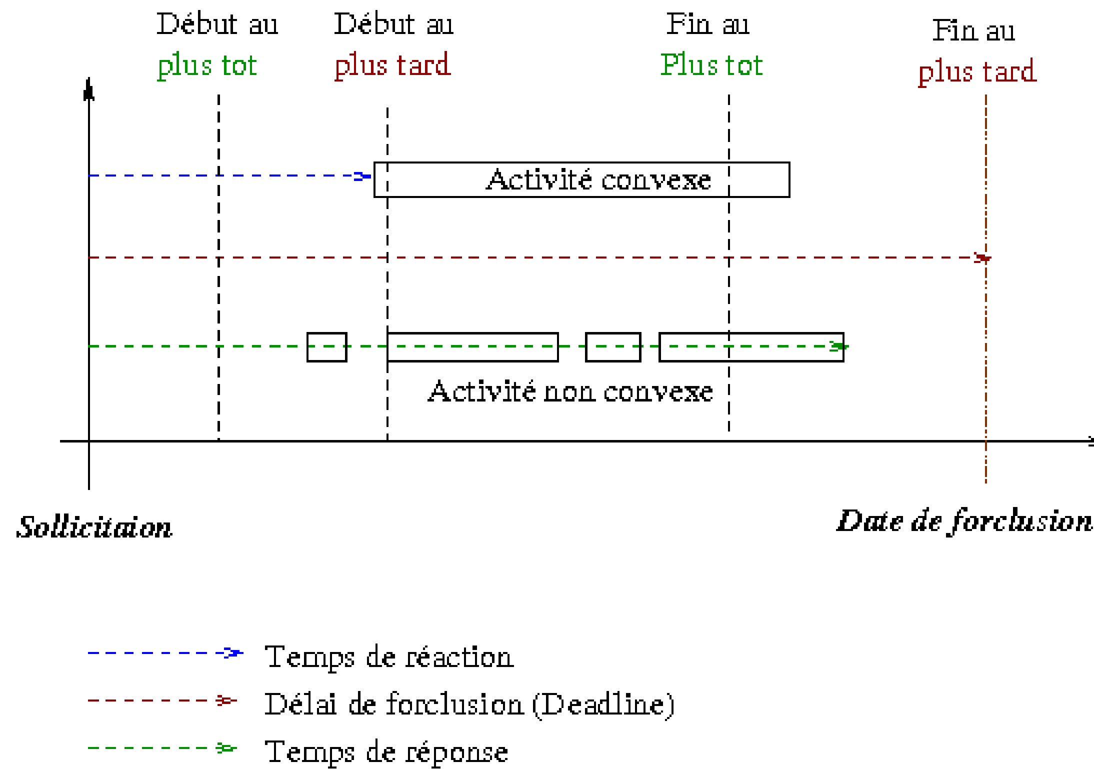
1.2.6 Multitâche préemptive

- Différentes activités pouvant communiquer ensemble ➙ *agents*.
- Chaque activité a ses contraintes temporelles précises.

➥ Après l'avènement déclencheur, l'agent doit réagir dans un temps minimum et surtout avant un délai maximum (Deadline) = au plus tard.

➥ De même, on définit un temps minimal avant lequel l'agent doit avoir fini son activité = au plus tôt.

Figure : Les contraintes temporelles dans un OSTR ../..



- 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 événement déclencheur.

1.2.6.1 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 de priorité).
  - ➥ d'où la notion de **péremption**.

1.2.6.2 Exemples

- 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 exemple 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 (processus *cron*)....

- **La gestion des activités nécessite un mécanisme de synchronisation.**

  ➥ 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, ...

1.2.7 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.

1.2.8 Sémaphores

• A considéré 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)

1.2.9 Code d'un sémaphore

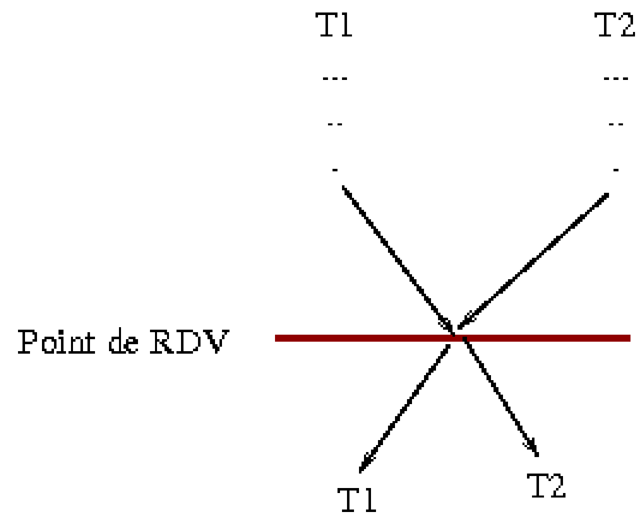
```
struct Semaphore
  { valeur : entier;
    attente : file d'attente;
  }

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
}
```

1.2.9.1 Sémaphores Binaires

- 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.



1.2.9.2 Sémaphores pour partage de ressources

- Une ressource peut être **partageable** ou non, à accès **simultané** possible,...
- **Exemples** de ressources ne pouvant être utilisées simultanément

  - l'accès à une imprimante (non gérée par une tâche *IMP_manager*),

  - l'accès à un driver matériel,

  - l'accès à une variable commune ou autres mécanismes (ressource)

- Solutions :

  ➥ Opération insécable (TAS, l'écriture d'un booléen) et **exclusion mutuelle**

  ➥ Sémaphore (plus souple, K tickets/accès possibles).

1.2.10 Gestion d'accès en exclusion mutuelle

1.2.11 Par un sémaphore

- 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)

1.2.12 Par Test and Set (TAS)

• **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;
```

• **TAS** résout le problème de **Section Critique** (SC) de la manière suivante :

➥ Les processus partagent une variable *Verrou*.

➥ Chaque processus Pi possède une variable locale $Test_i$.

- Code de chaque processus P_i :

```
Test_i : entier;
Répéter
    <SR>
    TAS(test_i, Verrou);
    Tantque Test_i=1 faire
        TAS(test_i, 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.

... TAS (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 :

$$Reg = 1; \quad XCHG \quad Reg, Verrou$$

$$\Rightarrow Reg := Verrou \text{ et } Verrou := Reg = 1 \quad \simeq \text{TAS.}$$

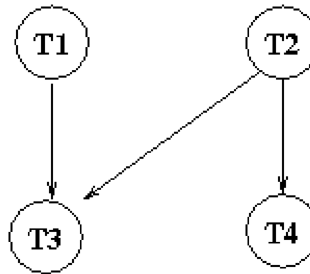
• NB : TAS et XCHG réalisés avec verrouillage du bus.

1.2.13 *Utilisation et exemples de sémaphores*

- Les sémaphores sont utilisés dans différentes situations.

1.2.14 *Exemple d'utilisation de sémaphores pour la synchronisation*

- Soient 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
```

1.2.15 Un exemple en C/C++

- Montrer un exemple de partage de l'écran.

1.2.16 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.

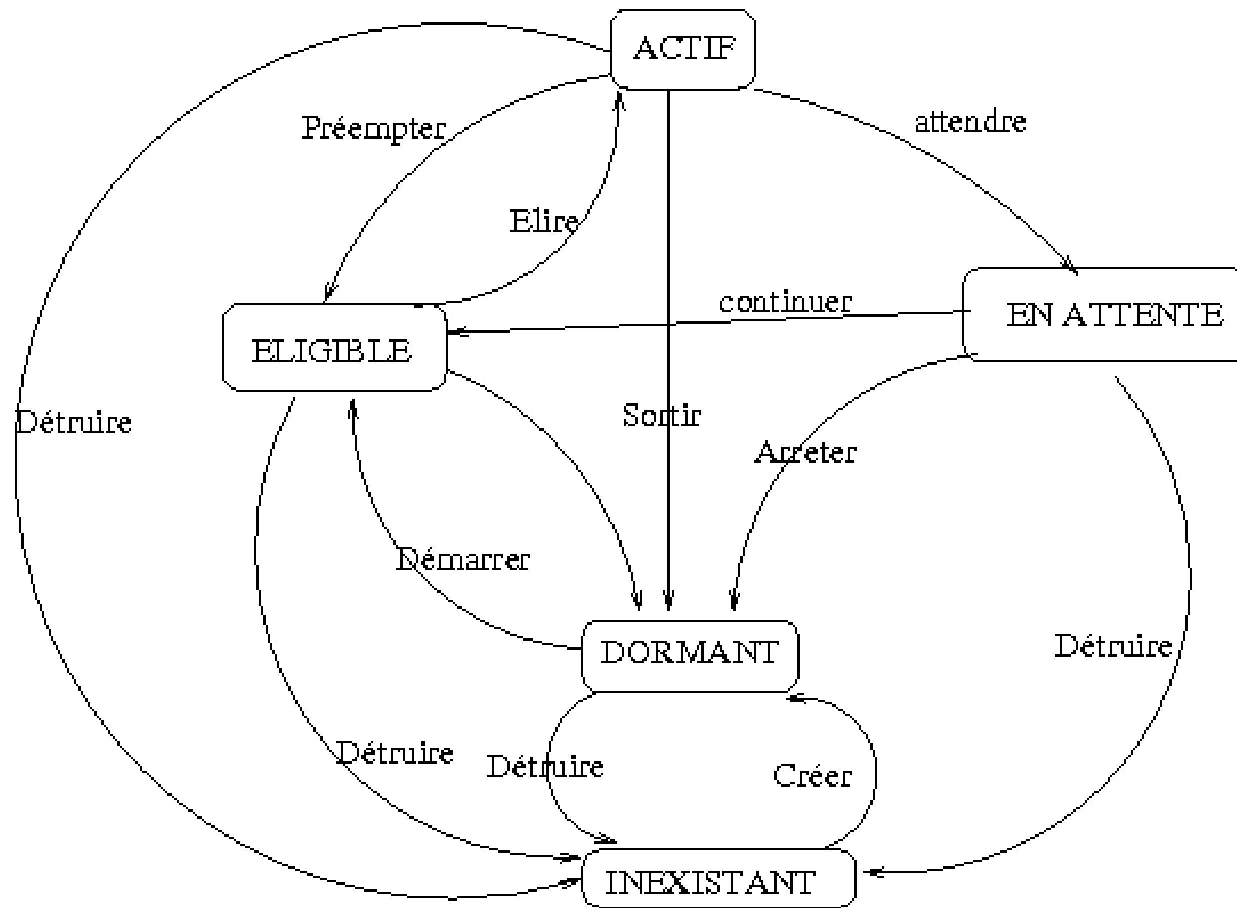
➤ Exemples divers, mécanismes de passage de messages (peuvent servir à la synchro), Client-Serveur, ...

1.3 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.
- 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, .../..

1.3.1 Le diagramme d'états d'une tâche



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

Taches : le principe commune à tout RTOS

• Une tâche : Inexistant ⇒ Créée ⇒ 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 ⇒ **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, on préfère une gestion par **priorité** (qui peut tenir compte des stratégies ci-dessus).

1.3.2 *Gestion des Priorités (des tâches)*

- 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**

... Gestions des priorités (suite) ...

• Il reste quand même des cas particuliers qu'il faut traiter/prévoir/débloquer.

• **Un problème courant : problème d'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 de priorité

➥ 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)) ../..

- **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.

1.4 Processus et threads

1.5 Temps Réel sous Linux = Temps partagé

• 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.

... OS Temps partagé (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 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 entraîner des fluctuations dans les temps d'exéc des taches....

• **Bref : Il y a du chemin pour passer du temps partagé au Temps Réel.**

1.5.1 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.

1.6 Programmation Robot

- Progrès en Robotique sur 2 fronts
  - ➤ Matériel
    - ➥ Microcontroleurs
    - ➥ Capteurs
  - ➤ Logiciel (soft)
    - ➥ Contrôle à base de réflex (**Bahavior-based** : nécessite peu de ressources de calcul)

1.6.1 Approches de programmation Robot

- Deux approches principales :
  - ➤ **centralisée** (classique)
  - ➤ **décentralisée**.

1.6.2 *Approche centralisée (planification)*

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

  ➥ Beaucoup de calculs :

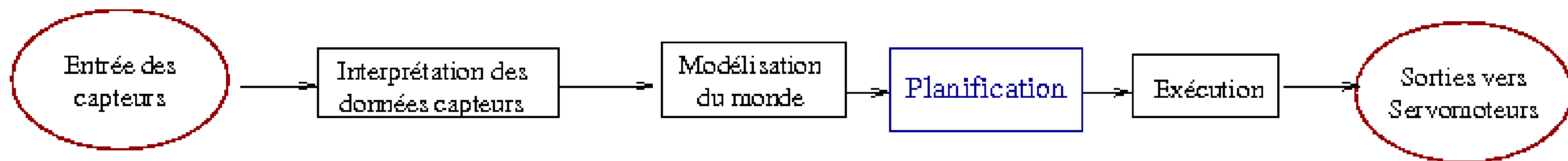


Fig. : les modules fonctionnels transforment les données capteurs en une série d'actions.

- Cette approche continue à faire l'objet des travaux importants.
- Le programme (soft) du robot est décomposé en une série de fonctions.
- Les bruits/conflits dans les données capteurs sont résolus ;

  ➤ un modèle (éventuellement incomplet) du monde réel est construit par le programmeur et par avance.

... Approche centralisée (suite) ...

• 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.*

1.6.2.1 Exemple Handey

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

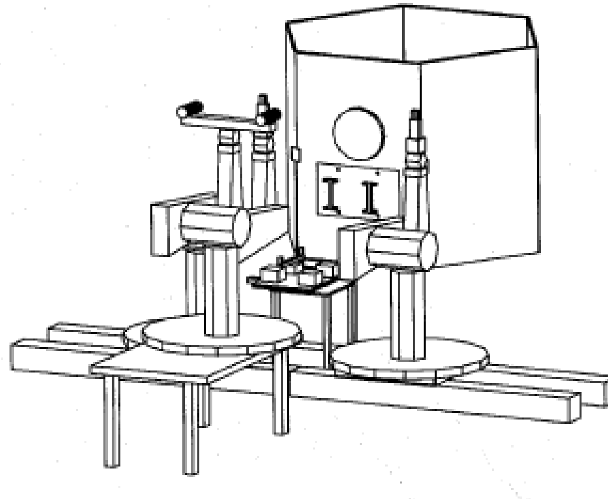


Fig. HANDEY (MIT) dans l'espace (manipulation par pince des satellites en orbite).

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

... 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 ?)

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

➤ Approche décentralisée : informations locales et partielles.

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

• Les robots sont peu autonomes dans ce modèle.

➤ Le monde alentour ne doit pas changer (surtout entre capture d'info et l'action !).

1.6.3 Approche décentralisée à base de réflexe (cybernétique)

• Plus récent ; Vient du MIT (R. Brooks, Labo IA)

• Idée : combiner le contrôle Temps Réel 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 <u>supérieur</u> peut suspendre une d'un niveau inférieur.

Approche décentralisée réactive (suite) ..

➥ Si les Réflexes du niveau supérieur sont déactivés (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.

1.6.3.1 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 d'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.

... Suite Robot R1 ...

• Entre les deux, le module Avoid s'appuie sur les données sonars et calcule en permanence des commandes envoyées au Motor.

• 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.

... Suite Robot R1 ...

- 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).

- La réflexe de cette stratégie permet au robot de rester à distance (loin) de tous les obstacles (repérables par les sonars).

1.6.3.2 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**.

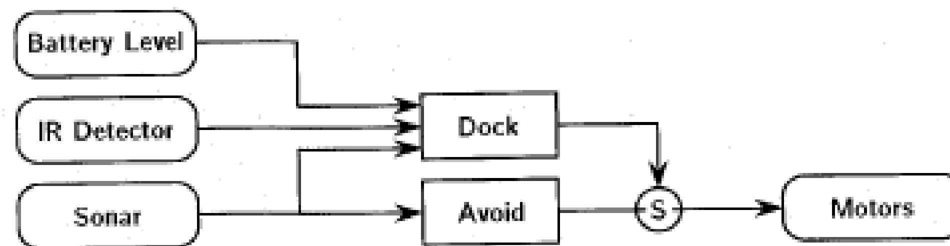


Diagramme à base de la réflexe du robot qui va en stand de charge

• **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.

... Suite Robot R2 ...

• Le noeud (S) du diagramme (suppressor) permet le passage de messages venant du câble d'origine directement vers la sortie.

• 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

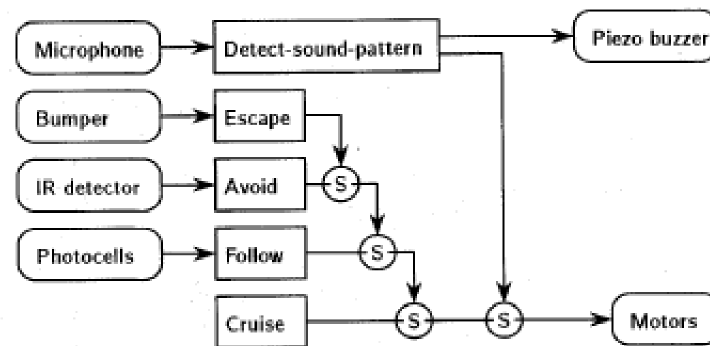
➥ Le Dock *subsume* Avoid et produit une compétence plus élevée.

• Un autre intérêt de cette approche réactive est la possibilité d'améliorer le système,

➥ **sans perturber** la compétence précédente ni l'alourdir l'ensemble.

1.6.4 *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 cellcules photoélectriques
  - ➤ 1 microphone
- La figure suivante donne un aperçu des compétences de R3 :



... R3 (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).

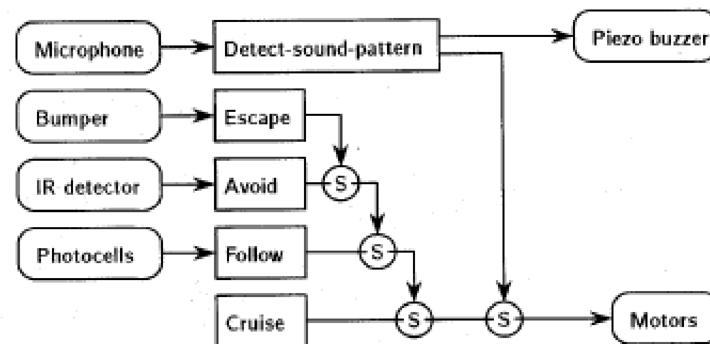
➤ Lorsqu'il y a une différence d'intensité entre les deux cellules, il commande les moteurs de tourner vers la source la plus forte.

... R3 (suite) ...

• 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



• 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.

• Si jamais les capteurs IR ne fonctionnent pas (pour éventuellement certains objets), le module "Escape" sera activé lorsqu'une collision est détectée (par la rangée de 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 commandes de Escape sont les plus prioritaires en matière de déplacement du Robot.

... R3 (suite) ...

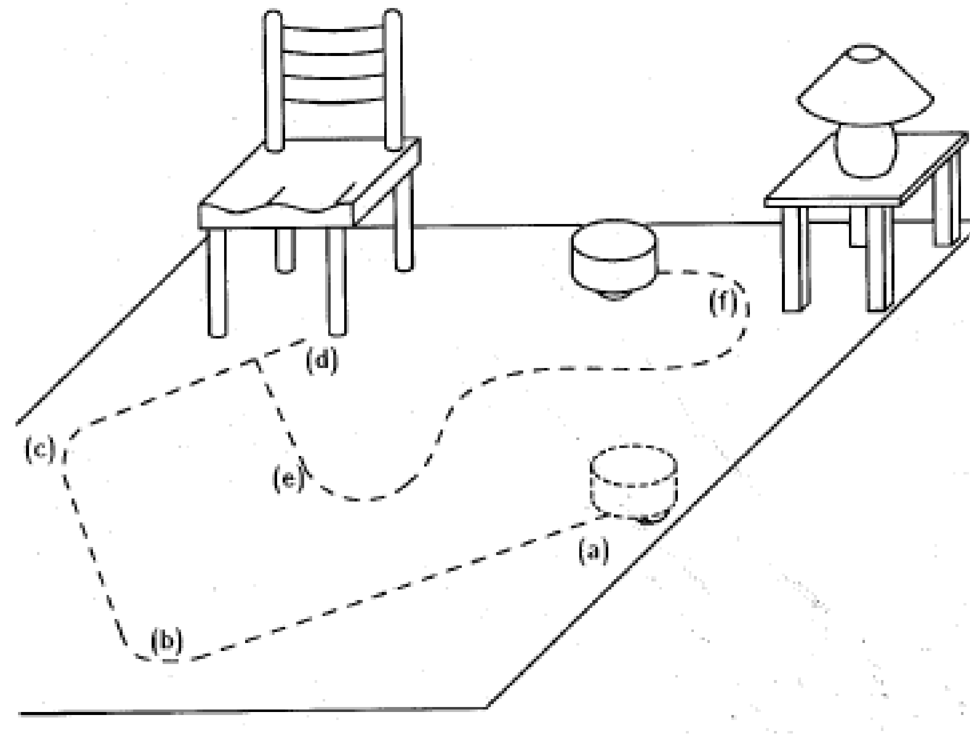
• Enfin, le module "Detect-Sound-pattern" fait jouer un son à R3 lorsque l'on frappe dans les mains avec une certaine fréquence suivi d'une pause.

➤ 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 object, R3 changera de direction et s'éloignera. Quand on frappera dans les mains d'une certaine manière, R3 s'arrête, joue un son puis se remet en route.

1.6.4.1 Un exemple de parcours pour R3



1.7 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 nécesaires : processus, scheduler et AEFs.

1.7.1 Processus et Scheduleurs

- 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
    }
}
```

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.

... Implantation (suite) ...

NB : les systèmes actuels fonctionnent à base de processus.

• Le code ci-dessus mobilisera toutes les ressources de ce programme mais celui-ci sera un processus parmi d'autres.

• Plus intéressant : que le code de R3 puisse 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 utiliser un **scheduleur**.

... Implantation (suite) ...

• Le scheduleur 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 scheduleur 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 scheduleur s'il y en a, sinon, à un autre processus).

➥ Mais alors, il faut retrouver où on était avant de donner la main au scheduleur / à un autre processus

➥ Brooks avait proposé d'assigner à chaque processus un MEF (machine d'état fini).

1.7.2 MEF

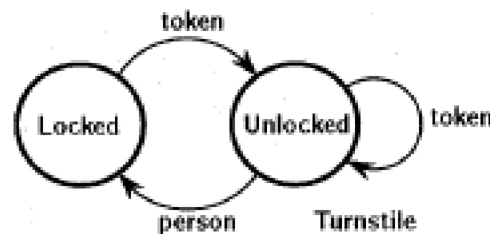
• En l'absence d'un schéduleur, on peut implanter un schéma de controle de réflexes par une MEF.

➤ Même en présence d'un schéduleur, cela nous aide à bien comprendre les réflexes.

• 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



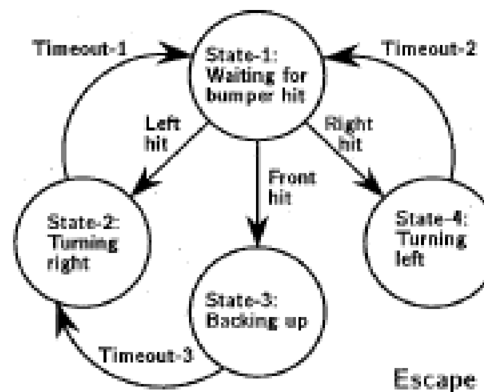
• On note $\sigma : S \times \alpha \rightarrow S'$

1.7.3 *Formalisme de contrôle des réflexes*

- Exemple : quand R3 se cogne dans un objet ➙ réflexe Escape.

  ➤ Escape utilsera un capteur de collision.

- La figure suivante donne la MEF.



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

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

• 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
```

- Le code ci-dessus est appelé par le schéduleur dans l'état 1.
- On utilise un driver **Bumper-Hit** pour connaitre le type de collision.
  - ➤ Si pas de collision, **Release** rend la main au scéduleur.
    - ➥ Cad. : lorsque la main est redonnéé 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 controle est passé à l'état-1. Sinon, on émet une commande moteur (à droite) et le controle est passé au schéduleur.
- Même principe pour les états 3 et 4 avec des timeout et commandes moteurs différentes.
- A l'état-3, on émet une comande pour reculer pendant Timeout-3 puis passe à l'état-2.

1.7.3.1 Le code d'un module Réflexe

```
Réflex xx
     Entrées : I1,I2, ..., In
     Sorties : O&, 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.

1.7.3.2 Le scheduleur

• Shéma utilisant MEF coopératif : le code du schéduleur sera le suivant :

```
Shéduleur Coopératif
   Call Réflexe-1
   Call Réflexe-2
     .........
   Call Réflexe-n
   Call Arbitre
```

• Pour un schéma Coopératif, le schéduleur **boucle à l'infini** et appelle les Réflexes un par un.

• Le réflexe actif calcule pendant un certain temps puis rend le controle au schéduleur.

• Une fois par itération (au moins), le schéduleur appelle **Arbitre** pour faire circuler (transmettre) les commandes et résudre les conflits (cf. noeud S).

1.7.3.3 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 Eccape à une MEF de moteur (controle les moteurs) :

Connecter (les entréess 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).

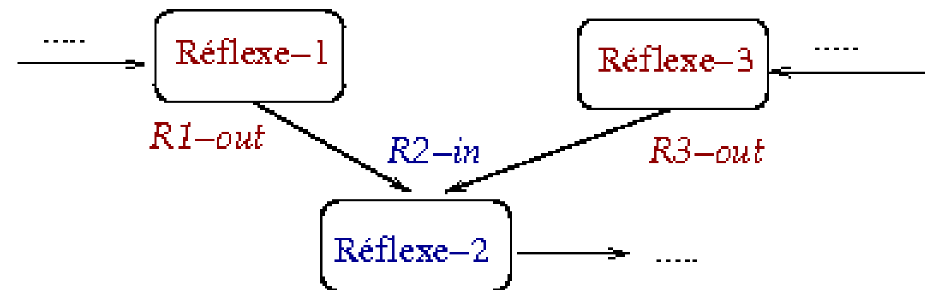
• La fonction Arbitre transmet la commande émise par Escape aux moteurs.

• La même commande émise peut être transmise en entrées de plusieurs autres. On implante le noeud S (supress) en **ordonnant** ces connexions.

Exemple :

Connecter :
Sortties : **Réflex-1**, Sortie-R1 *émise par Réflexe-1*
Entrées : **Réflex-2**, Entrée-R2 *vers Réflexe-2*

Connecter :
Sorties : **Réflex-3**, Sortie-R3
Entrées : **Réflex-2**, Entrée-R2



• Dans la figure ci-dessus, si dans la même itération du schéduleur, 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 neoud S par un simple tri/ordre).

... Suite ...

• Une commande exclue une autre ; une commande peut s'additionner à une autre,

➥ Exemple : reculer + touner ou même reculer + avancer ...

➤ Dans ce cas, l'arbitrage prendra une *décision* en eximinant toutes les commandes à destination d'un même réflexe.

• **Introduction de la notion de temps :**

➤ si un passge du schéduleur 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.

... Arbitrage : Suite ...

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

➤ Le principe du *subsumption* de Brook utilise aussi un 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).

Suite : Préemptif

1.8 Solution Système 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 schéduleur ; ce qui rend la programmation plus difficile.

- N.B. : les cartes embarquées n'ont pas forcément un dispositif multi-tâche.

- Suite : le principe préemptif disponible dans la plupart des sytèmes.

  ➤ Le schéduleur 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.

1.9 Multi-Tâche Préemptif...

- On reprend l'exemple de LEDs clignotantes.
- L'OS permet de lancer des **processus** qui exécutent un programme tel que :

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

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

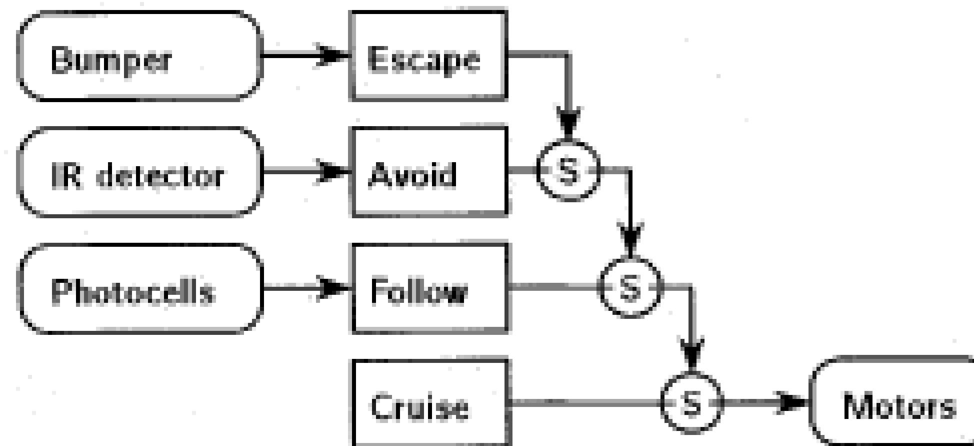
- Après ce lancement, on peut exécuter en parallèle d'autres commandes et processus, pendant que les LEDs clignotent.

- Pour supprimer ce processus par l'OS ou par un autre processus (sous conditions) :

  kill_processus(id) ;

1.9.1 Implantation de R3

- On implante R3 (légèrement modifié) :



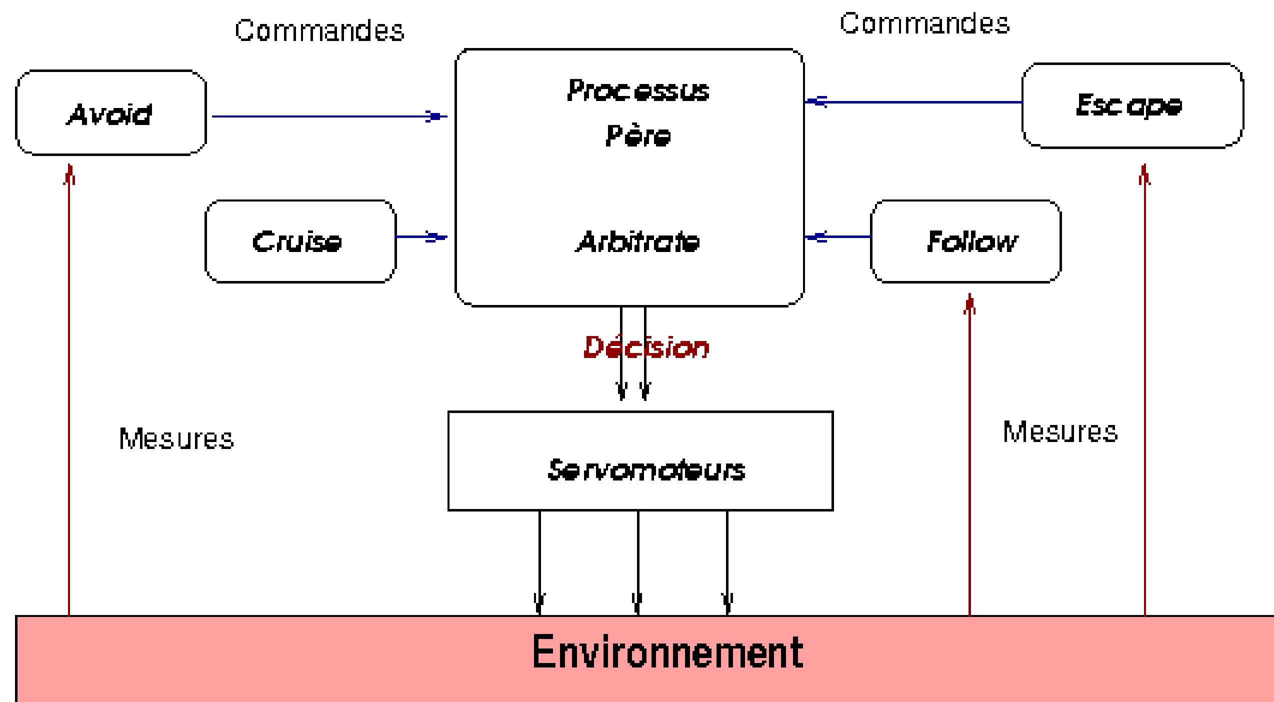
Le principe de fonctionnement :

➤ 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

1.9.2 Architecture du système



Les 6 processus parallèles

1.9.2.1 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_command = 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).

1.9.2.2 Code de Follow

- Rappel : suivre 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;     // Déactiver (par de différence)
  }
}
```

• 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*.

1.9.2.3 Code de Avoid

• Rappel : lire 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;               // Déactiver
    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
          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 :

```
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;             // Déactiver
    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)
     }
  }
}
```

1.9.2.4 Code de Escape

• Rappel : éviter les collisions avec les obstacles (détectée par la cinture 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 gache
       usleep(400);
    }
    else if (bump_left) {                           // choc à gauche
       escape_output_flag = True;
       escape_commande = TURN_RIGHT;                 // tourner à droite un peu
       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.

1.9.2.5 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.

1.9.3 Cap

xterm

```
CRUISE
 FORWARD

PHOTOS+FOLLOW
/  (R= 5)
NIHIL_cmd

IRS (+Avoid)
ARC_LEFT  (R= 1)

Bumpers G/D/B
---

  Escape
NIHIL_cmd

Arbitre (Moteur)
ARC_LEFT

Pos & Dir
14/12/SO
```

```
Robot : Attachement  des Zones  fait
>>>Le semaphore 688129 existe deja. Sa valeur actuelle est 1
>>>Le semaphore 720899 existe deja. Sa valeur actuelle est 1
Robot : Attachement  des Zones  fait
>>>Le semaphore 688129 existe deja. Sa valeur actuelle est 1
>>>Le semaphore 720899 existe deja. Sa valeur actuelle est 1
Robot : Attachement  des Zones  fait
>>>Le semaphore 688129 existe deja. Sa valeur actuelle est 1
>>>Le semaphore 720899 existe deja. Sa valeur actuelle est 1
Robot : Attachement  des Zones  fait
 LE PERE VA ATTENDRE LES FILS
>>>Le semaphore 688129 existe deja. Sa valeur actuelle est 1
>>>Le semaphore 720899 existe deja. Sa valeur actuelle est 1
Robot : Attachement  des Zones  fait
```

1.9.4 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 ajouetra à ces processus les deux autres :
  - ➤ Servomoteur
  - ➤ Affichage

1.9.5 *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 messsages* entre les processus.

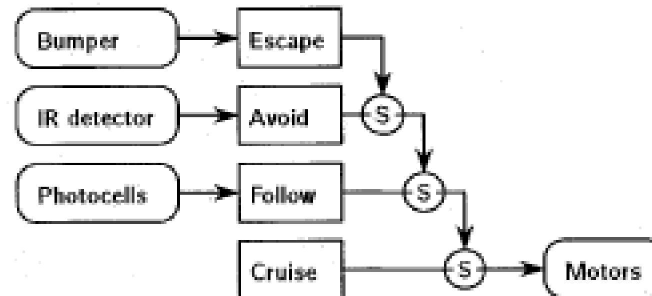
• Une fois les réflexes codés, le diagramme de conception précise comment les sorties et les entrées doivent être connectés.

➤ 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.

Suite (arbitrate) ...

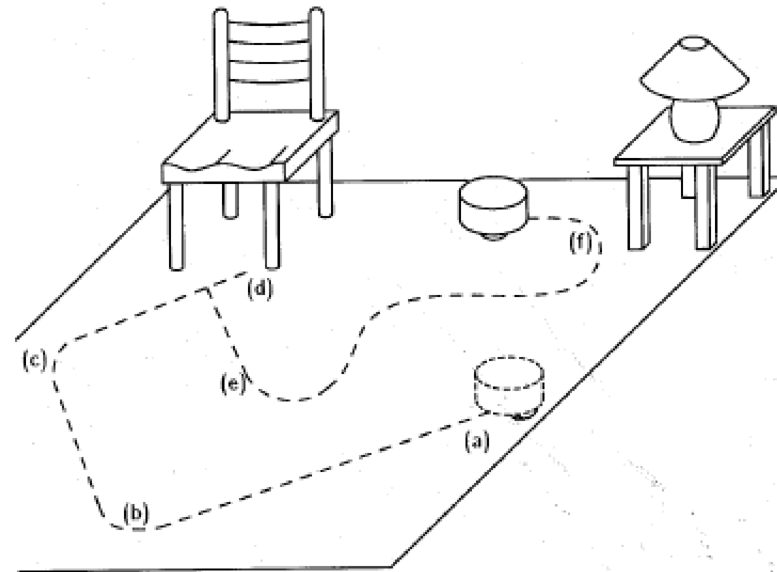
• Rappel du diagramme (dit *cablé*) de conception de R3 :



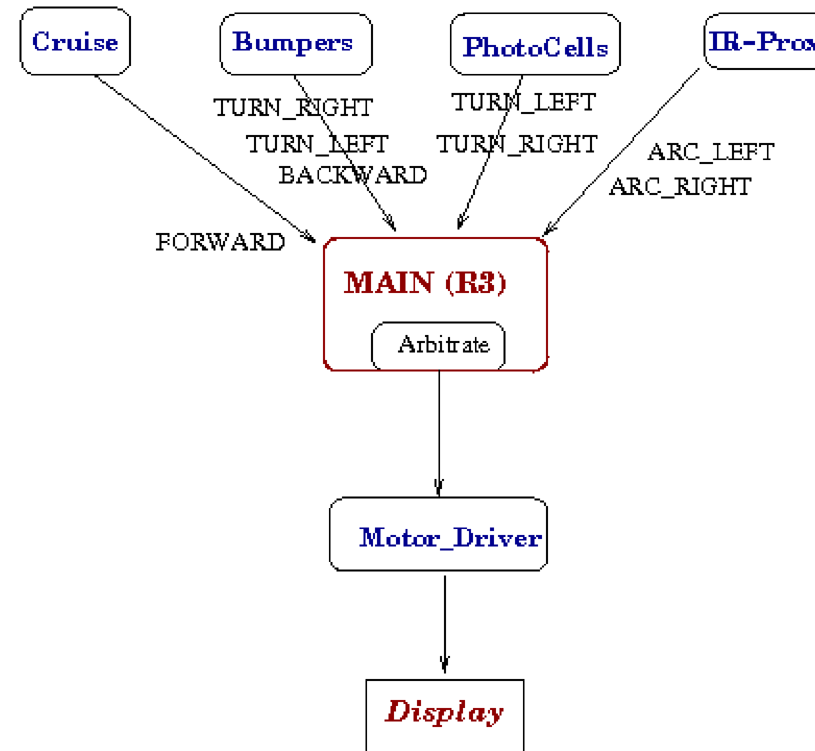

```
void  arbitrate()
{while (1) {
   if (cruise_output_flag == True)
      motor_input = cruise_commande;       // connecter les sorties des
   if (follow_output_flag == True)         // réflexes aux entrées du
      motor_input = follow_commande;       // moteur
   if (avoid_output_flag == True)
 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

```
  }
}
```

- Exemple du milieu :



1.9.6 Schéma Simulation



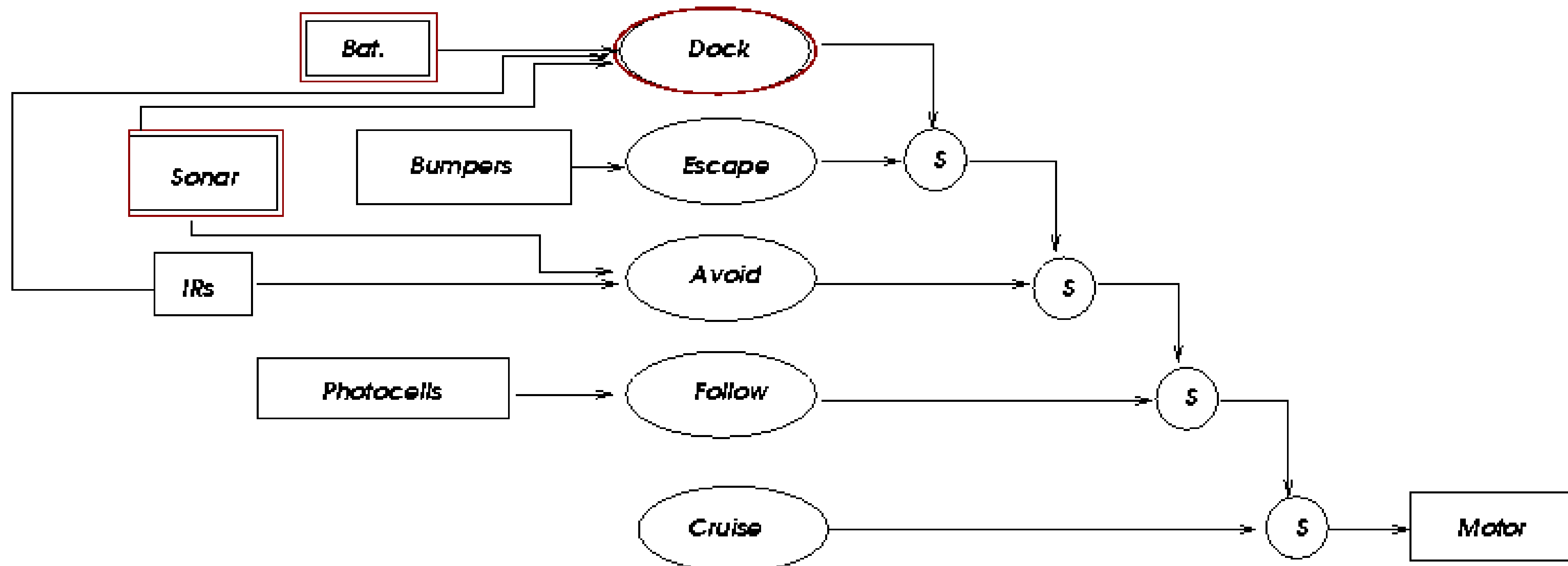
1.10 Programmation

1.11 Pour les PEs

- PE 47 :

- Ajouter le module Duck : si le niveau de batterie devient bas, aller au stand de recharge. La commande deviendra prioritaire sur toutes les autres.

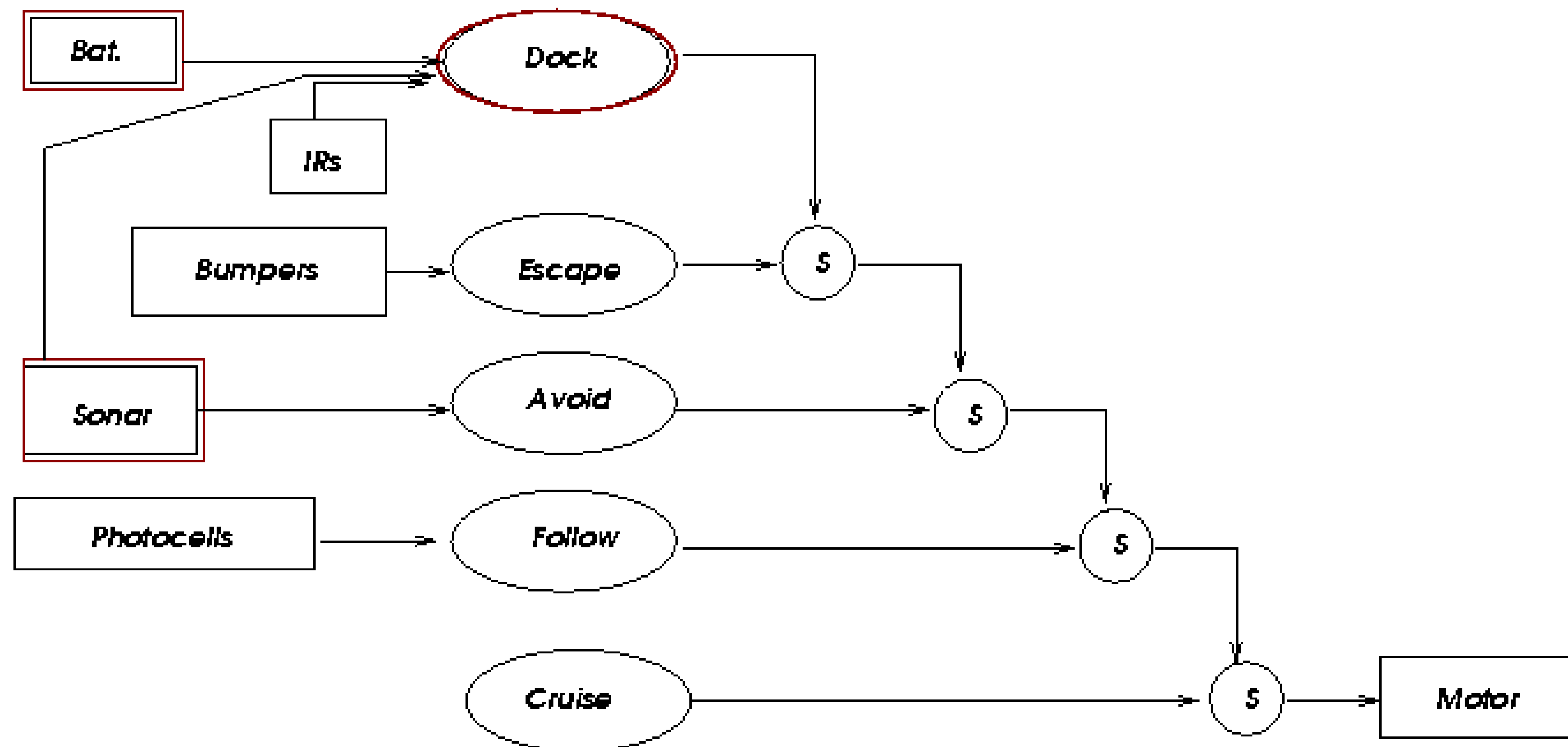
- Le stand est signalé par une balise IR.



1.11.1 PE47 bis

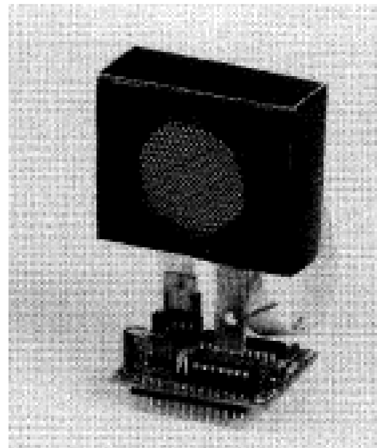
- Un autre schéma :

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



1.11.2 A propos du Sonar

- Rappel : IR permet de détecter une proximité : qq 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 :

```
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 converir 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

}
```

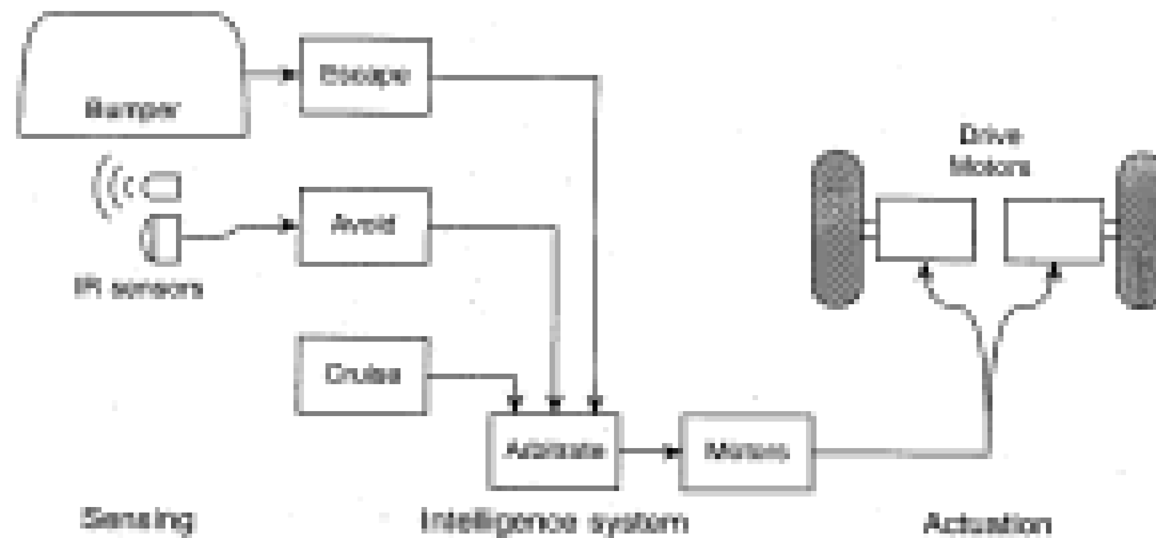
1.12 Quelques Projets

- Quelques projets Robotiques intéressants.

1. Lewis & Clark : on s'évite !
2.

1.12.1 Lewis & Clark

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



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 prgrammer (simulation) ce ensemble.

Autres : à suivre.

1.13 Déplacement Robot

- A*
- Monte Carlo
- ... etc

1.14 Eviter un obstacle mobile (un autre Robot)

Table des matières

1.1 Systèmes Embarqués 2

1.1.1 Caractéristiques principales d'un système embarqué 4

1.1.2 Embarquée : un marché énorme et croissant 5

1.1.3 Les domaines de l'embarqué 7

1.1.4 Un système embarqué typique (figure) 9

1.1.5 Aperçu des paradigmes **SPA** (Sense, Plan, Act) 10

1.1.6 Environnement d'embarqué 14

1.1.7 Les Contraintes de temps dans un système embarqué 16

1.1.8 Eléments de Conception d'Embarqué 17

1.1.9 Systèmes Embarqués Libres 18

1.2 Temps Réel 19

1.2.1 Le temps dans les RTOS 21

1.2.2 Gestions du temps dans un RTOS 23

1.2.3 Les critères des RTOS 24

1.2.3.1 Prévisibilité 25

1.2.4 Fonctionnement d'un RTOS 27

1.2.5 3 Approches en conception des RTOS 28

1.2.6 Multitâche préemptive 29

1.2.6.1 Quelle est la forme de ces événements ? 31

1.2.6.2 Exemples 32

1.2.7 Mécanismes de Synchronisation 33

1.2.8 Sémaphores 34

1.2.9 Code d'un sémaphore 35

1.2.9.1 Sémaphores Binaires 36

1.2.9.2 Sémaphores pour partage de ressources 37

1.2.10 Gestion d'accès en exclusion mutuelle 38

1.2.11 Par un sémaphore 38

1.2.12 Par Test and Set (TAS) 39

1.2.13 Utilisation et exemples de sémaphores 42

1.2.14 Exemple d'utilisation de sémaphores pour la synchronisation 43

1.2.15 Un exemple en C/C++ 44

1.2.16 Communication entre tâches 45

1.3 Le rôle du noyau de RTOS 46

1.3.1 Le diagramme d'états d'une tâche 47

1.3.2 Gestion des Priorités (des tâches) 49

1.4 Processus et threads 53

1.5 Temps Réel sous Linux = Temps partagé 54

1.5.1 Quelques RTOS Posix 56

1.6 Programmation Robot 57

1.6.1 Approches de programmation Robot 58

1.6.2 Approche centralisée (planification) 59

1.6.2.1 Exemple Handey 61

1.6.3 Approche décentralisée à base de réflexe (cybernétique) 65
1.6.3.1 R1 : un Robot simple 67
1.6.3.2 R2 : Recharge de batterie 70
1.6.4 R3 : un Robot plus évolué 72
1.6.4.1 Un exemple de parcours pour R3 77
1.7 Implantation 78
1.7.1 Processus et Scheduleurs 79
1.7.2 MEF 82
1.7.3 Formalisme de contrôle des réflexes 83
1.7.3.1 Le code d'un module Réflexe 86
1.7.3.2 Le scheduleur 87
1.7.3.3 L'arbitre 88
1.8 Solution Système Préemptif 92
1.9 Multi-Tâche Préemptif... 93
1.9.1 Implantation de R3 94
1.9.2 Architecture du système 95

1.9.2.1 Code de Cruise 96

1.9.2.2 Code de Follow 97

1.9.2.3 Code de Avoid 99

1.9.2.4 Code de Escape 102

1.9.2.5 Code du Servomoteur 104

1.9.3 Capture d'écran du simulateur 105

1.9.4 Le Processus Père : la fonction main() 106

1.9.5 La fonction Arbitrate() 108

1.9.6 Schéma Simulation 111

1.10 Programmation 112

1.11 Pour les PEs 113

1.11.1 PE47 bis 114

1.11.2 A propos du Sonar 115

1.12 Quelques Projets 118

1.12.1 Lewis & Clark 119

1.13 Déplacement Robot 121

1.14 Eviter un obstacle mobile (un autre Robot) 121