Systèmes embarqués 2
INFO 941

Sébastien Monnet
Polytech Annecy-Chambéry

Basé sur le cours de Guillaume Ginolhac

Organisation du module

- 6 Cours de 1h30
- 1 TD de 1h30
- 6 TP de 4h (mini projet)

- Examen final : 1h30

Supports de cours

- Transparents disponibles sur EAD
- Attention :
  - **Support** de cours (trame du cours)
  - **Nécessité** de prendre des notes

Contenu du module

- Objectifs
  - Fonctionnement d'un **système**
  - Spécificités des systèmes **embarqués**
  - Spécificités des systèmes embarqués **temps-réel**
  - Programmation par interruptions
  - Programmation multitâches

Chapitre 0
Rappels

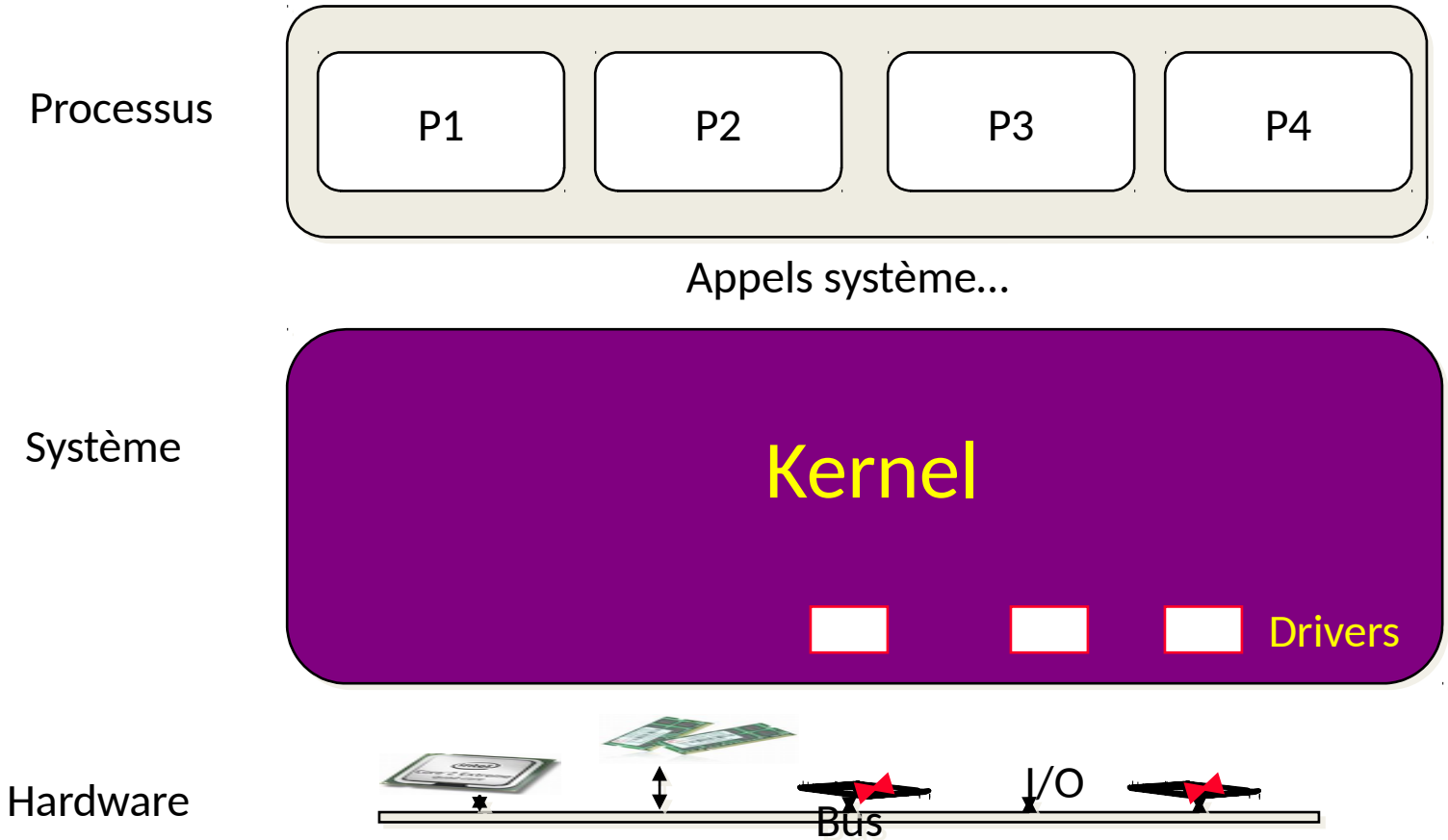
système d’exploitation

Processus

Fonctionement de la pile

La place du système

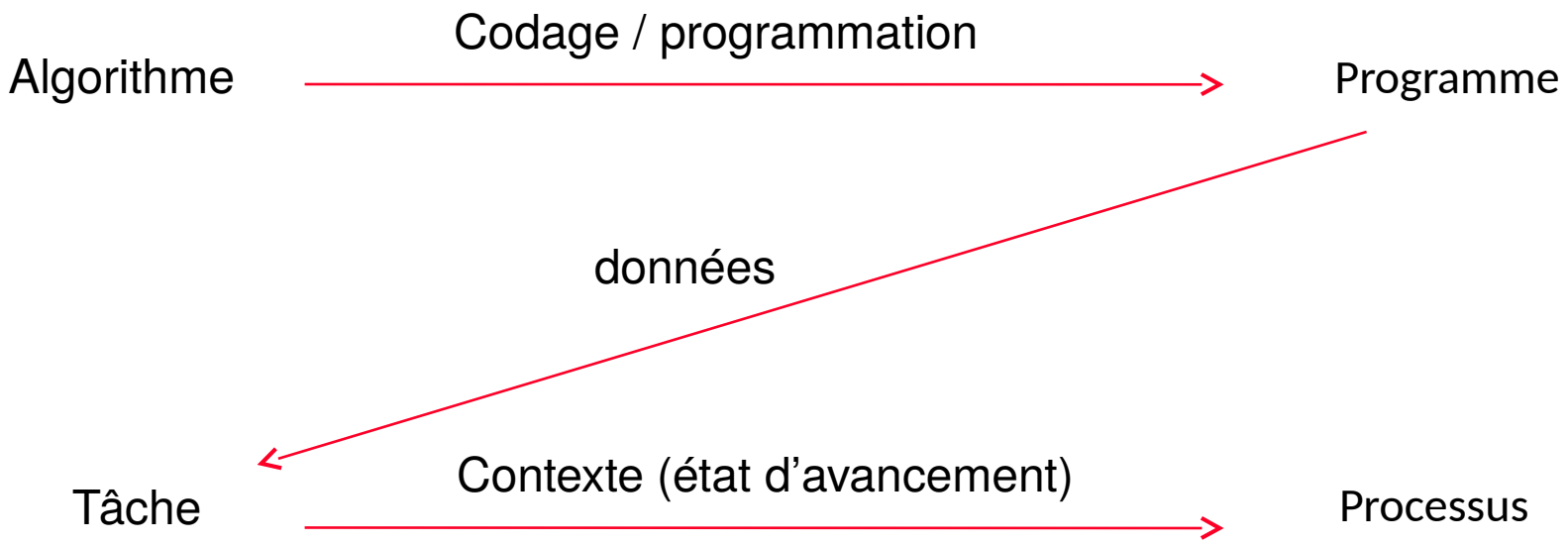
- Entre matériel et applications



Rôle du système

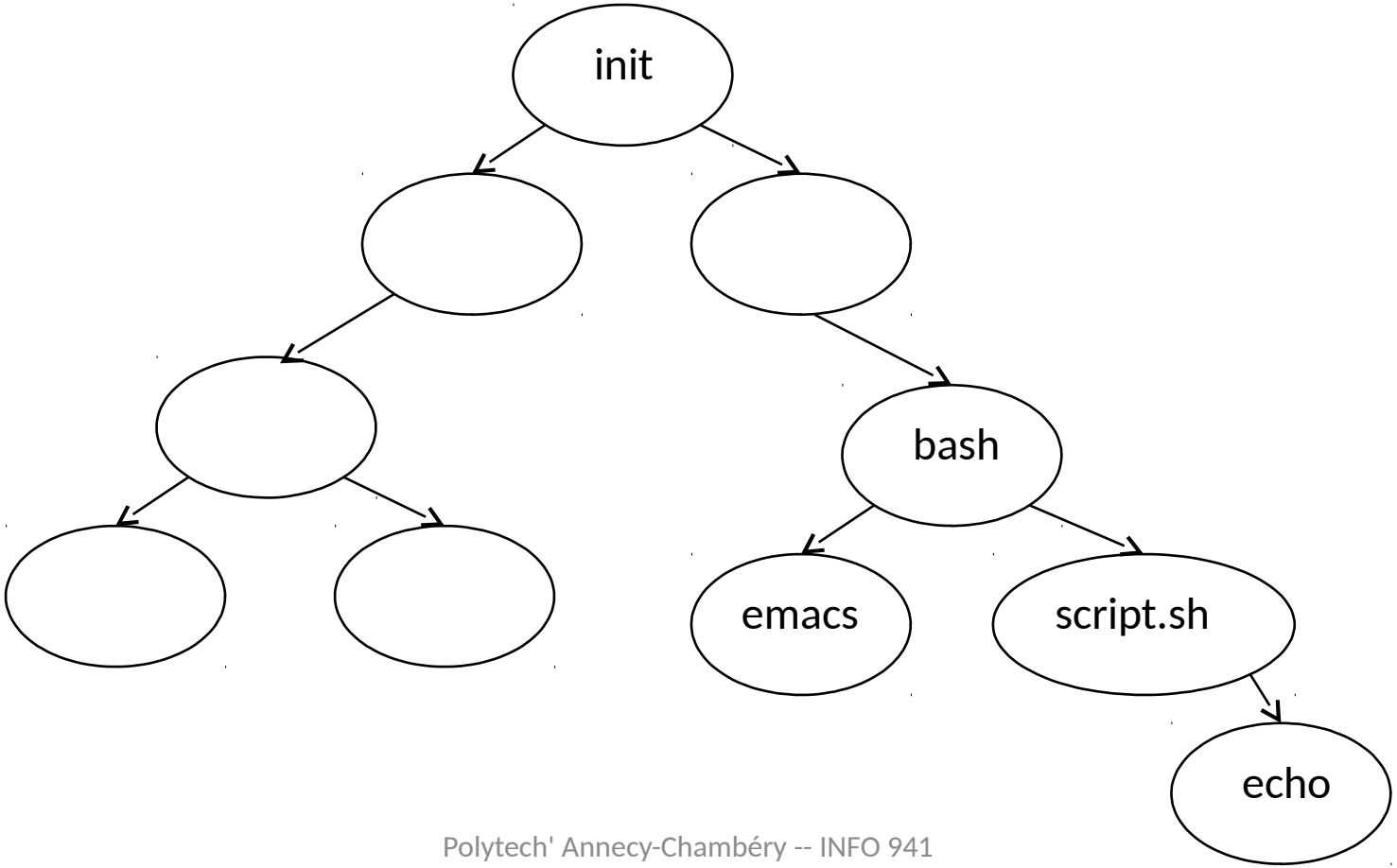
- Accès au matériel
- Abstraction
  - Exemple : systèmes de fichiers
- Partage “équitable” des ressources
  - Partage du processeur
  - Partage de la Mémoire
  - …
- Mécanismes de synchronisation, de communications
- Gestion des utilisateurs, processus,…

Programmes, taches, processus



Filiation de processus

- Héritage de contexte

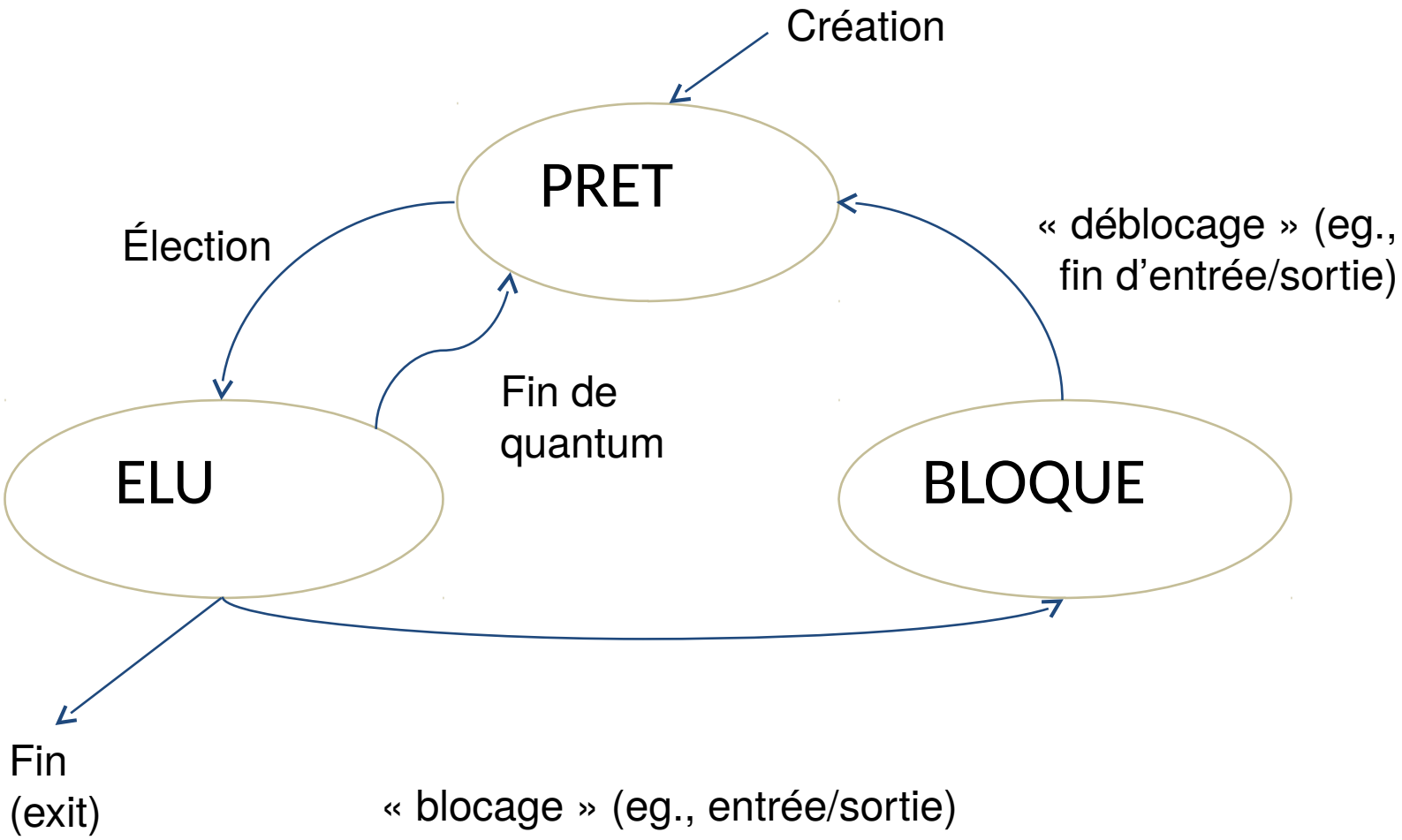


Exécution d’un processus

- Segments de mémoire
  - Code, données, pile
- Registres CPU

- Exécution des instruction
  - Séquentiellement, PC (*program counter*)
  - Sauts
  - Importance de la pile (appels de fonctions)

Etat d'un processus



Partage du processeur

- Mécanismes système
  - Election
    - Choix du prochain processus
  - Commutation
    - Remplacement du processus en cours d'exécution
  - Notion de quatum
    - Durée d'utilisation maximale

Election

- But : choisir le processus qui va avoir accès au processeur
  - Parmi les processus prêts
- Doit être simple et efficace
  - Exécution fréquente
  - Eg., une file d'attente
- Peut prendre en compte des priorités entre processus
  - Très utile en pratique
  - **Tâches temps-réel**
- Ne doit pas engendrer de famine
  - Tout processus prêt doit avoir accès au processeur en un temps fini

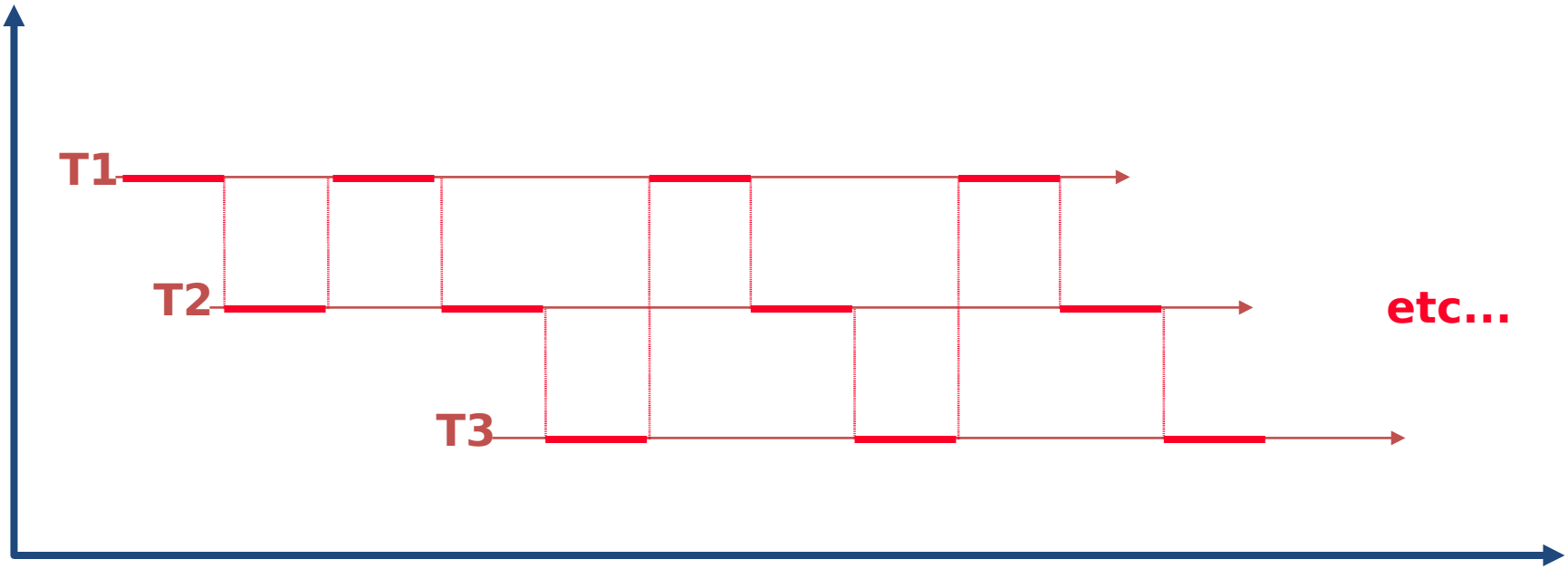
Commutation

- Remplacer le processus en cours d'exécution par celui qui vient d'être élu

- Sauvegarder le processus en cours (son contexte)
  - L'état de la mémoire allouée au processus
    - Mémoire virtuelle
  - L'état des registres / de la pile
  - Le compteur ordinal (où en est l'exécution ?)

- Restaurer le processus qui vient d'être élu avec sa dernière sauvegarde

Example d'exécution multitâche



Le "chemin" du compteur ordinal du processeur

Commutation invisible pour les processus

- Pour un processus, tout se passe comme si il conservait le processeur:
  - L'exécution reprend exactement là où elle s'était arrêtée
  - Les variables ont les même valeurs
  - Les commutations peuvent avoir lieu n'importe où dans le code
  - Le processus n'est pas prévenu

Commutations : quand ?

- Quand le processus en cours d'exécution abandonne le processeur
  - Exit
  - Sleep
  - Entrée / sortie bloquante, synchronisation
  - Attente de la fin d'un processus fils (wait)

- Le système « reprend la main »
  - Déclenche une élection
  - Réalise l'échange de processus (la commutation)

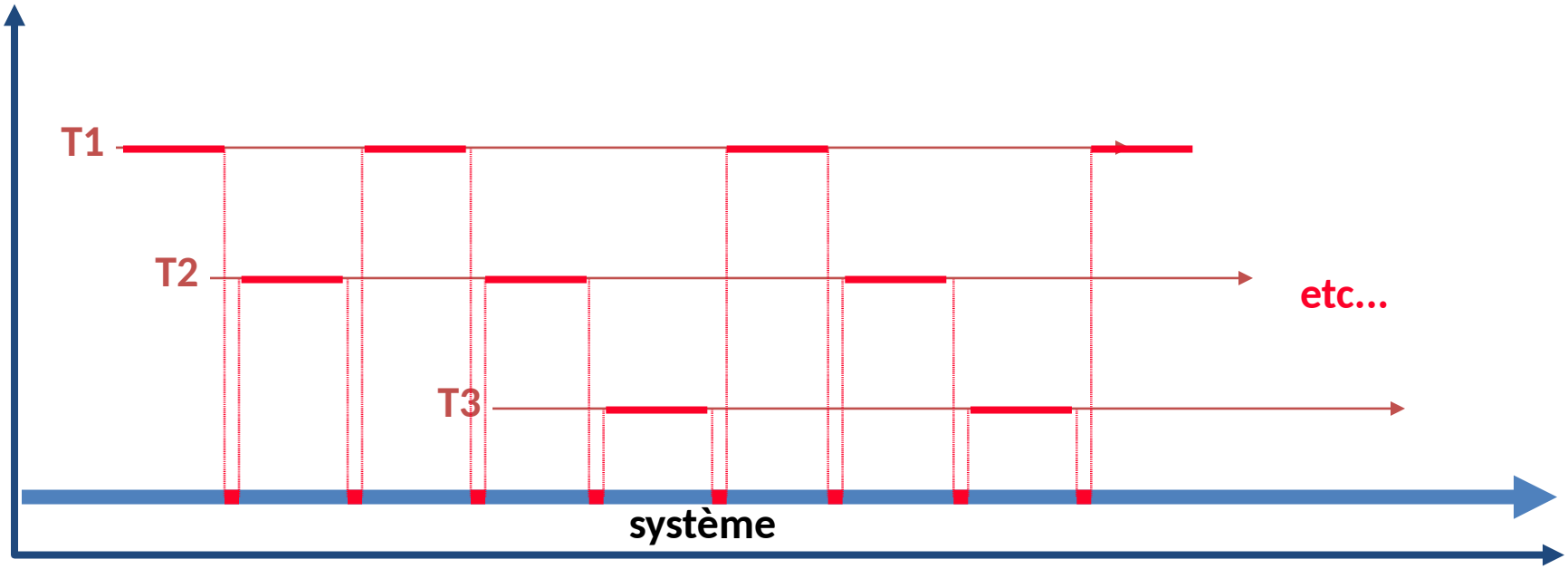
➢ Insuffisant pour assurer un pseudo-parallélisme
  - Notion de quantum / temps maximal sur le processeur

Notion de quantum

- Il existe un mécanisme matériel qui permet au système de reprendre la main régulièrement
  - Interruption horloge
  - Du code système sera exécuté périodiquement
  - Assure la vivacité du système (un processus ne peut pas accaparer le processeur)

- Quand un processus a utilisé le processeur un quantum de temps
  - Une commutation est déclenchée

Le système « reprend la main »



L’ ordonnanceur

- Mécanisme de commutation + mécanisme d’élection

- Politiques
  - Sans notion de quantum
  - Avec notion de quantum : temps partagé
  - Avec/sans réquisition

  a) sauvegarder la tâche qui perd le processeur

  b) mettre cette tâche en attente (éventuellement, modifier sa priorité)

  c) lancer un processus d'élection (prenant en compte les priorités ou non)

  d) restaurer le contexte du processus élu et se brancher à l'endroit où il s'était arrêté (PC)

Multitache

- Le système permet
  - L'accès aux ressources matérielles
  - Le partage des ressources
    - Le partage du processeur !

=> Multiprogrammation

- Exécutions concurrentes (entrelacements d'exécutions séquentielles)
- Exécutions parallèles (multi-cœurs, multi-processeurs)

⇒Possibilité de travail collaboratif

- Communications
  - Signaux
  - Ecrivain - > lecteur (synchronisation faible)
  - Pipes
  - …

Ressource partagée

- Afin de collaborer, les processus doivent se partager des ressources
  - Tableaux, matrices, listes, fichiers, structures complexes, etc.
- Schémas d'accès diverses et variés
  - Écritures / mises à jour potentiellement concurrentes !
- Besoin de cohérence
  - Le résultat ne doit pas dépendre de l'ordre d'exécution (ordonnancement des processus non déterministe !).

Deux programmes à exécuter

Considérons l'exécution des deux programmes suiva

Soit fic un fichier contenant un entier

A <- contenu de fic	B <- contenu de fic
A <- A + 10	B <- B - 10
Remplacer le contenu de fic par A	Remplacer le contenu de fic par B

Exécution 0

Processus 1	A	fic	B	Processus 2
A <- contenu de fic	10	10		
A <- A + 10	20	10		
A -> fic	20	20		
	20	20		B <- contenu de fic
	20	10		B <- B - 10
	10	10		B -> fic

Résultat : fic contient 10

Exécution 1

Processus 1	A	fic	B	Processus 2
A <- contenu de fic		10	10	
A <- A + 10		20	10	
	20	10	10	B <- contenu de fic
	20		10	0 B <- B - 10
	20		0	0 B -> fic
A -> fic			20	20

Résultat : fic contient 20

Exécution 2

Processus 1	A	fic	B	Processus 2
	10	10		B <- contenu de fic
	10	0		B <- B - 10
A <- contenu de fic	10	10	0	
A <- A + 10	20	10	0	
A -> fic	20	20	0	
			0	0 B -> fic

Résultat : fic contient 0

Chapitre 1
Systèmes embarqués

Définition systèmes embarqués

- « Un ordinateur qui ne ressemble pas à un ordinateur »
- Interagie avec l'extérieur
- Pas ou peu d'interface utilisateur (systèmes enfouis)
- Fait partie d'un produit complet qui répond à des besoins
- SOC : System On Chip
  - Système complet capable de réaliser une tache
  - En général SOC = 1 ou + processeur+ROM+ E/S

Motivations : Systèmes Embarqués

- Miniaturisation processeurs
- Dans tous les objets de la vie courante
  - Consoles de jeux vidéos
  - Photos, Télévision Digitales
  - Assistants personnels, Set-top-box
  - Informatique dans les transport : voiture
  - Outils de communications de l'information : GSM
  - Santé: implants, aide personnes handicapées, etc.
- Mais aussi
  - Production d'énergie (nucléaire)
  - Avionique
  - …

Différences SE / système classique

- Basse consommation
  - exemple : PDA , GSM, Etiquette électro. (tag)

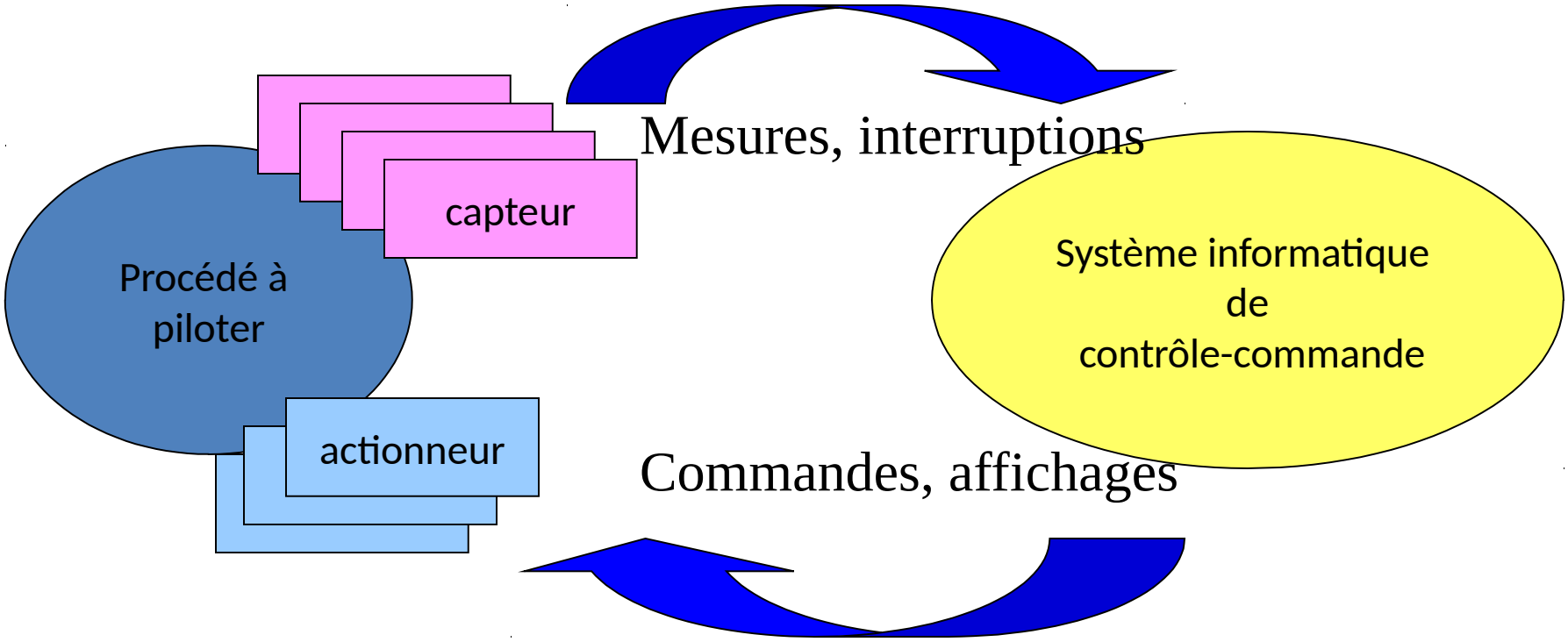
- Il faut trouver le bon compromis entre vitesse et consommation d'énergie
  - Le SE le plus rapide n'est pas obligatoirement = recherché

- Système temps réel (Real Time RT):
  - La plupart des SE sont RT
  - Vitesse liée aux performances RT

Systèmes embarqués : résumé

Un système informatique embarqué

est un système de contrôle-commande



Systèmes embarqués : résumé

- Un système informatique doit pouvoir :
  - **Communiquer avec l'extérieur** : *capteur, actionneur, …*
  - **Exécuter des instructions** : *processeur*
  - **Conserver des informations** : *mémoire*

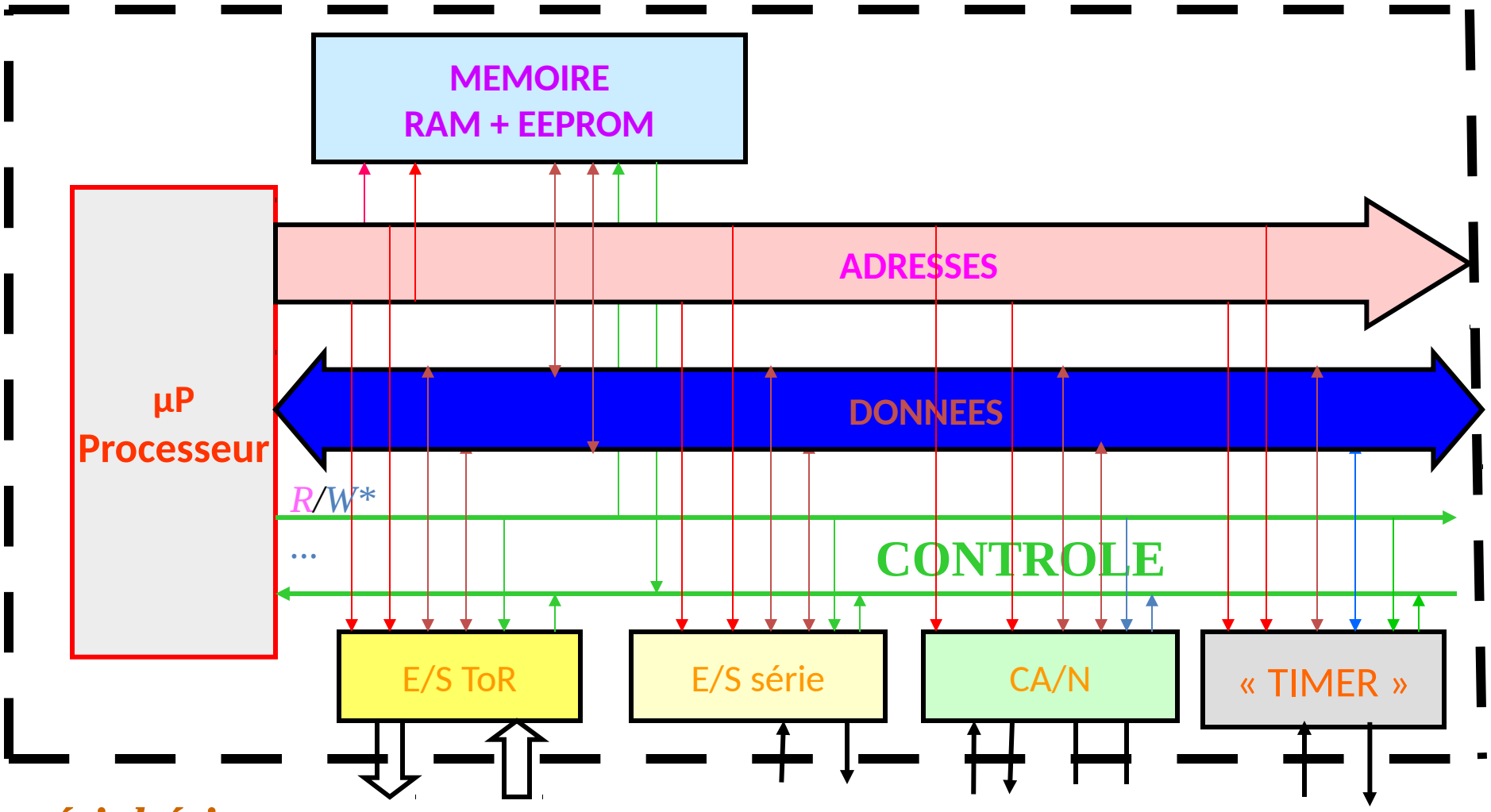
Schéma général

- Un système informatique est composé de circuits :
  - **processeur(s)**
  - **Mémoire**
  - coupleurs d'**Entrées/Sorties**

- reliés entre eux par des **bus** :
  - **d'adresses**
  - **de données**
  - **de contrôle** (commande et état)

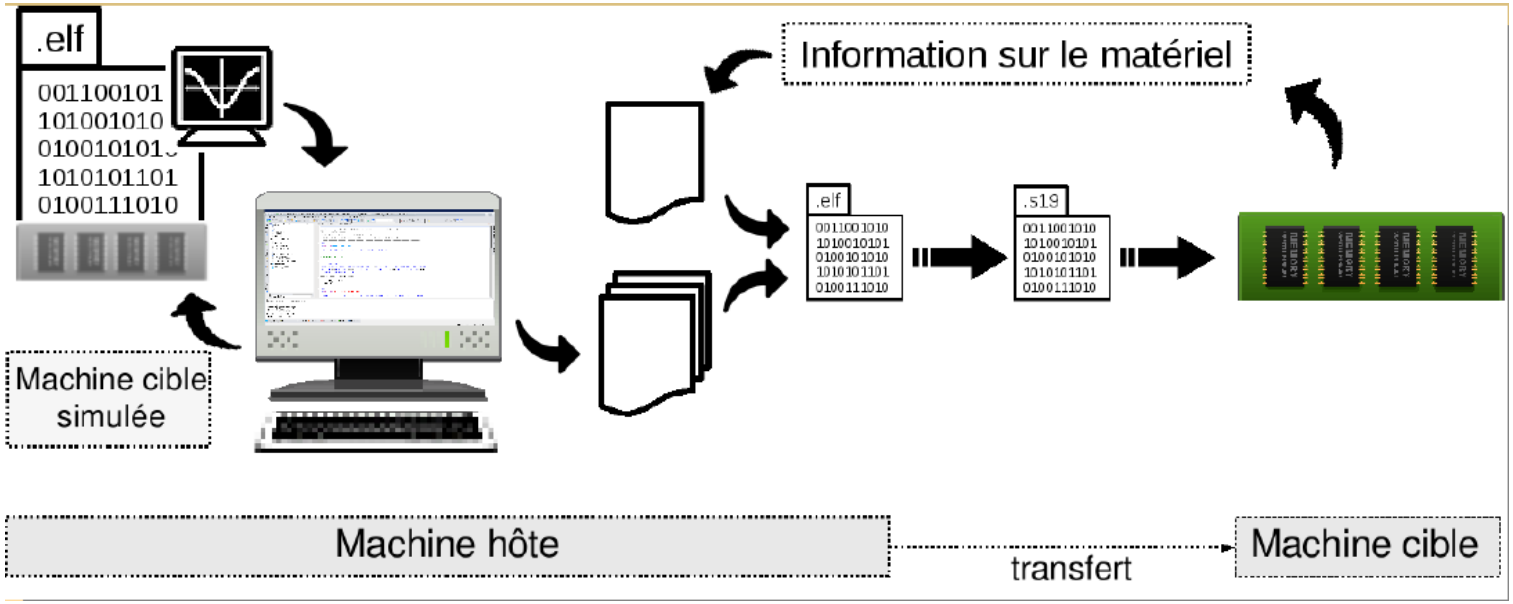
- Quand ces circuits sont intégrés sur une même puce
- on parle de **microcontrôleur (µC)**.

Schéma général (suite)



Développement sur un système embarqué

- Différence entre machine hôte / machine cible
- Développement sur machine hôte
- Test/validation avec machine cible simulée
- Cross compilation : jeu d'instruction / configuration matérielle
- Transfert vers machine cible



Programmation sans OS

- Pas d'OS sur de nombreux SE (coût, lourdeur, …)
- Programmation proche du matériel : optimisation temps, mémoire
- Mettre en place votre environnement de développement :
  - Choisir un langage de développement (Assembleur, C/C++)
  - Trouver compilateur / linker / simulateur
  - Datasheets du matériel
- Gérer les entrées / sorties
- Choisir son mapping : où placer les différentes informations dans la mémoire RAM
- Bien comprendre / utiliser les registres :
  - Configuration
  - Utilisateurs

=> Refaire une partie du travail d'un OS

Programmation avec OS

- Intéressant pour les programmes lourds contenant beaucoup de tâches
- Ordonnanceur intégré, gestion des priorités
- Abstraction/accès au matériel : drivers intégrés
- Exemple : linux (embarqué et/ou temps réel)

Exemples de SE

- **Microcontrôleurs** (comme arduino) :
  - Le SE le plus courant
  - Comprend un µP, des bus, de la ROM et des E/S
  - pas d'OS
- Avantages : pas cher, faible encombrement et faible consommation
- Inconvénients : énormément de choix (avec plus ou moins de périphériques), traitements plutôt léger

Exemples de SE

- **Mini-Ordinateurs** (ARM) :
  - Exemples :Raspberry Pi (processeur ARM), plateforme Armadeus (processeur ARM + FPGA)
  - OS Linux Embarqué
- Avantages : puissance de linux (drivers, ordonnanceur intégré, gestion de la mémoire…)
- Inconvénients : consommation, moins de liberté (dans la programmation, la gestion de la mémoire)
- Nécessite de très bonnes connaissances de Linux

Exemples de SE

- **FPGA** (basé sur des portes logiques) :
  - Microprocesseurs reconfigurables
  - Programmation en VHDL (possible d'autres langages mais sans garantie de compilation)
- Avantages :
  - Circuits flexibles permettant des évolutions, plus performants qu'un logiciel, moins chers que des ASICs
- Inconvénients :
  - Développement longs pour des applications compliquées, gourmands

Exemples de SE

- **DSP** : Digital Signal Processing
- SE conçu pour les applications en Traitement Signal / Images / Vidéo
- Exemples : Télévision, amplificateur HiFi, Routeurs, Radar/Sonar, ...
- Avantages :
  - précision, prédiction (simulations sur ordinateur),
  - bibliothèques de calcul (fft, ...),
  - Cycle Multiplication/Accumulation (MAC) : A = A+(X.Y) : 1 cycle au lieu de 4 cycles
- Inconvénients :
  - Coût élevé (inutile pour des réalisations simples),
  - Complexité (optimisation du calcul)

Chapitre 2
Systèmes embarqués – temps réel

Système temps réel

- Le temps réel (physique) entre dans le critère de correction des applications en plus de leur correction fonctionnelle

JUSTE + RETARD = FAUX

- Besoin de garanties sur le respect de contraintes temporelles (prévisibilité)

- Définition d'une contrainte de temps :
  - limite quantifiée sur le temps séparant deux évènements (limite min et/ou max)
    - Quantifiée = en rapport avec le temps réel (environnement extérieur)
    - Ex : échéance de terminaison au plus tard (deadline) = limite maximale entre arrivée d'un calcul (tâche) et sa terminaison

Classes de systèmes temps réel

- Temps-réel strict/dur (hard real-time) : le non respect d'une contrainte de temps a des conséquences graves (humaines, économiques, écologiques) : besoin de garanties
  - Ex : applications de contrôle dans l'avionique, le spatial, le nucléaire, contrôle de production de matériaux toxiques
  - On se mettra par défaut dans ce cadre
- Temps-réel souple/mou (soft real-time) : on peut tolérer le non respect occasionnel d'une contrainte de temps (garanties probabilistes)
  - Ex : applications multimédia grand public (vidéo à la demande, TV)

(Source : cours Isabelle Puaut – IFSIC, Rennes)

Vers une programmation par tâches

- Un système RT est nécessairement multitâche
- Un système RT est un système dit préemptif : tâche en cours peut perdre involontairement le processeur au profit d'une autre tâche (jugée plus urgente)
- Un système RT doit savoir prendre en compte des interruptions (matérielles ou logicielles)
- Il faut gérer :
    - création, ordonnancement de tâches
    - synchronisation + protection ressources
    - communications
    - mémoire
    - temps (délai, activation périodique)
    - périphériques

Caractéristiques Systèmes Temps Réel

Gestion en général de 2 ensembles de taches :

1° les taches **périodiques**, « pilotées par le temps », exécutant des fonctions de contrôle avec des **contraintes de temps strictes** ;

2° les taches **apériodiques**, « pilotées par événements » et pouvant avoir des **contraintes de temps strictes, relatives ou inexistantes**.

Objectifs :

- garantir les échéances des taches à contraintes strictes dans les pires conditions ;

- donner un temps de réponse moyen correct aux autres taches.

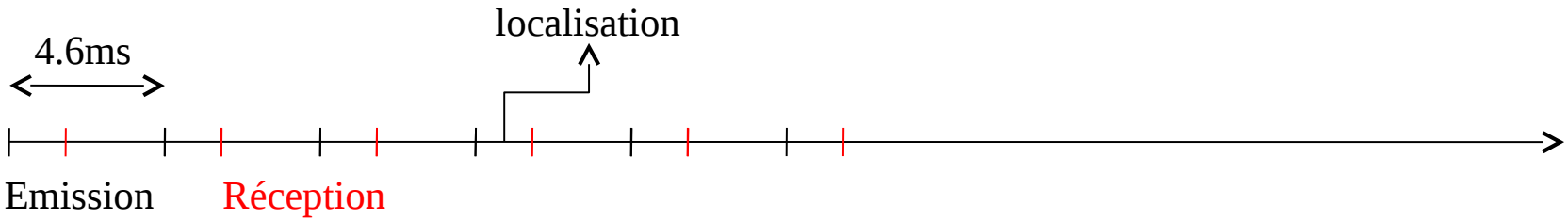
Exemple: téléphone mobile

Fonctions à remplir :

- émettre et recevoir les signaux de paroles

  577µs de parole émise chaque 4,6ms

  577µs de parole reçue chaque 4,6ms ≠ de l'émission ;
- localiser en permanence le relai le plus proche et synchroniser les envois par rapport à cette distance ;
- des comptes rendus de la communication doivent être émis périodiquement (qques secondes) ;
- des applications diverses (agenda, photos, …).



Solution mono-tache : exécutif cyclique

- Taches à effectuer découpées en procédures.
- Création table d'appels de procédures.
- Programme principal : boucle permanente périodique.
- Taches exécutées dans l'ordre prévu : attention au respect de toutes les échéances.

EXEMPLE (temps exprimé en tickets d'horloge)

tache	période τ	temps d'exécution C
A	**25**	**10**
B	**25**	**8**
C	**50**	**5**
D	**50**	**4**
E	**100**	**2**

Ex : Exécutif cyclique

Mise en œuvre par interruption toutes les 25 ticks d'horloge

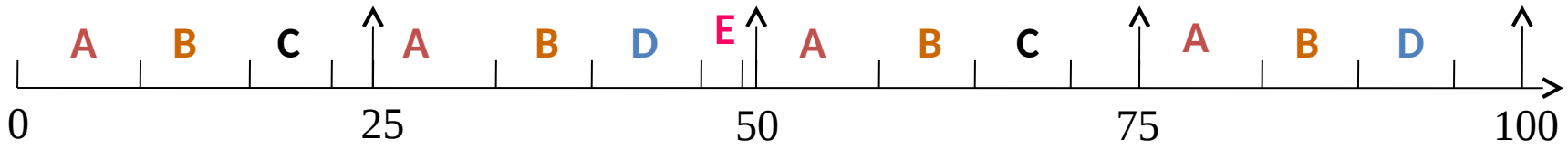
Répéter (1°, 2°, 3°, 4°) à l'infini

1° attendre IT
- **procédure A (10)**
- **procédure B (8)**
- **procédure C (5)**

2° attendre IT
- **procédure A (10)**
- **procédure B (8)**
- **procédure D (4)**
- **procédure E (2)**

3° attendre IT
- **procédure A (10)**
- **procédure B (8)**
- **procédure C (5)**

4° attendre IT
- **procédure A (10)**
- **procédure B (8)**
- **procédure D (4)**



Caractéristiques de l'exécutif cyclique

1 cycle MAJEUR (répété à l'infini) composé de **N cycles MINEURS** constitués d'appels de procédures.

AVANTAGES :
Simplicité (au permier abord), **déterminisme**, accès séquentiel aux données (**pas de protection nécessaire**).

INCONVENIENTS :
durée du cycle majeur, **difficultés de construction** (découpage en K procédures de durée calibrée) et de prise en compte des taches apériodiques, de changement/évolution du cahier des charges, …

Solution multi-taches : noyau temps réel

Caractéristiques :

- Application découpée en taches à exécuter.

- Toutes les taches en **compétition** pour obtenir le µP.

- **Niveau de priorité** affectée à chaque tache (pas obligatoire).

- **Ajout de contraintes** de synchronisation / protection ressources

- Ordre d'exécution des taches **ordonnanceur** (« scheduler »).

Organisation d'un système multi-taches

Application découpée en **taches autonomes exécutées en parallèle** (pseudo-parallélisme) :

compétition ⇔ *primitives* ***d'exclusion mutuelle***
pour l'accès aux ressources

⇓

coordination ⇔ **algorithme d'ordonnancement**

⇑

coopération ⇔ *primitives de* ***synchronisation*** *et*
pour réaliser l'application *de* ***communication***

Définition d’une tache

Tache (processus):
concept pour représenter des activités se déroulant naturellement en parallèle sans considération de la mise en œuvre réelle;
Possibilité d’associer à chaque tache un processeur dans le cadre de processeur multi-coeur.

Programme classique logiciel séquentiel (un fil d’exécution) :

du code exécutable (programmes)
des données
un espace adressable
un contexte d’exécution (valeur des différents registres du processeur)
un état (en exécution, prête, bloquée, …).

Tache définie dans le système : structure de données spécifique
descripteur de tache (« Task Control Block »)

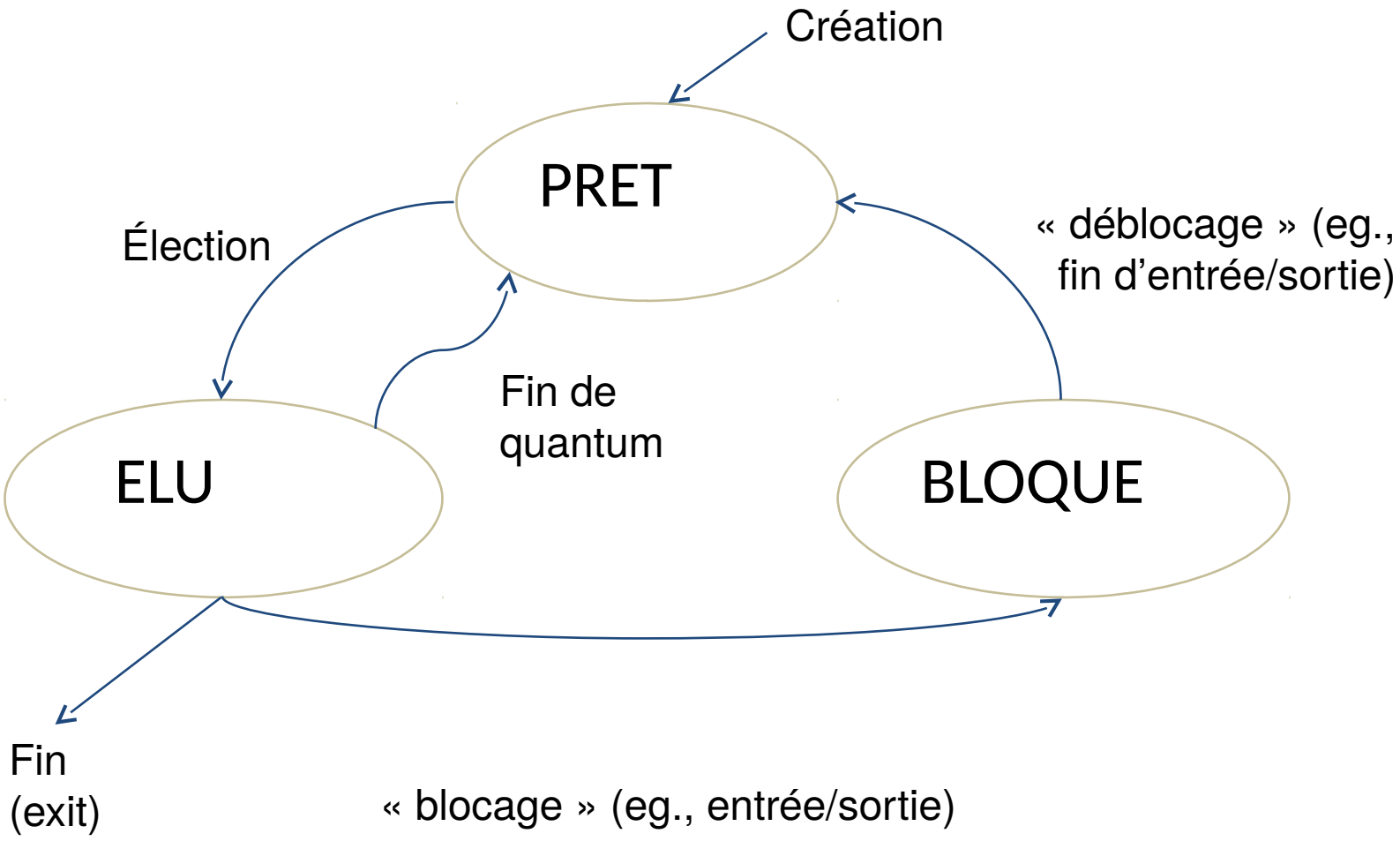
Types de taches

Application comprend 2 ensembles de taches :

1° taches **périodiques**, *« pilotées par le temps », exécutant des fonctions de contrôle avec des* ***contraintes de temps strictes,***
2° taches **apériodiques**, *« pilotées par événements » et pouvant avoir des* ***contraintes de temps strictes, relatives ou inexistantes.***

Objectif d'un système temps réel / multi-tache :
- garantir les échéances des taches à contraintes strictes dans les pires conditions,
- donner un temps de réponse moyen correct aux autres taches.

Etat d'un processus



Chapitre 3
Programmation par interruptions

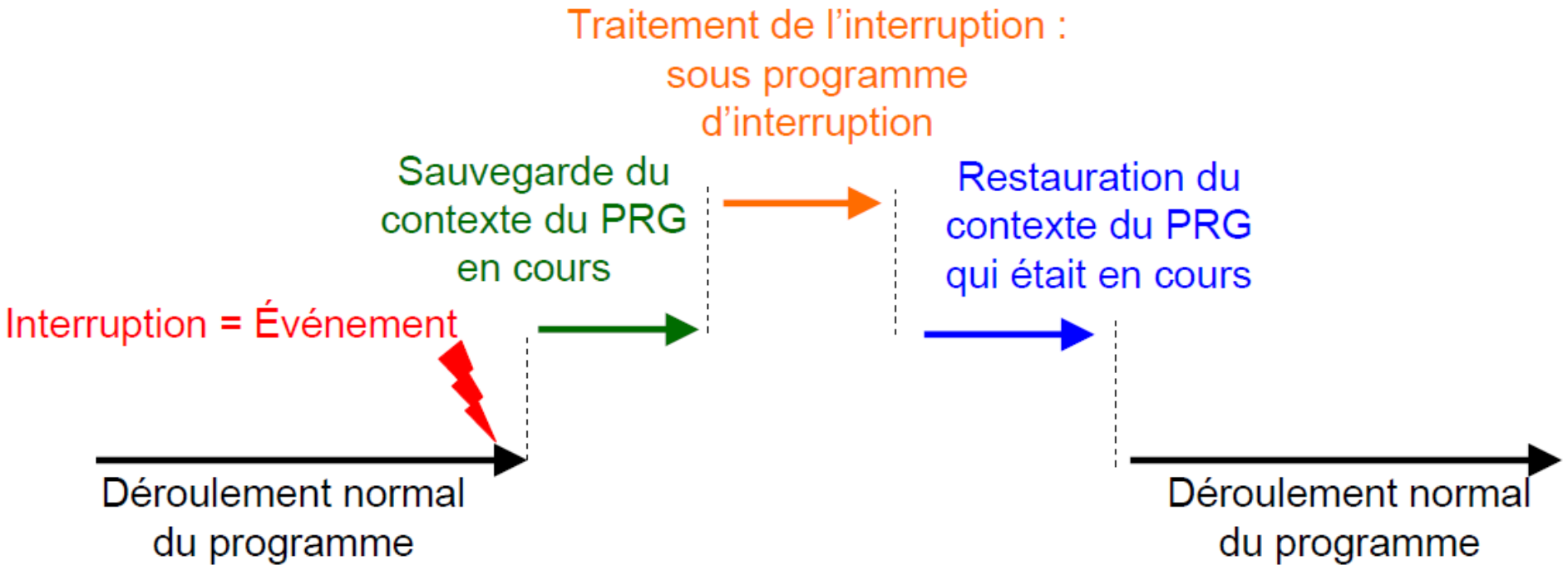
LE PROCESSEUR REPETE SANS ARRET :

- Lire l’instruction pointée par le PC
- Mettre à jour le PC
- Décoder l’instruction
- Exécuter l’instruction
- **Traiter demande d’IT en attente**

(saut à la procédure de service de l’IT)

Définition et déroulement d'une interruption

• Définition : une **interruption est un arrêt temporaire de l'exécution normale d'un** programme informatique par le microprocesseur afin d'exécuter un autre programme (appelé routine d'interruption).



Types d’interruption

- Interruption masquable : un masque d’interruption est un mot binaire de configuration du microprocesseur qui permet de choisir **(démasquer)** quels modules pourront interrompre le processeur parmi les interruptions disponibles :
  - **interruption matérielle** : composant d’entrées/sorties. Il faut alors initialiser les registres de contrôle, les vecteurs d’interruptions et autoriser le niveau d’interruption pour le processeur.
  - **Interruption logicielle :** déclenchée par l’exécution d’une instruction particulière dans le programme

- Interruption non masquable : elles s’exécutent quoi qu’il arrive, souvent avec une priorité élevé (ex : Reset)

Configuration des interruptions

- Un système peut accepter plusieurs sources d'interruption. Chacune est configurable par un registre d'interruption

- Méthode de configuration des interruptions :
  - Sélectionner les interruptions qui nous intéressent
  - Valider les interruptions de façon globale
  - Ecrire le/les sous programme d'interruption
  - Définir les priorités entres interruptions

- Remarque : une procédure de service d'IT étant exécutée à des **moments inconnus à l'avance,** le **passage de paramètres est donc impossible**: *la communication avec les autres progr. se fait par variables globales exclusivement.*

Traitement de la demande d'interruption

- Le processeur exécute les actions suivantes :
  - Envoyer signal d'acceptation de l'IT (IACK).
  - Sauver dans la pile le PC (+ certains reg.).
  - Obtenir l'adresse de la procédure d'IT.
  - Recopier cette adresse dans le PC.

- Dans le sous programme d'interruption :
  - Sauvegarder le contexte (fait automatique en langage C) : registres, …
  - Définir la source d'interruption (si le sous programme est commun entres plusieurs sources d'interruption)
  - Réinitialiser les flags d'interruption
  - Ecrire le code relatif à l'application
  - Restituer le contexte (fait automatique en langage C)

Accès à la procédure d'IT

- Les adresses des procédures de service d'IT sont disponibles dans une table résidant en mémoire principale, *à un endroit fixée par le constructeur.*

⇒ Cette table est généralement appelée table des vecteurs d'IT.

- L'élément n°i ($1 \leq i \leq N$) de la table contient l'adresse de la procédure associée à la borne IRQ_i du µP : dans la zone mémoire ISR_IRQi

- Pour sauter à la procédure d'IT associée à la requête émise sur la borne IRQ_i, le µP fait :

PC ← tab_vec[i]

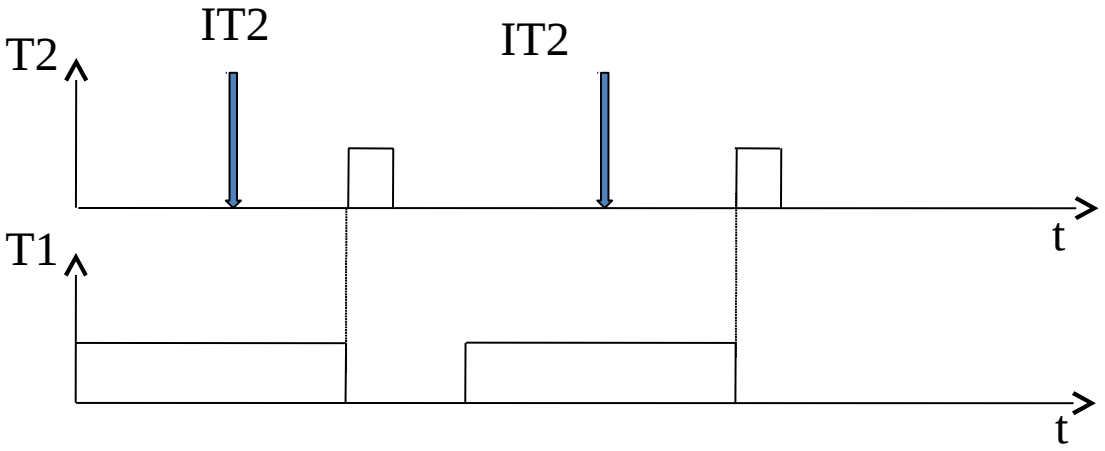
Rôle de la pile dans le traitement d'une interruption

- Elle va servir à sauvegarder le contexte avant le départ en routine :
  - adresse du PC pour le retour
  - valeur des registres au moment de l'interruption

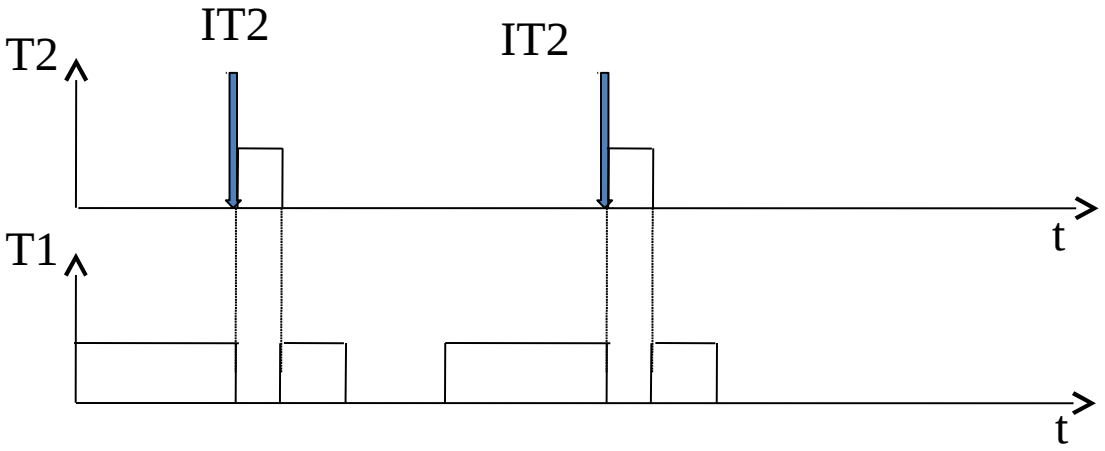
- Elle va servir à restituer le contexte à la fin de la routine :
  - valeur des registres au moment de l'interruption
  - adresse du PC pour le retour

- Remarque :
  - sauvegarde et restitution faite implicitement en C mais à faire en assembleur

Ordonnancement de deux tâches par interruption



Priorité T1 > priorité T2
donc T1 n'est pas interrompu



Priorité T1 < priorité T2
donc T1 interrompu

Chapitre 4
Programmation multitâches

Bibliographie

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

https://computing.llnl.gov/tutorials/pthreads/index.html

Norme POSIX

«**P**ortable **O**perating **S**ystem **I**nterface (as in uni**X**)»

BUT :

garantir la **portabilité des applications** au niveau du **code source** entre des Systèmes d'Exploitation (« OS ») respectant cette norme (« POSIX compliant »).

MOYEN :

définir les services offerts par le Système d'Exploitation et les interfaces avec les langages de programmation (« API : Application Program Interface »).

Execution concurrente avec POSIX

- Deux mécanismes : fork et **pthreads**
- fork : création d'un nouveau ***processus***
- **pthreads** : extension à POSIX pour permettre de créer des ***threads*** (dans un même processus)
  - **Utilisation de thread dans ce cours**
  - ***Threads*** sont crées en utilisant un objet avec des attributs appropriés
- Librairie :
  - `#include <pthread.h>`
  - Compilation : en utilisant –lpthread.

Thread : fil d’exécution d’un programme

- Pour quoi faire : avoir plusieurs fils d’exécution en même temps

- Threads d’un processus partagent :
  - En mémoire : text (code), data (variables globales)
  - Descripteur : PID, gestion signaux et fichiers ouverts

- Threads d’un processus ne partagent pas :
  - En mémoire : pile (variables locales)
  - Descripteur : copie registres

Création de threads POSIX : pthread_create()

- Synopsis :

```
#include <pthread.h>
int pthread_create(pthread_t * thread,
            const pthread_attr_t * attr,
            void * (*start_routine)(void *)
            void * arg);
```

- Description :
  - Permet de créer un nouveau thread
  - Thread exécute procédure `start_routine` en lui passant `arg` comme argument
  - `attr` est généralement `NULL`
  - En cas de succès dans la création, identifiant thread placé dans `thread`

Sortie de threads POSIX : pthread_exit()

- Synopsis :

```
#include <pthread.h>
int pthread_exit(void *value_ptr);
```

- Description :
  - Permet de savoir quand un thread se termine
  - `value_ptr` est généralement `0`
  - Utilisation dans la procédure du thread associé

Attente de la fin thread POSIX : pthread_join()

- Synopsis :

```
#include <pthread.h>
int pthread_join(pthread_t th,
            void **thread_return);
```

- Description :
  - Suspend l'exécution du thread **appelant** jusqu'à la fin du thread `th`
  - `thread_return` est généralement `NULL`
  - Tout thread doit être attendu si on veut éviter des fuites mémoires
  - Utilisation dans le programme principal

Gestion variables

- Variables partagées : déclaration en globale et volatile

```
volatile int total = 0;
```

- Passage de paramètre à la procédure exécutée :
  - dans le processus du thread :

```
int mon_numero = *(int *) arg ;
```

  - dans la procédure principale :

```
int numero = 2;
pthread_create(&mon_thread,
  NULL,ma_procedure, (void *)&numero)
```

Exemple complet 1 :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <pthread.h>

void *fonction_thread (void *
arg)
{
   int i;
   for (i = 0 ; i < 5 ; i++) {
      printf ("%s thread:
%d\n", (char*)arg, i);
      usleep(5000000);
   }
pthread_exit(0);
}
```

```
int main (void) {
   pthread_t th1, th2;
   void *ret;

   if (pthread_create (&th1,
NULL, fonction_thread,
"Premier") < 0) {
      perror("premier
(pthread_create)");
     exit (-1);
   }
   if (pthread_create (&th2,
NULL, fonction_thread,
"Second") < 0) {
      perror("second
(pthread_create)");
      exit (-1);
   }
   pthread_join (th1, &ret);
   pthread_join (th2, &ret);
   return 0;
}
```

- Commenter ce programme
- Exécuter ce programme et expliquer les résultats

Exemple complet 2

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <pthread.h>

volatile int total = 0;

void *fonction_thread (void *
arg)
{
   int mon_numero=*(int *)arg;
   for (i = 0 ; i < 5 ; i++)
      total = total + 1 ;
   printf ("Je suis le thread:
%d\n", mon_numero);
pthread_exit(0);
}
```

```
int main (void) {
   pthread_t th;
   int numero = 2 ;

   pthread_create (&th, NULL,
fonction_thread, (void
*)&numero);
   pthread_join (th1, NULL);
   printf("total=%d\n",total);
   return 0;
}
```

- Commenter ce programme
- Exécuter ce programme et expliquer les résultats

Exercice :

Tache Somme_1

```
printf("je dors pendant 10
secondes\n");
sleep(10);
u=a+b;
printf("u=%d\n",u);
printf("je dors pendant 5
secondes\n");
sleep(5);
```

Tache Produit_1

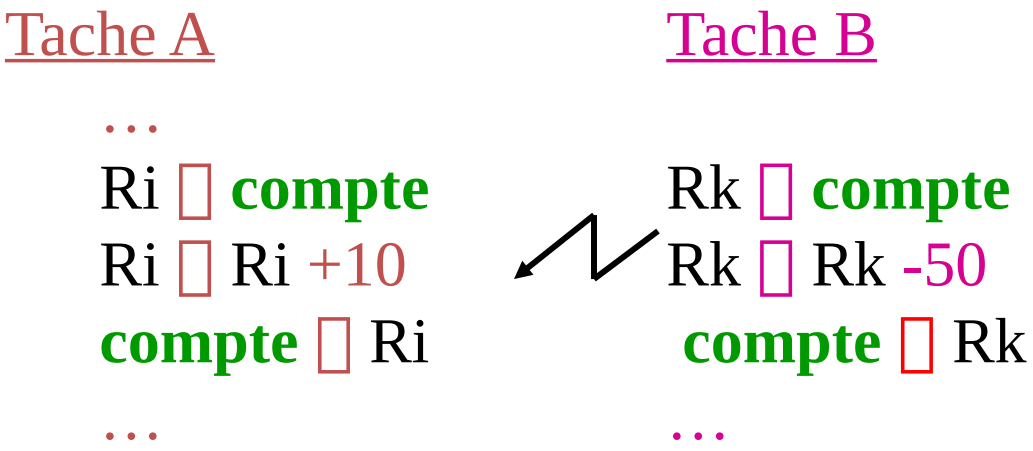
```
printf("je dors pendant 5
secondes\n");
sleep(5);
z=x*y;
printf("z=%d\n",z);
printf("je dors pendant 10
secondes\n");
sleep(10);
```

- Faire un programme monotache appelant successivement une fonction Somme_1 et Produit_1 pour a,x=3 et b,y=4
- Ecrire le même programme en multi-tache
- Comparer les temps de calculs en utilisant la commande *time*

Section critique : verrou

- Différentes taches : même ressources
- Protéger ces ressources : section critique
- Plusieurs outils, mais dans ce cours : verrou (mutex)

Protection ressource : exemple Compte



Au début, **compte** = 100.

La tache A est interrompue au moment où Ri =110 puis la tache B est exécutée. A la fin de la tache B, **compte** = 100-50 = 50

La tache A reprend son exécution et finalement, **compte** = 110 (au lieu de 60)

→ Protéger la variable **compte**

Section critique

- Définition : morceau de code manipulant une ressource critique (partagée)
- Propriétés :
  - Exclusion mutuelle : un seul thread accède à une ressource partagée

protéger la ressource quand prise par un thread

  - Famine : situation dans laquelle un thread ne parvient pas à accéder à une ressource commune

assurer au thread l'entrée de la section critique au bout d'un temps fini

Principe verrou

- Accès à une ressource partagée protégée par un verrou (mutex)

- Variable mutex de type enregistrement :
  - champ état de type booléen (OUVERT, FERME)
  - champ queue de type file_attente (FIFO).

- A chaque ressource partagée : variable verrou et les fonctions pour l'ouvrir et le fermer :
  - Verrouiller (lock)
  - Déverrouiller (unlock)

Utilisation d'un verrou

- Procédure **lock** (*mutex*)

  Si *mutex.etat* = OUVERT **Alors**

  mutex.etat ← FERME

  Sinon *mutex.queue* ← ID_tache et **mettre dans l'état Bloquée**

  Fin Si

- Procédure **unlock** (*mutex*)

  Si *mutex.queue* ≠ VIDE **Alors**

  lire la première ID_tache et **mettre la tache dans l'état Prête**

  Sinon *mutex.etat* = OUVERT

  Fin Si

- Pour qu'une tache et une seule accède à une ressource partagée R, protégée par variable *mutex_R*, on écrit les instructions suivantes:

…

lock (*mutex_R*) // réserver R

accéder à la ressource

unlock (*mutex_R*) // libérer R

ATTENTION : il faut que **tous** les processus respectent la convention

Exemple Compte

Tache A
lock (*mutex_cpte*)
Ri ← **compte**
Ri ← Ri +10
compte ← Ri
unlock (*mutex_cpte*)

Tache B
lock (*mutex_cpte*)
Rk ← **compte**
Rk ← Rk -50
compte ← Rk
unlock (*mutex_cpte*)

Au début, **compte** = 100.
Si la tache A « prend la main », la tache B ne sera exécutée que quand , **compte** = 110.
A la fin de la tache B, **compte** = 110-50 = 60
ATTENTION: « posséder » un verrou n'empèche pas les interruptions, ni les commutations

Verrou – Norme POSIX

- Déclaration (en variable globale) :

```
pthread_mutex_t mon_mutex;
```

- Initialisation (dans `main()` avant création des threads) :

```
pthread_mutex_init(&mon_mutex, NULL);
```

- Verrouiller (dans la procédure exécutée par un thread) :

```
pthread_mutex_lock(&mon_mutex);
```

- Déverrouiller (dans la procédure exécutée par un thread) :

```
pthread_mutex_unlock(&mon_mutex);
```

- Destruction (dans `main()` après fin des threads) :

```
void pthread_mutex_destroy(&mon_mutex);
```

Verrou – Norme POSIX : précautions

- Attention : verrouiller deux fois un même mutex pas toujours bloquant
- Seul le thread ayant verrouiller mutex peut déverrouiller
- Attention aux sections critiques imbriquées : interblocage

Exemple Comptage

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex1 =
PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

void *functionC1()
{
   pthread_mutex_lock( &mutex1 );
   printf("C1 dort 5 secondes\n");
   sleep(5);
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
void *functionC2()
{
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("C2 dort 10 secondes\n");
   sleep(10);
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

```
main()
{
  int rc1, rc2;
  pthread_t thread1, thread2;

  /* Create independent threads each of
which will execute functionC */
  if( (rc1=pthread_create( &thread1, NULL,
&functionC1, NULL)) )
   {
      printf("Thread creation failed: %d\n",
rc1);
   }
  if( (rc2=pthread_create( &thread2, NULL,
&functionC2, NULL)) )
   {
      printf("Thread creation failed: %d\n",
rc2);
   }

  /* Wait till threads are complete before
main continues.*/
  pthread_join( thread1, NULL);
  pthread_join( thread2, NULL);
  exit(0);
}
```

Commenter et exécuter ce programme

Exercice :

Reprendre l'exercice sur la somme et le produit avec la description des tâches ci-dessous (en prenant a=4, b=3, x=4) :

1. En reprenant le programme précédent, décrire ce qui se passe
2. Utiliser un MUTEX (pour protéger z) pour corriger le problème vu précédemment

```
TACHE SOMME
z←10
dodo(10)
u←x+z
dodo(5)
```

```
TACHE PRODUIT
dodo(5)
z←a*b
dodo(10)
```

Sémaphores: extension verrou / synchronisation

- Ressource : plusieurs accès simultanés peuvent être possible

  - Verrou binaire : insuffisant

  nécessite extension concept

- Synchronisation de plusieurs taches

Extension d’un verrou : Sémaphore

- Utilisation : ressource accessible par plusieurs taches à la fois

- Sémaphore à compte :

type semaphore sem = enregistrement

- champ *jeton* de type entier
- champ *attente* de type file de taches.

- Valeur initiale : sem.jeton = N (≥1) // nombre de threads passant
  sem.attente = VIDE

- Exemples :
  - accès à une des N cabines d’essayage : distribue ticket avec le n° de la cabine si au moins une cabine libre et sinon un n° d’ordre dans file d’attente.
  - parking de N places.

Opérations sur sémaphores

- Avant utilisation, le sémaphore sem doit être initialisé ainsi :

  sem.jeton = N /*au départ */

  sem.attente = VIDE /* et aucune tache n'est en attente*/

- Opération indivisible P (sem) /* Prendre un jeton */

  Début P

  Si sem.jeton > 0 Alors

  sem.jeton sem.jeton - 1

  Sinon mettre la tache dans la file sem.attente et dans l'état bloqué

  Fin Si

  Fin P

- Opération indivisible V(sem) /* Rendre son jeton */

  Début V

  Si sem.attente ≠ VIDE Alors

  sortir tache de tête de sem.attente et la mettre dans l'état prête

  Sinon sem.jeton sem.jeton + 1

  Fin Si

  Fin V

Sémaphore – Norme POSIX

- Librairie :

```
#include <semaphore.h>
```

- Déclaration (en variable globale) :

```
sem_t mon_sem;
```

- Initialisation (dans `main()` avant création des threads) :

```
sem_init(&mon_sem, 0, val_init);
```

- Prendre jeton (dans la procédure exécutée par un thread) :

```
sem_wait(&mon_sem);
```

- Rendre jeton (dans la procédure exécutée par un thread) :

```
sem_post(&mon_sem);
```

- Destruction (dans `main()` après fin des threads) :

```
sem_destroy(&mon_sem);
```

Exemple Sémaphore

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <pthread.h>
#include<semaphore.h>
#include<unistd.h>
#include<fcntl.h>

sem_t semaphore;

static int aleatoire( int maximum){
 double d;
 d = (double)maximum* rand();
 d = d / (RAND_MAX + 1.0);
 return ( (int)d);
}
void * process_thread( void * num_thread){
 int i;
 for( i=0; i<2; i++){
    sem_wait( &semaphore);
    fprintf(stdout,"Thread %d dans portion
critique\n",(long int)num_thread);
    sleep( aleatoire(4));
    fprintf(stdout, "Thread %d sort de la
portion critique\n",(long int)num_thread);
    sem_post( &semaphore);
    sleep( aleatoire(4));
   }
return NULL;
}
```

```
int main( void){

   long int i;
   pthread_t thread;

   sem_init( &semaphore, 0, 3);

   for( i=0; i<10; i++){
      pthread_create( &thread, NULL,
process_thread, (void *)i);
   }

   pthread_exit( NULL);
}
```

Commenter et exécuter ce programme

Autre utilisation : Synchronisation / Rendez vous

La partie de programme **depuis l'instruction** I_{n+1} de $\mathbf{T_k}$ **ne doit être exécutée qu'après exécution de l'instruction** $\mathbf{I_p}$ de $\mathbf{T_j}$:

Tache $\mathbf{T_k}$	Tache $\mathbf{T_j}$
instruction I_1	*instruction* I_1
…	…
instruction I_n	***instruction*** $\boldsymbol{I_p}$
sem_wait (sem) //P	**sem_post (sem) // V**
instruction $\boldsymbol{I_{n+1}}$	*instruction* I_{p+1}
…	…

sem.jeton = 0 /*au départ, l'événement n'a pas eu lieu */
sem.attente = VIDE /* et aucune tache n'est en attente*/

Exercice :

1. On veut maintenant réaliser le programme contenant les deux tâches suivantes. Utiliser un sémaphore pour réaliser ce programme à l'aide des deux threads somme_1 et produit_1

```
TACHE SOMME
dodo(10)
u←a+b
dodo(5)
```

```
TACHE PRODUIT
dodo(5)
z←a*u
dodo(10)
```

2. On veut maintenant réaliser le programme contenant les trois tâches suivantes. Utiliser des sémaphores pour réaliser ce programme à l'aide des trois threads somme_1, somme_2 et produit_1

```
TACHE SOMME 1
dodo(10)
u←a+b
dodo(5)
```

```
TACHE SOMME 2
dodo(10)
v←c+d
dodo(5)
```

```
TACHE PRODUIT
dodo(5)
z←v*u
dodo(10)
```

Moniteur: Variables conditionnelles

Contrer limites sémaphores :
- Utiliser mutex pour protéger variables globales nécessaires à synchronisation
- Interblocage

Moniteur propose :
- Encapsulation variables nécessaire à synchro
- primitive de blocage systématique (point d'attente)
- primitive de réveil sans conséquence si aucune tache bloquée au point d'attente

Exemple compte avec sémaphore

Comment faire pour que la mise à jour d'une variable partagée ne puisse se faire que si une condition est vérifiée ?

Tache A	Tache B
lock(*mutex_compte*)	**lock**(*mutex_compte*)
Si compte ≤ 0 Alors	compte compte +1
sem_wait (*positif*)	Si compte > 0 Alors
Finsi	**sem_post**(*positif*)
compte compte – 1	Finsi
unlock (*mutex_compte*)	**unlock** (*mutex_compte*)

où compte protégé par *mutex_compte* et *positif* sémaphore.

Problèmes :
- Blocage : on attend *positif* alors que *compte* est protégé : interblocage
- Sémaphore à point bloquant dangereux car difficile de faire le compte des jetons

Moniteur / Variable Conditionnelle

Nature : variable sans valeur

Propriétés :
- toujours associée à un mutex.

Principe :
- permet à une tache possédant un mutex **d'attendre un signal** *indiquant qu'une condition est vérifiée* **en déverrouillant le mutex de manière atomique** (indivisible) : `wait()`
- autre tache puisse :
    - avoir accès au mutex,
    - signaler que la condition attendue est vérifiée,

pour libérer la tache bloquée : `signal()`.

En pratique : POSIX

Principe :

- Encapsulation : réalisée par une Section Critique crée par un mutex (pthread_mutex_t)

- Variable Condition : pthread_cond_t
  - associée aux fonctions `wait()`et `signal()`.

- Remarques :
  - appel à wait() est toujours bloquant (à la différence de mutex_lock ou sem_wait)
  - signal() essaye de débloquer un thread, le signal est perdu si aucun thread n'est bloqué lors de son émission (alors que V incrémente un compteur).

Condition + mutex – Norme POSIX

- Librairie :

```
#include <pthread.h>
```

- Déclaration (en variable globale) :

```
pthread_cond_t ma_condition;
pthread mon_mutex;
int predicat;
```

- Initialisation (dans `main()` avant création des threads) :

```
pthread_mutex_init(&mon_mutex, NULL);
pthread_cond_init(&ma_condition, NULL);
```

- Destruction (dans `main()` après fin des threads) :

```
pthread_cond_destroy(&ma_condition);
pthread_mutex_destroy(&mon_mutex);
```

Condition + mutex – Norme POSIX : Attente

- wait() (dans la procédure exécutée par un thread) :

```
pthread_mutex_lock(&mon_mutex);
…
pthread_cond_wait(&ma_condition,&mon_mutex);
…
pthread_mutex_unlock(&mon_mutex);
```

- à l'exécution de **pthread_cond_wait()** un **pthread_mutex_unlock()** est automatiquement exécuté afin de libérer la section protégée
- lors du réveil, un **pthread_mutex_lock()** est automatiquement exécuté afin de retourner en section protégée

Condition + mutex – Norme POSIX : Réveil

- Principe :
  - Tache qui réveille envoie signal : sans précaution ou en fonction d'une condition

- signal (dans une autre procédure exécutée par un thread) :

```
pthread_mutex_lock(&mon_mutex);
…
pthread_cond_signal(&ma_condition);
…
pthread_mutex_unlock(&mon_mutex);
```

- à l'exécution de **pthread_cond_signal()** un signal est envoyé à tous les autres threads pour éventuellement débloquer certaines

Exemple compte avec moniteur

Tache A	Tache B
lock(*mutex_compte*)	**lock**(*mutex_compte*)
Si compte ≤ 0 Alors	compte compte +1
wait (*cond_positif*)	Si compte > 0 Alors
Finsi	**signal**(*cond_positif*)
compte compte – 1	Finsi
unlock (*mutex_compte*)	**unlock** (*mutex_compte*)

où compte protégé par *mutex_compte* et *cond_positif* variable conditionnelle.

Aucun risque d'interblocage !

Exemple Variables Conditionnelles : Comptage

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t count_mutex     =
PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_var   =
PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

int main()
{
   pthread_t thread1, thread2;

   pthread_create( &thread1, NULL,
&functionCount1, NULL);
   pthread_create( &thread2, NULL,
&functionCount2, NULL);

   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   printf("Final count: %d\n",count);
   exit(0);
}
```

```
// Write 1-3 and 8-10
void *functionCount1(){
 for(;;){
  // Lock mutex and wait to relase mutex
  pthread_mutex_lock( &count_mutex );
  // Wait functionCount2() operates on count
  //mutex unlocked signal in functionCount2()
  pthread_cond_wait(&condition_var,
&count_mutex);
  count++;
  printf("Counter value fC1: %d\n",count);
  pthread_mutex_unlock( &count_mutex );
  if(count >= COUNT_DONE) return(NULL);
  }
}
// Write numbers 4-7
void *functionCount2(){
 for(;;){
  pthread_mutex_lock( &count_mutex );
  if(count<COUNT_HALT1||count>COUNT_HALT2){
   // Cond of if met. Signal to free
   // waiting thread by freeing the mutex.
   // is now permitted to modify "count".
   pthread_cond_signal( &condition_var );
  }else{
   count++;
   printf("Counter value fC2: %d\n",count);
  }
  pthread_mutex_unlock( &count_mutex );
  if(count >= COUNT_DONE) return(NULL);
  }
}
```

Exemple Variables Conditionnelles : lecture / écriture

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <pthread.h>
/*Declaration du moniteur*/
int fait=0;
pthread_cond_t data_ok;
static pthread_mutex_t mon_verrou; /* verrou */
volatile int tableau[5]; /* variable partagee pour
eviter des problemes d'atomicite */

int main()
{
pthread_t thread1, thread2;
void *retour_thread;

/* initialisation du verrou et du moniteur */
pthread_mutex_init (&mon_verrou, NULL);
pthread_cond_init( &data_ok, 0);
/* creation du thread d'ecriture */
if(pthread_create (&thread1, NULL, ecrire_tableau,
NULL) < 0){
   fprintf(stderr,"pthread_create : erreur thread
ecrire");
   exit(1);}
/* creation du thread de lecture */
if(pthread_create (&thread2, NULL, lire_tableau,
NULL) < 0){
   fprintf(stderr,"pthread_create : erreur thread
lire");
   exit(1);}
pthread_join(thread1,&retour_thread);
pthread_join(thread2,&retour_thread);
return 0;
}
```

```
void *lire_tableau(void *nom_du_thread){
   int compteur;
   sleep(5);
   pthread_mutex_lock (&mon_verrou); /* on attend
de pouvoir ouvrir le verrou ,le thread d'ecriture
est lance avant la lecture */
   if( fait==0 ) {
      pthread_cond_wait( &data_ok, &mon_verrou);}
   fait=0;
   for(compteur=0;compteur!=5;compteur++){
      printf("lire_tableau, tableau[%d] est egal a
%d\n", compteur, tableau[compteur]);
   }
   //fait=0;
   pthread_mutex_unlock (&mon_verrou);
   pthread_exit(0);
}
void *ecrire_tableau(void *nom_du_thread){
   int compteur;
   pthread_mutex_lock (&mon_verrou); /* on
verrouille pendant l'ecriture */
   for(compteur=0;compteur!=5;compteur++){
      tableau[compteur] = 5*compteur+3;
      printf("ecrire_tableau, tableau[%d] est egal
a %d\n", compteur, tableau[compteur]);
   sleep(2); /* ralentissement du thread pour
simuler que le thread est plus lent que celui de
lecture */
   }
   fait=1;
   pthread_cond_signal( &data_ok);
   pthread_mutex_unlock (&mon_verrou);
   pthread_exit(0);
}
```

Exercice :

- Commenter et exécuter les programmes précédents

- Reprendre un précédent programme sur la somme et le produit avec les nouvelles descriptions des taches données ci-dessous.

```
                TACHE SOMME
u←-10
Pour (i←1:4) Faire
      dodo(10)
      u←u+a
      dodo(5)
FinPour
```

```
                TACHE PRODUIT
dodo(5)
Si (u>0) alors
   z←u*v
FinSi
dodo(10)
```

Exemple de problème lié à la programmation Multi-Tache

Plusieurs difficultés sont liés au multi-tache et au temps réel:

- Interblocage

Exemple simple d'un interblocage

Tache 1 :
- Verrouille R1
- Utilise R1

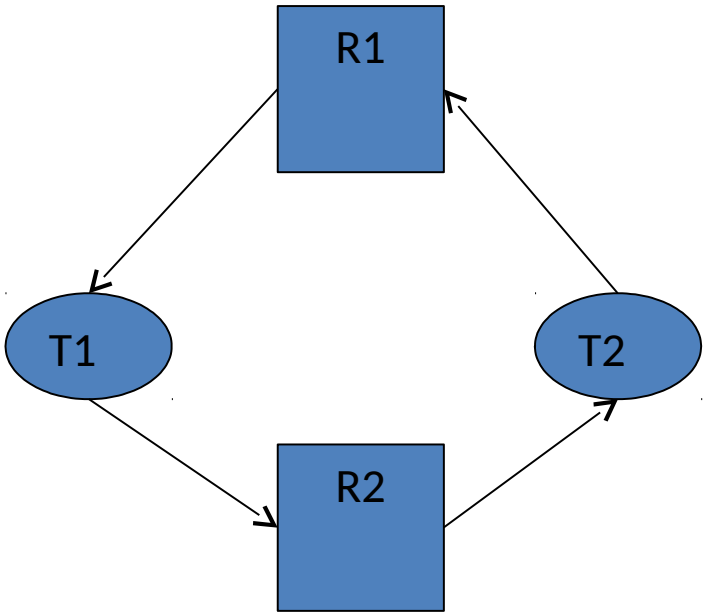
- Verrouille R2
- Utilise R2
- Déverrouille R2

- Utilise R1
- Déverrouille R1

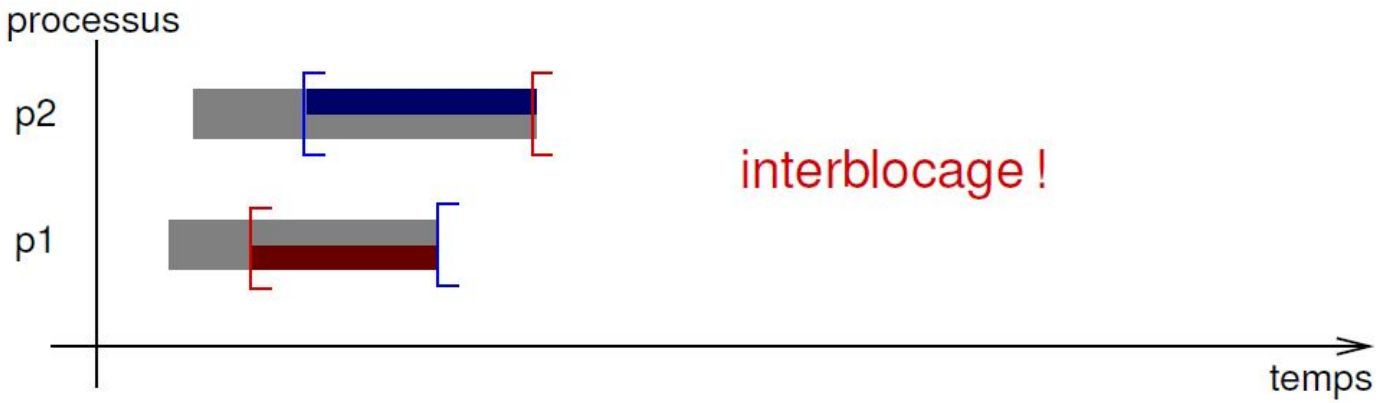
Tache 2 :
- Verrouille R2
- Utilise R2

- Verrouille R1
- Utilise R1
- Déverrouille R1

- Utilise R2
- Déverrouille R1



Si tache 1 – tache 2 : blocage



Définition interblocage

Pour avoir un interblocage, il est nécessaire d'avoir les 4 conditions suivantes :

- Exclusion mutuelle
- Détention et attente : demande de nouvelles ressources sans en relâcher d’autres
- Pas de réquisition : on ne peut prendre de force une ressource allouée à un processus
- Attente circulaire : au moins deux entités chacune en attente d’une ressource détenue par l’autre

on ne touche pas aux conditions 1 et 3. Restent les 2 et 4.

Solution 1

Moyen simple (pas toujours possible) : établir un ordre global entre les ressources et demander que chaque processus alloue les ressources selon cet ordre global

pas de cycle (éviter condition 4).

Solutions pour éviter interblocage - 1

Tache 1 :
- Verrouille R1
- Verrouille R2

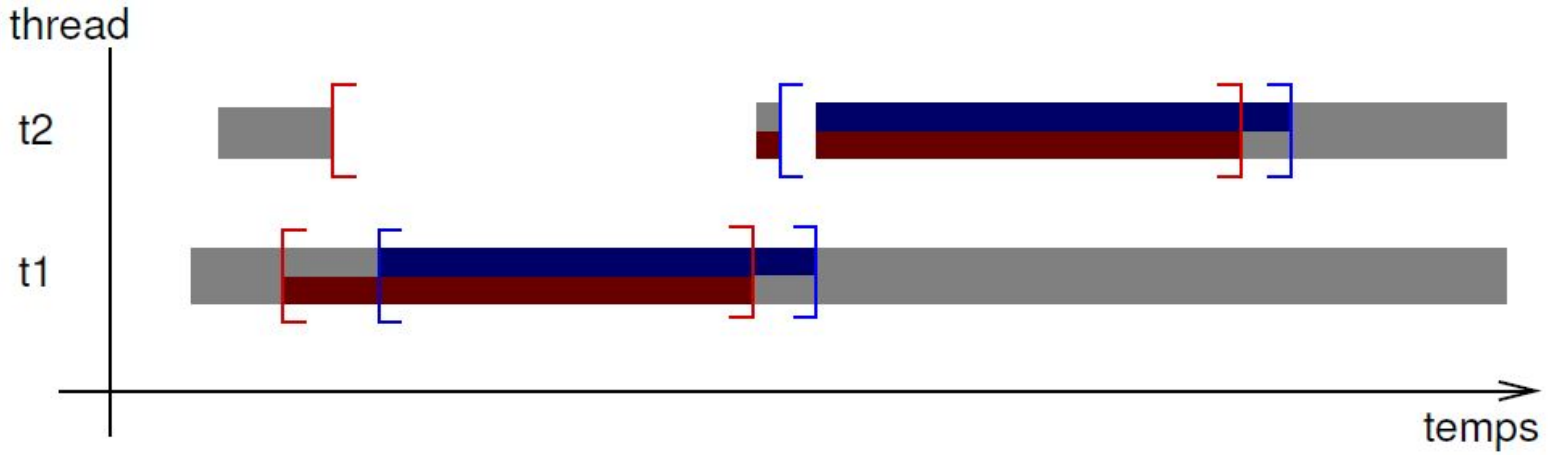
- Utilise R1
- Utilise R2
- Utilise R1

- Déverrouille R1
- Déverrouille R2

Tache 2 :
- Verrouille R1
- Verrouille R2

- Utilise R2
- Utilise R1
- Utilise R2

- Déverrouille R1
- Déverrouille R2



Définition interblocage

Pour avoir un interblocage, il est nécessaire d'avoir les 4 conditions suivantes :

- Exclusion mutuelle
- Détention et attente : demande de nouvelles ressources sans en relâcher d’autres
- Pas de réquisition : on ne peut prendre de force une ressource allouée à un processus
- Attente circulaire : au moins deux entités chacune en attente d’une ressource détenue par l’autre

on ne touche pas aux conditions 1 et 3. Restent les 2 et 4.

Solution 2

relâcher condition 2 : imaginer tous les scenarii possibles et être certain qu'aucun cycle d'attente ne peut apparaître

on développera alors des processus qui relâcheront certaines ressources avant d'en allouer d'autres afin d'éviter ces possibles cycle d'attente.

ATTENTION => pas toujours possible !

Solutions pour éviter interblocage - 2

Tache 1 :
- Verrouille R1
- Utilise R1
- Déverrouille R1

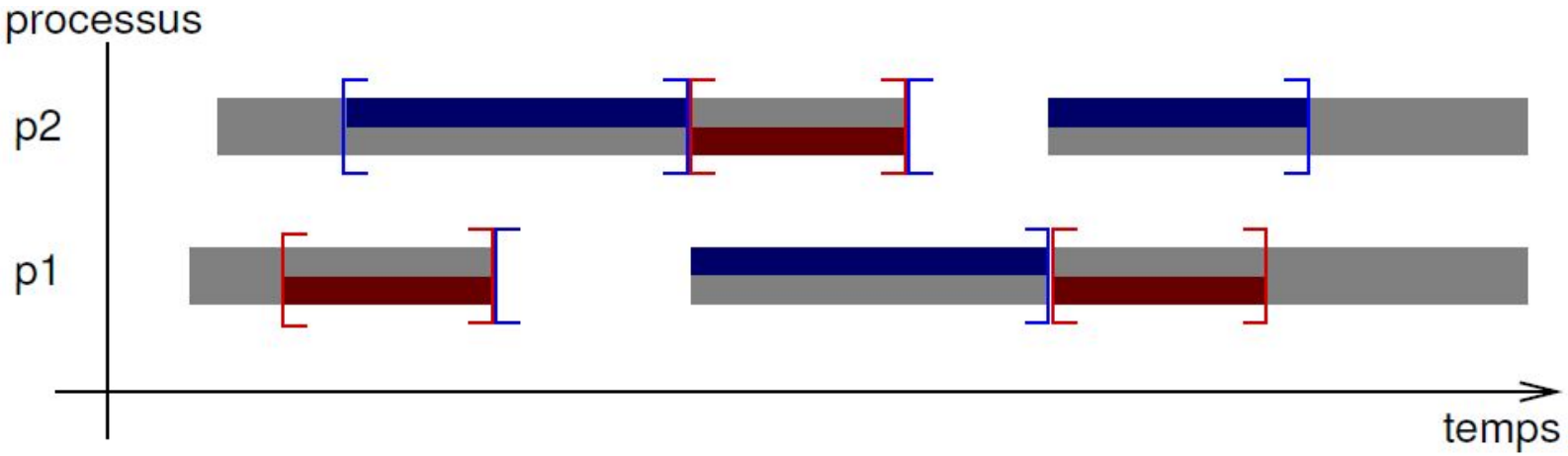
- Verrouille R2
- Utilise R2
- Déverrouille R2

- Verrouille R1
- Utilise R1
- Déverrouille R1

Tache 2 :
- Verrouille R2
- Utilise R2
- Déverrouille R2

- Verrouille R1
- Utilise R1
- Déverrouille R1

- Verrouille R2
- Utilise R2
- Déverrouille R2



Exercice

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
#include <time.h>

#include "pthread_sleep.h"

#define NB_THREADS 5

// variables partagées entre les threads
volatile int var[NB_THREADS];
pthread_mutex_t mutex[NB_THREADS];

//fonction exécutée par chaque thread
void* main_thread (void* num)
{
  int* numero = num;
  int autre = (1+*numero)%NB_THREADS;

  if ( var[*numero] == -1 && var[autre] == -1)
  {
    pthread_sleep(1.0);//latence pour mettre en
évidence le problème
    var[*numero]=*numero;
    var[autre]=*numero;
  }
  return NULL;
}
```

```
int main(void)
{
  int numero[NB_THREADS];
  pthread_t t[NB_THREADS];

  //initialisation des mutex et du tableau var
  for (int i=0; i<NB_THREADS; i++){
    pthread_mutex_init(mutex+i, NULL);
    var[i]=-1;
  }
  //itération pour créer les threads
  for (int i=0; i<NB_THREADS; i++){
    numero[i]=i;
    if (pthread_create(&t[i], NULL, main_thread,
numero+i) != 0)
    {perror("Erreur creation thread");
exit(errno);}
  }
  //itération pour attendre la fin des threads
  for (int i=0; i<NB_THREADS; i++)
pthread_join(t[i], NULL);
  //destruction des mutex
  for (int i=0; i<NB_THREADS; i++)
    if (pthread_mutex_destroy(mutex+i) != 0)
    { perror("Erreur destruction mutex");
exit(errno); }
  //itération pour attendre la fin des threads
  for (int i=0; i<NB_THREADS; i++) printf("var[%d]
= %d\n", i,var[i]);
  return 0;
}
```

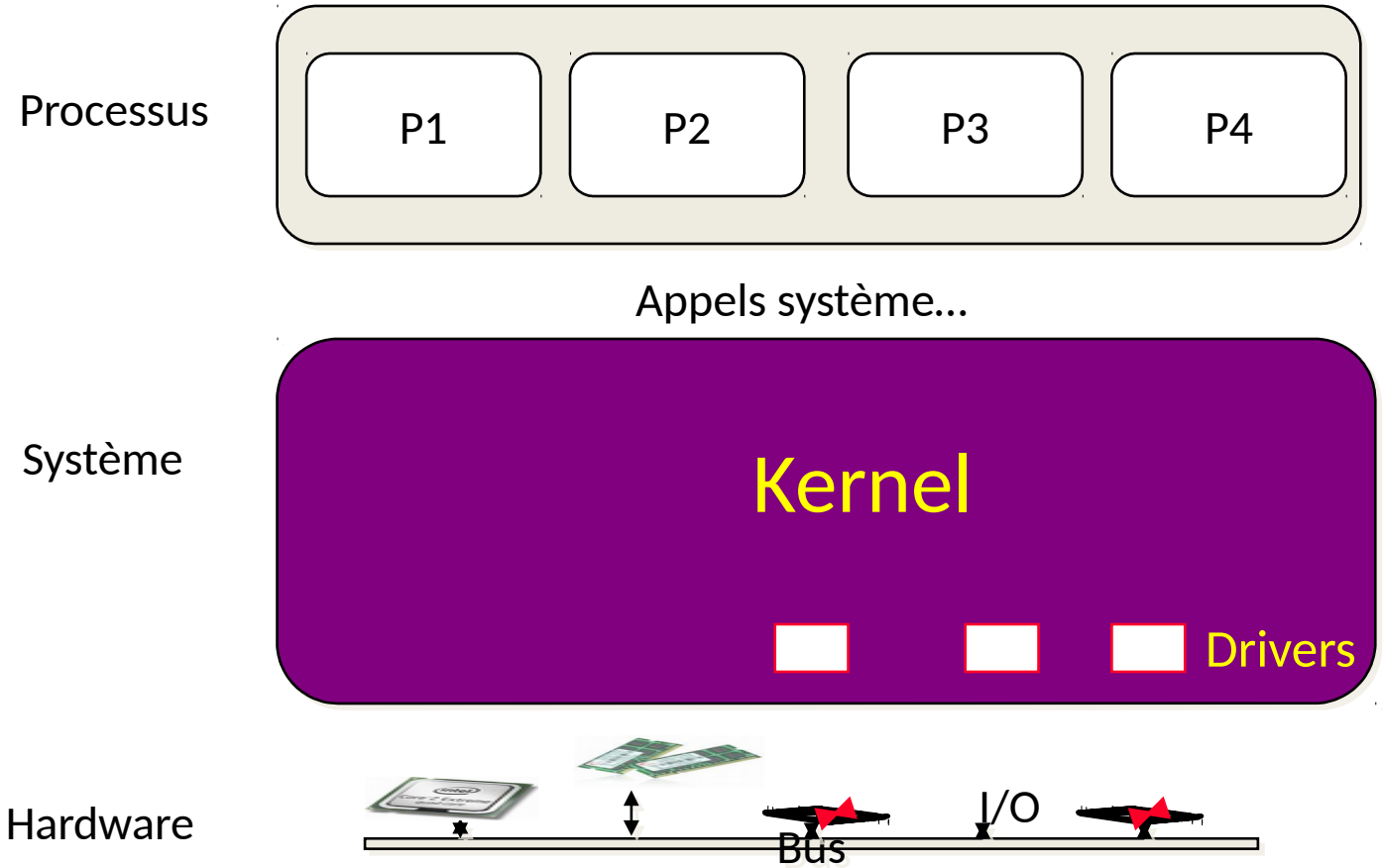
Exercice

- NB_THREAD threads et NB_THREADS ressources numérotées.
- Thread No i demande deux ressources numérotées i et (i+1)%NB_THREADS.
- Chaque ressource ne peut être attribuée qu'à un seul thread. var[i]==p si la ressource No i a été attribué au thread numéro p.

1. Exécuter le programme. Quel est le problème ?
2. Protéger les ressources. Quel est le problème ?
3. Utiliser la solution 1 pour éviter l'interblocage. Quel est le problème ?
4. Utiliser la solution 2 pour éviter l'interblocage.

Chapitre 5
Modes d’exécution

Les processus et le système…

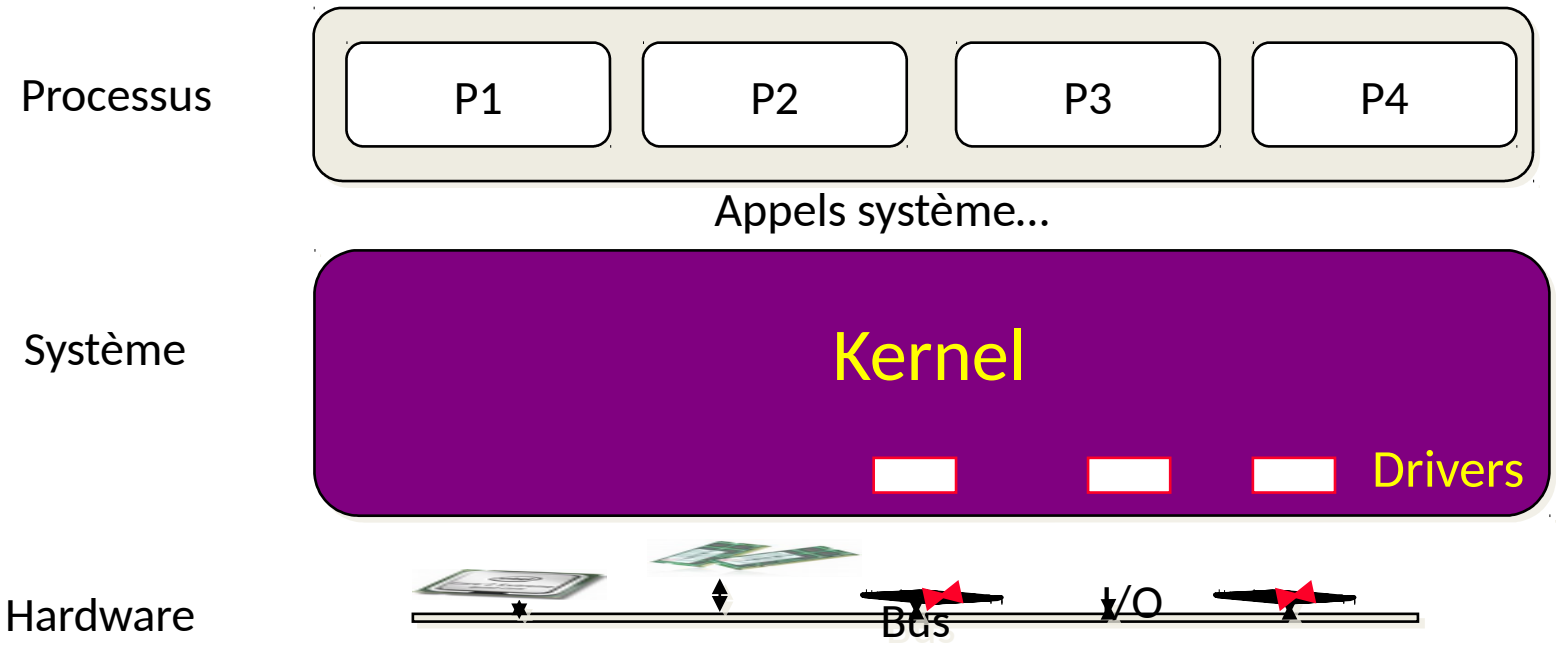


Le système

- Ensemble de routines
  - Qui s'exécutent à la suite d'événements
    - Demandes des processus
      - Accès aux périphériques d'entrée/sortie (disque, carte graphique, carte réseau,…)
    - Événements matériels
      - Eg. Interruption horloge qui permet au système de reprendre la main régulièrement
- Des structures de données systèmes
  - Représenter les processus (PID, PPID,…)
  - Représenter les fichiers (offsets…)
- Quelques « deamons »
  - Maintenance
  - Attente de connexion

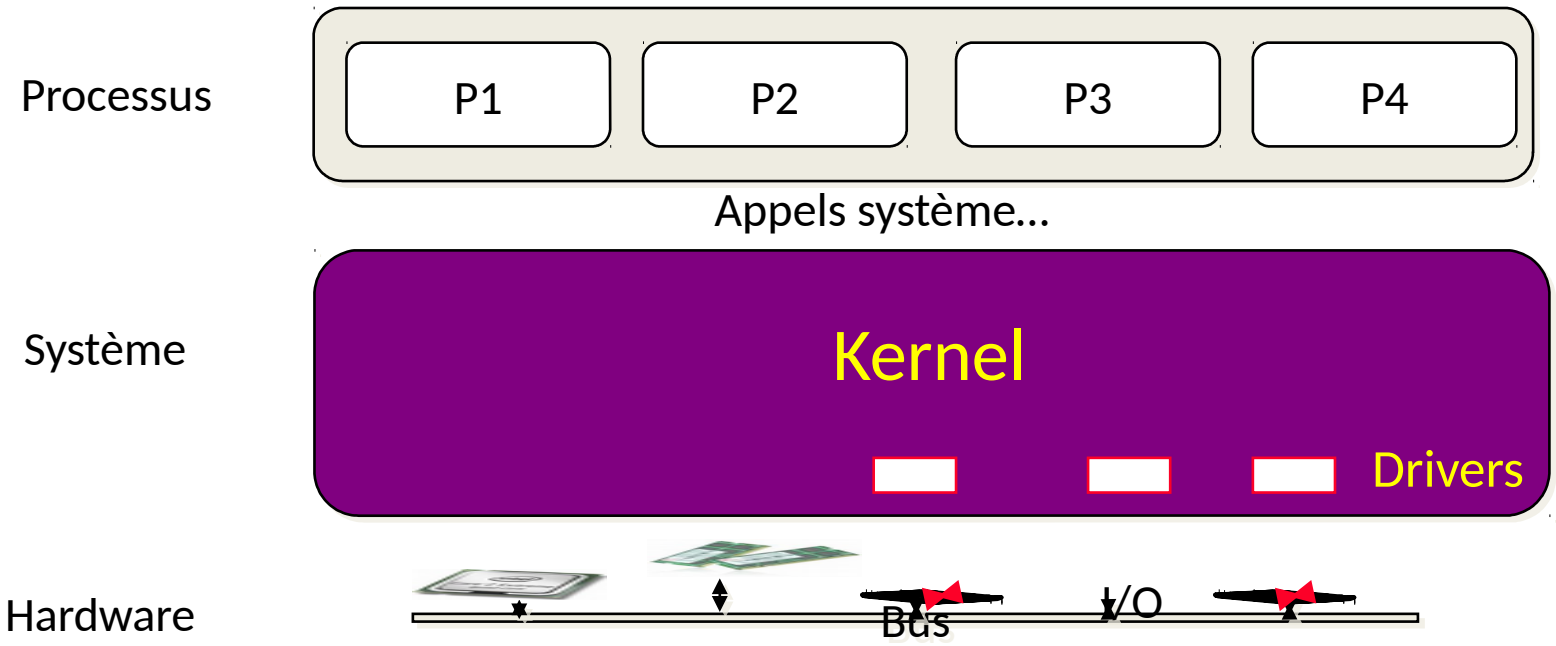
Isolation mémoire (1)

- Un processus ne peut accéder qu'à sa propre mémoire
  - Interdiction d'accéder à la mémoire d'un autre processus
  - Interdiction d'accéder à la mémoire du système
    - Structures de données système…
  - Segmentation fault…



Isolation mémoire (2)

- Comment accéder aux structures système ?
  - Eg. L'offset courant d'un fichier ouvert ?



Mode utilisateur / mode système (1)

- Un processus s'exécute en mode utilisateur
  - Accès restreint
- Passage en mode système
  - Une « trappe » permet aux processus de passer en mode système (appel système)
- Exécution mode système
  - Accès non restreint
    - Accès à l'ensemble de la mémoire / des structures systèmes…
  - Le code (les instructions) qui s'exécute(nt)
    - code système
- Fin de l'appel => retour en mode utilisateur

Mode utilisateur / mode système (2)

- En dehors du mode système un processus ne manipule que sa propre mémoire

- Passage en mode système pour
  - Accéder au reste de la mémoire (plus de limitations)
  - Accéder aux périphériques
  - …

- ATTENTION: en mode système, seul du code système peut s'exécuter
  - Installé par un administrateur (root)
  - Attention aux drivers…

Chapitre 6
Boot -- TPs

Démarrage

- Démarrage en mode S
- Exécution code indépendant de l'OS
- BIOS (Basic Input Output System)
  - Ensemble de routines pour fonctionner au démarrage
  - Charge un noyau (image vmlinuz ou bzImage) ou un boot loader à partir du système de fichiers
  - Initialisation des structures / création des premiers processus

Armadeus

- Bootloader : U-Boot (joue le rôle du BIOS des PC
- Noyau : kernel Linux
- Système de fichiers : rootfs

U-Boot

- Accès aux ressources de la carte
- Réglage de paramètres (IPs)
- Charger
  - Noyau
  - Rootfs
- Booter un noyau