# Introduction aux Systèmes d'Exploitation

email : kadionik@enseirb-matmeca.fr
web : http://kadionik.vvv.enseirb-matmeca.fr/

**Patrice KADIONIK**
**ENSEIRB-MATMECA**



## OBJECTIFS DU COURS

# Objectifs

- A l'issue des cours d'informatique, nous savons :
  - Programmer en langage C.
  - Utiliser un système d'exploitation *NIX.
  - Utiliser l'API (*Application Programming Interface*) de programmation réseaux *sockets*.
  - Comment marche peu ou prou un processeur, un ordinateur…
- Nous savons jusqu'à présent développer un programme autonome en langage C.



# Objectifs

- Nous savons jusqu'à présent développer un programme en langage C qui communique avec un autre programme distant par le réseau en utilisant l'API *sockets*.
- On peut envisager d'utiliser l'API *sockets* pour faire communiquer 2 programmes sur une même machine. Mais ce n'est pas le moyen le plus efficace.
- Il se pose donc le problème de savoir comment faire communiquer entre eux 2 programmes d'une même machine.

# Objectifs

- Il nous faut donc aborder les mécanismes de communication entre programmes : *Inter Processus Communications* (IPC).
- Cela oblige à avoir des bases sur ce qu'est un système d'exploitation multitâche (ou multiprocessus).
- Les objectifs de ce cours est de donner :
  - Les bases de conception et de fonctionnement d'un système d'exploitation multitâche. On abordera ce point sous l'angle de l'électronicien.
  - Les bases sur des moyens simples de communications IPC.



# Objectifs

- Contenu du module :
  - Cours : 3 séances de 1h20 (seulement).
  - TP : 2 séances de 3h : le but est de voir des moyens simples de communications interprocessus : signaux, tubes, tubes nommés, IPC.

# INTRODUCTION



# Une définition



**Qu'est-ce qu'un système d'exploitation ?**




# Système d'Exploitation : une définition

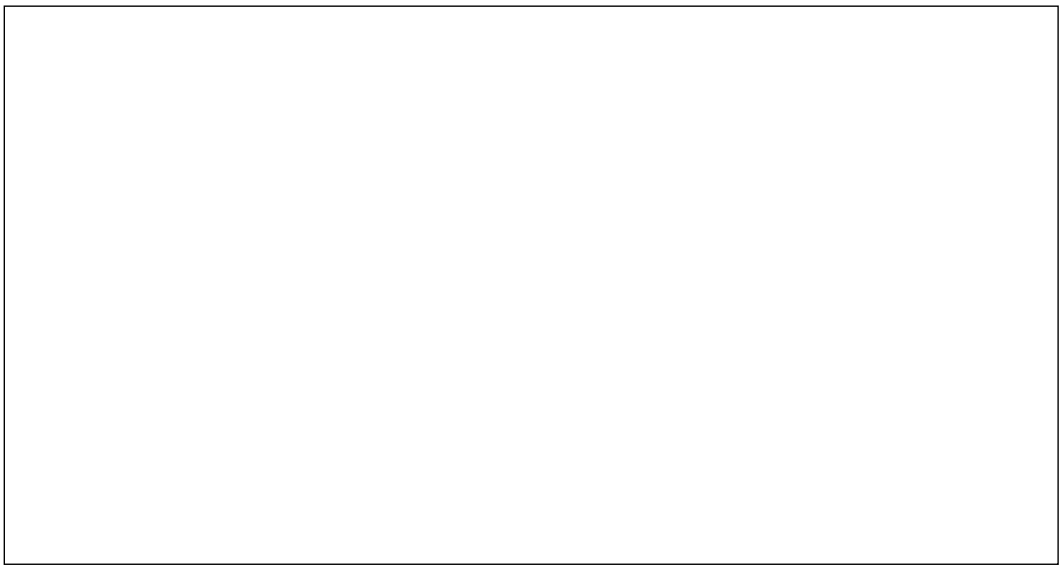
- Un **Système d'Exploitation** SE (*Operating System* ou *OS*) est un ensemble de programmes responsables de la liaison entre les ressources matérielles d'un ordinateur et les applications informatiques de l'utilisateur (traitement de texte, jeu vidéo… ).
- Il fournit aussi aux programmes d'application des points d'entrée standards aux périphériques matériels.



# Système d'Exploitation : une définition

- Le SE est donc un ensemble de programmes qui interfacent le matériel d'une plateforme (ordinateur PC ou système embarqué) et l'utilisateur ou les utilisateurs.
- Il permet de construire au-dessus du matériel une machine « virtuelle » conviviale qui cache les spécificités matérielles de la plateforme.
- Le SE fournit ainsi une API pour développer les programmes (appels système).

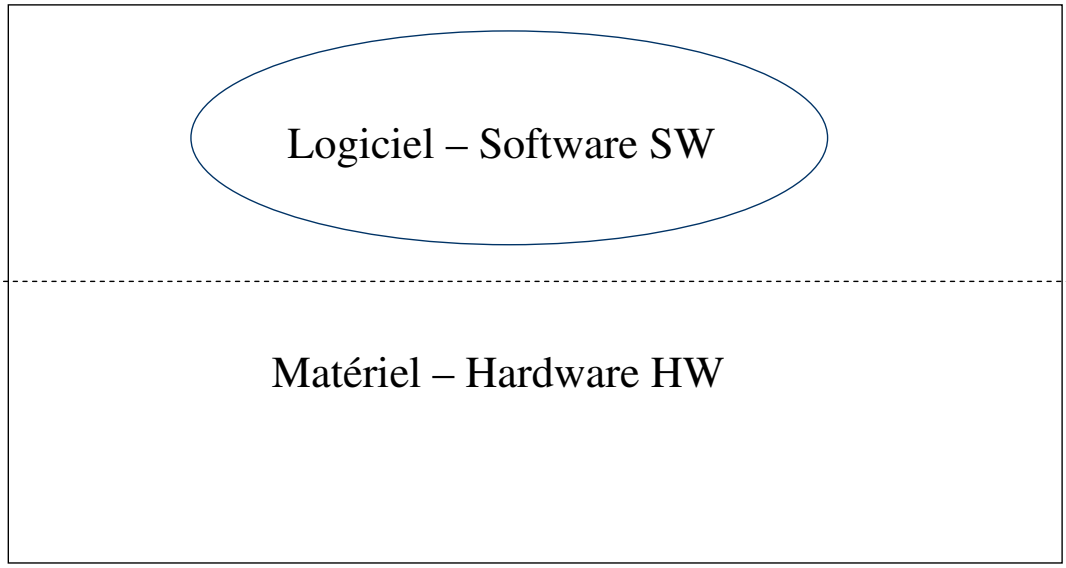
# Les briques de base



Plateforme matérielle



# Les briques de base




Plateforme matérielle

# Les briques de base

- Le matériel :
  - Processeur. On utilise plutôt un microcontrôleur ou un processeur DSP dans l'électronique embarqué.
  - Mémoire : RAM, ROM, mémoire FLASH.
  - Composants FPGA : glue logique, coprocesseurs spécialisés.
  - Stockage de masse : disque dur. On utilise pas ou peu de disque dur dans l'embarqué.
  - E/S :
    - Liaison série : RS232, RS485, bus de terrain (CAN, Profibus…)…
    - E/S générales : leds, interrupteurs, écran, clavier…
    - E/S spécifiques : commandes de moteur, acquisition de données…
    - Connectivité réseaux : Ethernet, Wifi…
  - La richesse de la plateforme est liée à la richesse de ses E/S. Dans l'électronique embarquée (ou l'embarqué), tout l'enjeu est lié à la richesse de ces E/S !



# Les briques de base

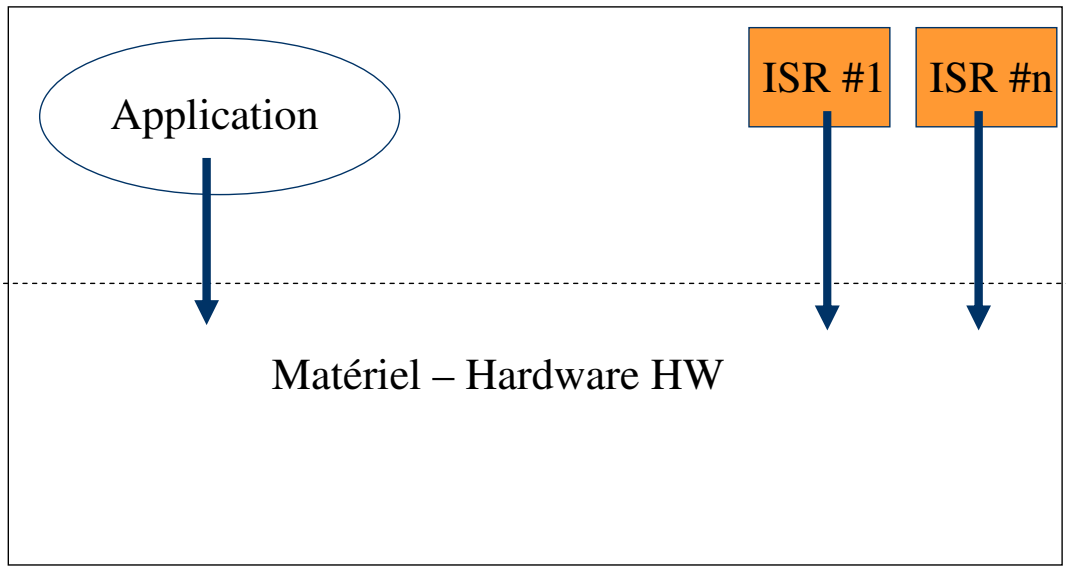
- Le logiciel :
  - C'est l'application ou les applications de l'utilisateur !
  - L'application est développée suivant les besoins avec divers langages :
    - Langage assembleur.
    - Langage C.
    - Langages orientés objet : C++, Java.
    - …

# La super boucle

- Il est possible de se passer de système d'exploitation.
- L'application accède directement au matériel et aux E/S.
- On retrouve typiquement une boucle infinie en concurrence avec les interruptions du système.
- On parle de super boucle !



# La super boucle




Plateforme matérielle

Priorités : ISR #1 > ISR #n > application

# La super boucle

- Une routine d'interruption ou ISR (*Interrupt Sub Routine*) est une « fonction logicielle » exécutée en cas d'occurrence d'une interruption matérielle ou logicielle (ex : division par 0, TRAP) ou d'une exception matérielle (ex : interruption).
- Une routine d'interruption est plus prioritaire que l'application logicielle.
- Il y a des priorités entre les routines d'interruption généralement de l'exception matérielle à l'interruption logicielle :
  - Exception la plus prioritaire : le reset.



# La super boucle

- Pseudo code en langage C :

```
main() {
while (1) {
     read input#1;
     do action#1;
     write output#1;

     read input#n;
     do action#n;
     write output#n;
     }
}

interrupt isr1() {
    acknowledge it#1;
    do something#1;
}

interrupt isrn() {
    acknowledge it#n;
    do something#n;
}
```

# La super boucle

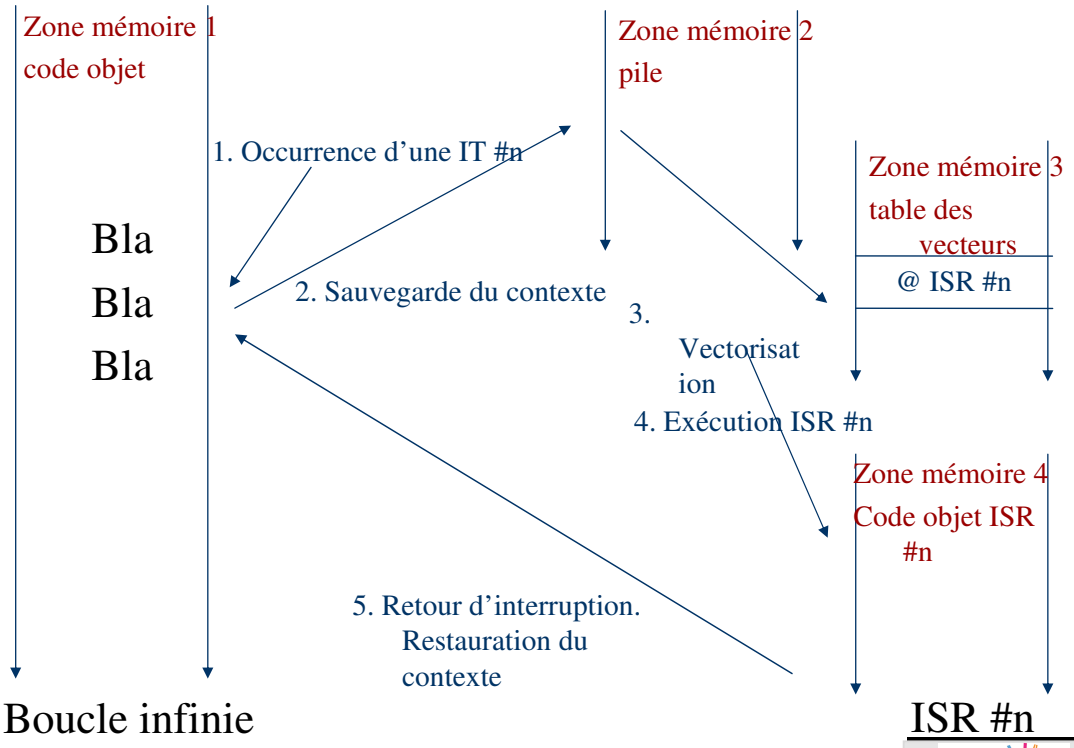
- Dans la boucle infinie, l'ordre des actions est important :
  - Suite de :
    - 1. Lecture entrées.
    - 2. Action.
    - 3. Positionnement sorties.
- On place séquentiellement en premier la suite d'instructions correspondant au contrôle d'E/S le plus prioritaire (ici #1).
- Le temps de réaction du système est la durée d'exécution d'une itération de la boucle.



# La super boucle

- Sur l'exemple de code en langage C donné, le mot clé *interrupt* n'est pas un mot réservé du langage C.
- C'est généralement une facilité rajoutée par le compilateur C utilisé pour le processeur embarqué dans la plateforme électronique pour faciliter l'installation de la routine d'interruption dans la table des vecteurs d'interruption.
- La table des vecteurs est une zone mémoire dont chaque vecteur (32 bits pour un processeur 32 bits) précise le point d'entrée (adresse) de la routine d'interruption ISR à exécuter en cas d'occurrence d'une exception du processeur.

# La super boucle






# La super boucle

- En cas d'occurrence d'une interruption, il existe un temps de latence matérielle avant sa prise en compte.
- Sauvegarde du contexte : automatiquement, le processeur sauvegarde sur la pile un ensemble de registres du processeur qui correspond à une photo instantanée de l'état du processeur au moment de l'occurrence de l'interruption.
- La sauvegarde du contexte est cette photo. Dans le cas d'un processeur Motorola 68000, on sauvegarde le registre d'état (SR) et le compteur programme (PC) pour le retour d'interruption.

# La super boucle

- Vectorisation : le processeur calcule en fonction de l'origine de l'interruption (vectorisée) le vecteur (point d'entrée de l'ISR) dans la table des vecteurs. L'adresse du point d'entrée de l'ISR sert alors à réinitialiser le compteur programme PC du processeur avec cette valeur.

- Exécution de la routine d'interruption ISR. A la fin de l'exécution, il faut restaurer l'état courant du processeur au moment de l'occurrence de l'interruption. Cela n'est pas fait automatiquement et il faudra utiliser une instruction assembleur pour cela (instruction RTE pour un processeur 68000).



# La super boucle

- Restauration du contexte : elle est réalisée dans l'ISR par l'exécution de l'instruction assembleur de restauration de contexte. Cette instruction va chercher sur la pile la sauvegarde qui avait été faite automatiquement précédemment et réinitialise les registres avec les valeurs courantes au moment de l'occurrence de l'interruption.

- On voit ainsi que l'usage du mot clé *interrupt* dans l'exemple va créer un enrobage de la routine que l'on veut installer comme ISR et rajouter automatiquement la restauration de contexte en fin d'exécution. La table des vecteurs sera aussi renseignée. C'est bien une facilité offerte par le compilateur C…

# La super boucle : bilan

- On retiendra sur l'exemple de la super boucle les mécanismes :
  - De sauvegarde de contexte entre la boucle infinie et l'ISR.
  - De restauration de contexte entre l'ISR et la boucle infinie.
- On doit pouvoir généraliser ce mécanisme de sauvegarde/restauration de contexte…
  - Pourquoi ne pas l'appliquer entre plusieurs applications concurrentes pour partager l'accès au processeur ?



# La super boucle : bilan

- L'approche super boucle pose des problèmes.
- La réactivité du système devient mauvaise quand on augmente le nombre d'instructions dans la boucle infinie.
- Le code source devient non maintenable rapidement. Il vaudrait mieux pouvoir segmenter le travail sous forme de tâches distinctes. Ce découpage facilite la maintenance du code :
  - Une tâche à réaliser ne deviendrait-elle pas une tâche au sens informatique ?
  - Ne serait-ce pas la justification d'avoir un système d'exploitation multitâche ?

# La super boucle : bilan

- La super boucle n'est pas périmée mais reste réservée aux petits systèmes.
- Le choix de l'usage d'un SE ou non doit être bien sûr le bon choix pour minimiser les coûts du système. On n'oublie pas les bonnes recettes du passé !



# La super boucle : bilan

- L'échange entre la boucle infinie et les routines d'interruption se fera par des variables partagées.
- Il faut gérer l'accès exclusif à la variable partagée car la boucle infinie ne sait pas que la valeur courante d'une variable partagée a été modifiée par une ISR en cas d'occurrence d'une interruption.
  - Il faut utiliser le mot clé *volatile* en langage C pour toute variable partagée entre la boucle infinie et une ISR !

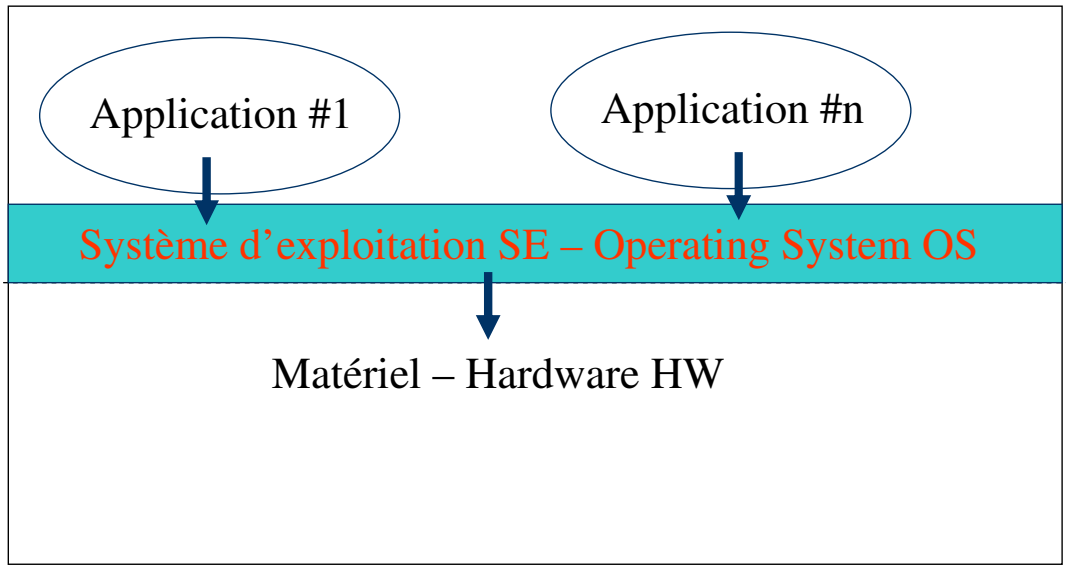
# APPORTS GENERAUX D'UN SE



## Le Système d'Exploitation

- L'usage d'un système d'exploitation va nous simplifier la vie sur différents points…
- Le SE sera donc une facilité logicielle placée entre la ou les applications et le matériel…

# Le Système d'Exploitation



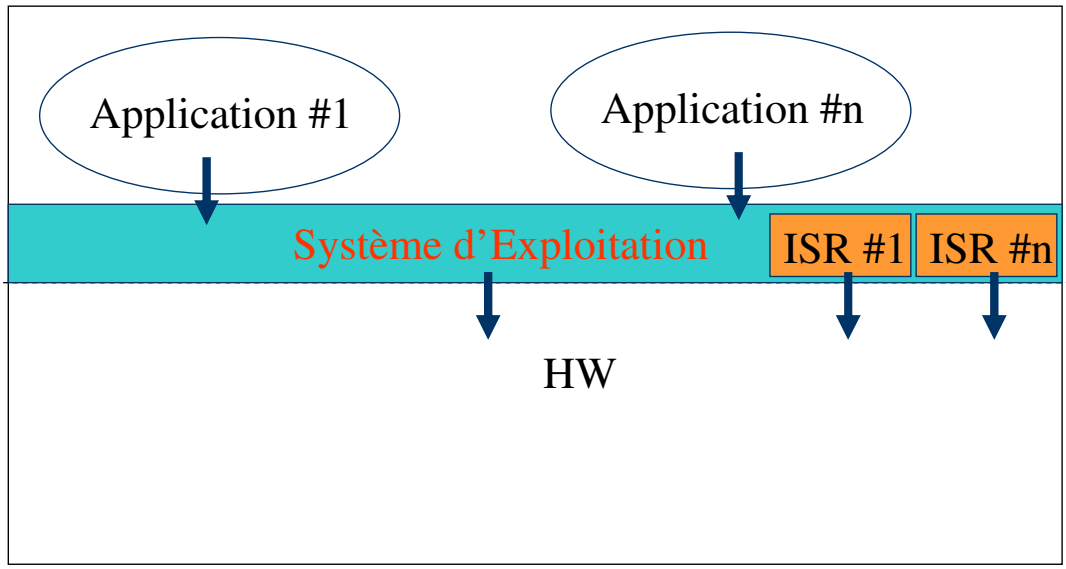

Plateforme matérielle



# Le Système d'Exploitation

- Le SE sépare les applications logicielles du matériel.
- Il faut passer par le SE pour accéder au matériel. Cela demandera en cas d'intégration d'un matériel spécifique dans la plateforme de développer un pilote de périphérique (*driver*) pour pouvoir y accéder.
- De même, on ne pourra plus accéder aux interruptions depuis la ou les applications. Les ISR sont « sanctuarisées » dans le SE.
- Les 2 principes précédents admettent des exceptions notamment quand on utilise un Système d'Exploitation Temps Réel ou quand le processeur n'a pas d'unité de gestion mémoire…

# Le Système d'Exploitation




Plateforme matérielle



# Les apports d'un Système d'Exploitation

- Le SE assure ainsi le partage des ressources physiques (matérielles) de la plateforme matérielle.
- Cette gestion doit assurer l'équité de l'accès à ces ressources par les applications et assurer aussi que les accès s'effectuent correctement. Il ne faut pas qu'une application ne corrompt les données d'une autre application par exemple.
- Le partage des ressources concerne essentiellement :
  - Le processeur.
  - La mémoire.
  - Les E/S.

# Les apports d'un Système d'Exploitation

- Une application n'accèdera aux ressources physiques que par l'intermédiaire d'une API de haut niveau composée d'appels système.
- L'accès aux E/S se fera en utilisant les pilotes de périphérique (*drivers*) qui sont aussi une API homogène d'accès au matériel.



# QUELQUES DEFINITIONS

# Quelques définitions

- Un système embarqué peut être défini comme un système électronique et informatique autonome ne possédant pas des entrées/sorties standards comme un clavier ou un écran d'ordinateur (PC).

- Le système matériel et l'application sont intimement liés et noyés dans le matériel et ne sont pas aussi facilement discernables comme dans un environnement de travail classique de type PC.

- *Un système embarqué est donc un système électronique et informatique autonome, qui est dédié à une tâche bien précise.*



# Quelques définitions

- Généralement, un système embarqué doit respecter :
  - Des contraintes temporelles fortes *(Hard Real Time)*.
  - On y trouve enfoui un système d'exploitation ou un noyau Temps Réel (*Real Time Operating System,* RTOS).

- Le Temps Réel est un concept un peu vague. On pourrait le définir comme : "*Un système est dit Temps Réel lorsque l'information après acquisition et traitement reste encore pertinente*".

# Quelques définitions

- Un système d'exploitation ou un noyau Temps Réel (*Real Time Operating System,* RTOS) est un SE spécialisé pour garantir dans tous les cas de figure les contraintes temporelles.
- Un RTOS garantit un temps maximum d'exécution pour une opération précise.



# Quelques définitions

- Un programme est une application logicielle exécutable, inerte, généralement stockée dans un fichier s'il existe un système de fichiers.
- L'exécution d'un programme revient à créer une instance en cours d'exécution de ce programme :
  - On parlera de processus pour cette instance dans le cas d'un SE généraliste (à temps partagé).
  - On parlera de tâche pour cette instance dans le cas d'un SE Temps Réel.
- On peut avoir plusieurs instances du même programme en cours d'exécution. Chaque processus a son propre contexte d'exécution…

# CLASSIFICATION DES SE



## Classification des SE

- On peut classifier les SE par la configuration matérielle :
  - Un seul processeur : système monoprocesseur.
  - Plusieurs processeurs : système multiprocesseur.
  - Basée sur un réseau : système réseau (système distribué).
- On peut classifier les SE par la méthode d'accès au système par l'usager :
  - Par session : ce sont les systèmes transactionnels ou conversationnels (réservation d'un billet SNCF).
  - Par travaux (*batch*) : traitement par lots.
- On peut classifier les SE par le nombre d'usagers simultanés.

# Traitement par lots

- Le traitement par lots (*batch processing*) est apparu dans les années 1950 et rentabilise l'utilisation des ordinateurs.
- Mise en œuvre du concept de moniteur : un programme veille sur les tâches (*jobs*) des utilisateurs.



# Traitement par lots

- Chaque utilisateur soumet ses travaux à l'opérateur de l'ordinateur :
  - Tâches à réaliser codées sous forme de cartes perforées…
  - Regroupement des tâches.
  - Exécution séquentielle des regroupements.
- La fin d'une tâche est suivie par un branchement vers le moniteur.
- Ce dernier charge en mémoire la tâche suivante et l'exécute dans le processeur.

# Traitement par lots

- L'ordinateur est soit en exécution d'un programme utilisateur soit en exécution du programme moniteur.
- Ce double niveau d'exécution sera repris dans la notion de niveaux d'exécution dans les SE modernes.
- Une tâche à exécuter peut monopoliser longtemps le processeur (attente de la fin d'une opération d'E/S). Il faudrait pouvoir récupérer ce temps perdu à attendre pour exécuter une autre tâche (partager le temps).



# Traitement en temps partagé

- Le traitement en temps partagé ou technique CTSS (*Compatible Time Sharing System*) est apparu dans les années 1960.
- Chaque utilisateur du système est relié à l'ordinateur par le biais d'un terminal. 1 utilisateur = 1 terminal = 1 tâche.
- Le processeur est contrôlé par chaque terminal durant une brève période de temps ou Intervalle de Temps IT (typiquement 0,2 s).

# Traitement en temps partagé

- Les tâches (terminaux) ont tour à tour accès au processeur.
- Politique de gestion (ordonnancement) en tourniquet (*round robin*).
- Lorsqu'une tâche est bloquée sur une opération d'E/S :
  - Elle est immédiatement bloquée.
  - Le contrôle du processeur est passé à une autre tâche.



# Traitement multitâche et multiutilisateur

- On définit un environnement multiprogrammé en temps partagé comme :
  - Multitâche : plusieurs programmes (tâches) en même temps.
  - Multiutilisateur : temps partagé au niveau de l'accès au processeur.
- Premier système : MULTICS (*MULTiplexed Information and Computer Service*) du MIT, Bell et General Electric dans les années 1960 (ancêtre d'UNIX).

# Traitement multitâche et multiutilisateur

- MULTICS a introduit différents concepts à la base des SE modernes :
  - Découplage entre le SE et le matériel.
  - Mémoire virtuelle : pagination de la mémoire.
  - Edition de liens dynamique.
  - Système de fichiers.
  - Concept de processus.



# Traitement en Temps Réel

- Dans le cas d'un système multitâche multiutilisateur, on peut garantir un temps d'exécution maximum quel que soit l'état du système (charge, nombre de tâches…). On garantit un traitement en « Temps Réel ».
- On a donc un système d'exploitation Temps Réel utilisé dans les systèmes embarqués « critiques ».
- Quelques exemples : VxWorks, QNX, µC/OS II, RTAI, Xenomai, FreeRTOS…

# CONCEPTION D'UN SE MULTITACHE : CONTRAINTES MATERIELLES



## Conception d'un SE : contraintes matérielles

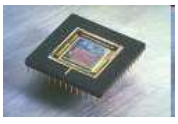
- Nous allons essayer de lister quels sont les composants matériels dont on a besoin pour mettre en place un SE multitâche moderne.
- Nous allons dresser une liste minimale…

# Conception d'un SE : contraintes matérielles

Plateforme matérielle



# Conception d'un SE : contraintes matérielles



Processeur

Plateforme matérielle

# Conception d'un SE : contraintes matérielles

- Il nous faut un processeur avec sa logique de contrôle (horloges…).
- C'est bien sûr le « cœur » du système afin de pourvoir exécuter de la logique programmée.

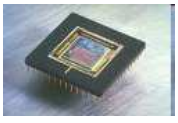


# Conception d'un SE : contraintes matérielles

RAM



ROM



Processeur

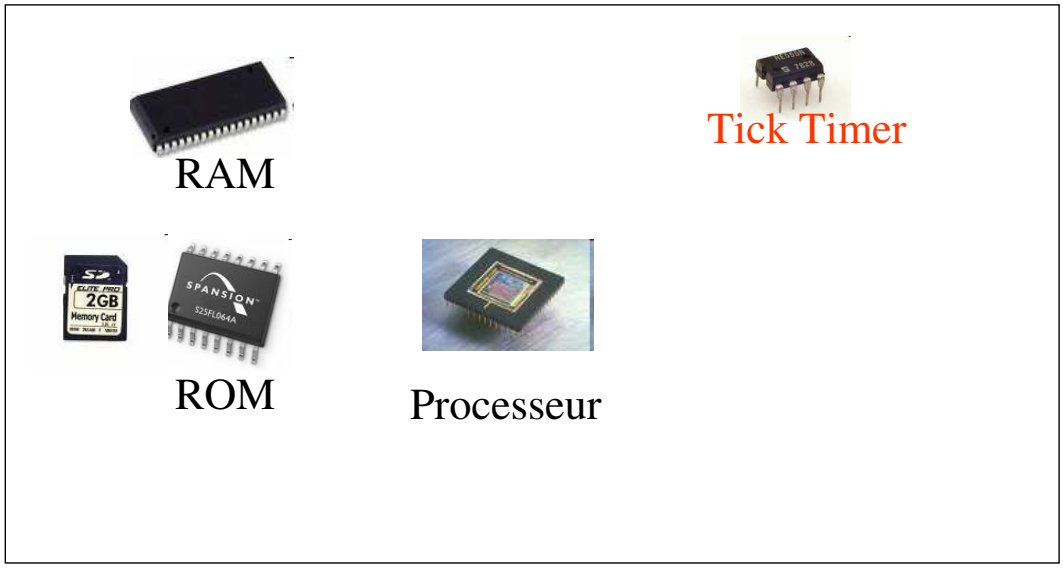
Plateforme matérielle

# Conception d'un SE : contraintes matérielles

- Il nous faut de la mémoire pour y stocker programmes et données
  - De façon temporaire : RAM, SRAM, DRAM…
  - De façon permanente : ROM, EPROM, EEPROM, FLASH…
- La mémoire RAM servira à stocker le SE, les applications et leurs données…
- La mémoire ROM est actuellement de la mémoire FLASH par facilité de reprogrammation. On y trouve dedans un programme *firmware* ou *bootloader* pour initialiser la plateforme matérielle, recopier de la FLASH en RAM le SE puis lancer le SE…



# Conception d'un SE : contraintes matérielles



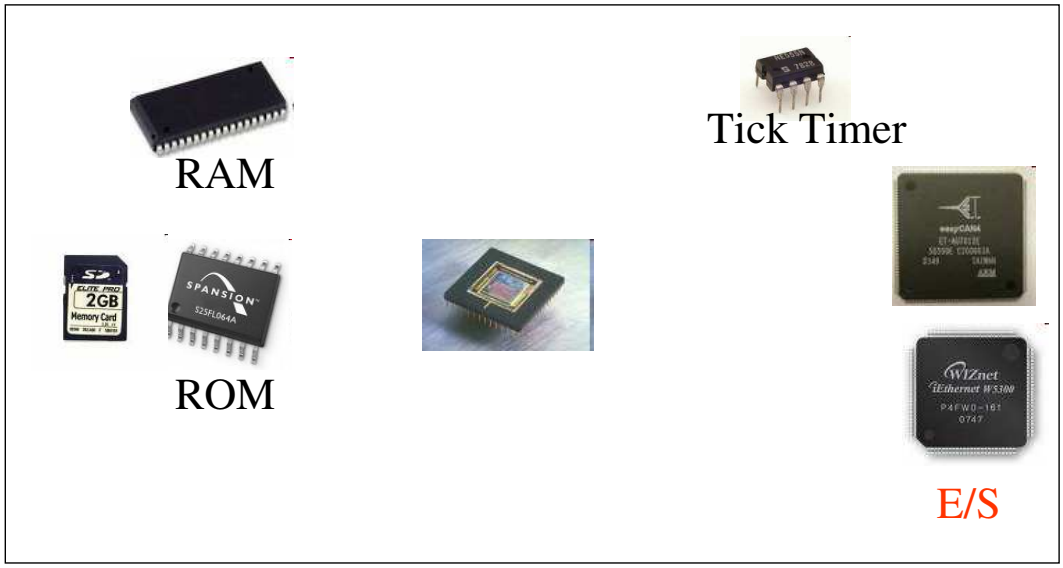

Plateforme matérielle

# Conception d'un SE : contraintes matérielles

- Le *tick timer* est une interruption périodique du processeur du système.
- C'est donc une base de temps.
- Périodiquement, on va exécuter l'ISR du *tick timer*.
- La question est de savoir quoi exécuter…



# Conception d'un SE : contraintes matérielles




Plateforme matérielle

# Conception d'un SE : contraintes matérielles

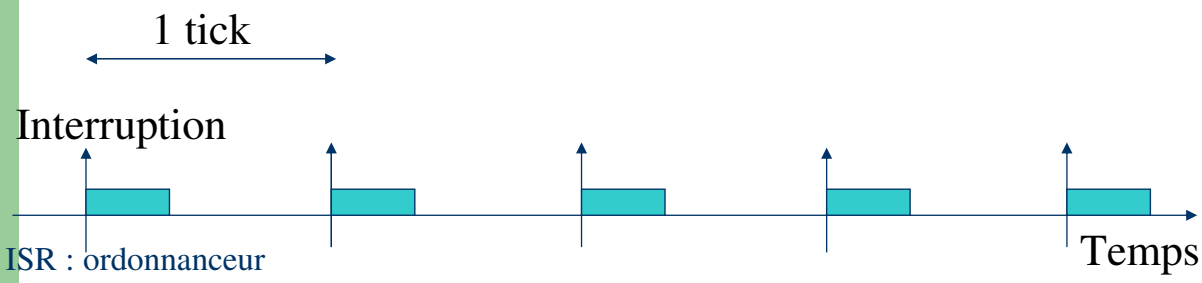
- Les E/S sont primordiales pour la plateforme matérielle car c'est par les E/S que l'on communique avec le monde extérieur !
  - Bus CAN.
  - Bus SPI, I2C, 1-Wire…
  - RS232, RS485.
  - Ethernet, ZigBee, Wifi…
  - IDE, SATA… : interfaçage de disques durs. Dans l'embarqué, on préfèrera l'usage de SSD (*Solid State Disk*).



# Conception d'un SE : le tick timer

- Nous avons vu que le *tick timer* est une source d'interruption périodique du processeur.
- Nous allons donc installer une ISR.
- Nous allons appeler cette ISR ordonnanceur ou *scheduler*.

# Conception d'un SE : le tick timer






# Conception d'un SE : le tick timer

- La durée d'exécution de l'ordonnanceur doit être négligeable par rapport à la période entre 2 interruptions du *tick timer*. En pratique, cette perte de temps est de l'ordre de 5 %.
- La durée entre 2 interruptions périodiques s'appelle le quantum de temps ou *tick*.
- Pour un SE moderne, il va de 100 ms à 1 ms (Linux 2.4 : 100 ms, Linux 2.6 et 3.x : 10 ms puis 4 ms).
- L'ordonnanceur va apporter un service : l'ordonnancement.

# Conception d'un SE : le tick timer

- L'ordonnancement est le processus de choix d'une tâche (processus) qui aura accès pour le prochain *tick* au processeur.
- C'est le principe fondateur du multitâche !
- Il existe différents algorithmes d'ordonnancement :
  - Basés sur la priorité de la tâche. La tâche de plus forte priorité prête à être exécutée aura accès au processeur.
  - Basés sur la périodicité des tâches. La tâche de plus petite période sera la plus prioritaire (*Rate Monotonic scheduling*).
  - …



# Conception d'un SE : le tick timer


Ordonnancement de 3 tâches avec 2 états : prêt (*ready)* et élu (*running)*

# Conception d'un SE : le tick timer

- On assiste donc à une commutation de tâches.
- Chaque tâche possède un environnement propre en plus du code exécuté : c'est son contexte.
- Dans le contexte, on retrouve la priorité de la tâche comme on l'a dit précédemment.
- Pour passer d'une tâche à une autre, l'ordonnanceur va changer de contextes.



# Conception d'un SE : le tick timer

- L'ordonnanceur exécute donc 3 opérations élémentaires :
  - 1. Sauvegarde du contexte de la tâche $T_p$.
  - 2. Election suivant l'algorithme d'ordonnancement choisi de la tâche $T_q$.
  - 3. Restauration du contexte de la tâche $T_q$.

# Conception d'un SE : le tick timer

- La sauvegarde du contexte de la tâche $T_p$ impose à sauvegarder au minimum dans une zone mémoire :
  - L'état courant des registres du processeur.
  - L'état courant de la tâche. La tâche $T_p$ passe de l'état élu *running* à l'état prêt *ready*.



# Conception d'un SE : le tick timer

- La restauration du contexte de la tâche $T_q$ impose à restaurer au minimum depuis une zone mémoire :
  - L'état courant de la tâche. La tâche $T_q$ passe de l'état *ready* à l'état *running*.
  - L'état courant des registres du processeur.
- Sur notre exemple simplifié, on a discerné 2 états des tâches *ready* et *running*.
- Une tâche peut avoir d'autres états d'exécution…

# Conception d'un SE : le tick timer

- La zone réservée pour sauvegarder/restaurer le contexte d'une tâche s'appelle le TCB : *Task Control Block* (ou *Processus Control Bloc*).
- Chaque TCB d'une tâche est lié aux autres en utilisant généralement une structure de liste doublement chaînée.
- La taille de cette liste est configurée au moment de la configuration du SE (liste de taille finie).



# Conception d'un SE : le tick timer

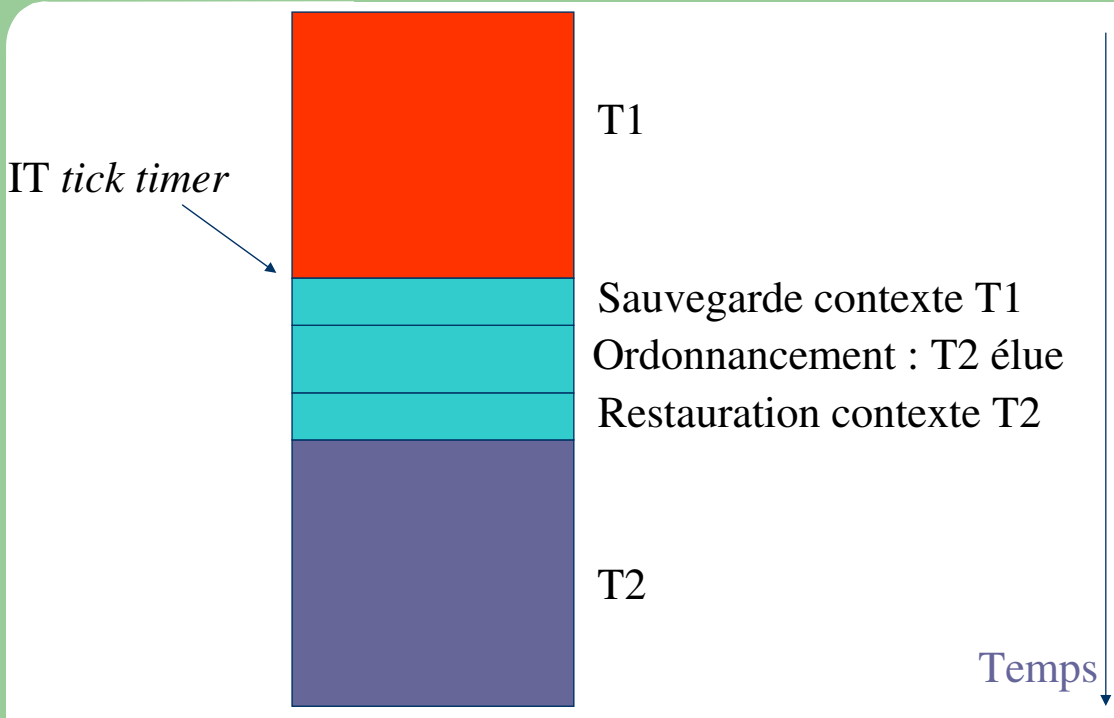
- Lors de l'élection de la tâche, l'ordonnanceur parcourt la liste des TCB pour élire la tâche $T_q$.
- Il faut que le temps de parcours de la liste des TCB soit le plus constant et le plus court possibles.
- On distingue alors l'ordre de l'ordonnanceur :
  - O(1) : durée fixe pour l'exécution de l'ordonnanceur quel que soit le nombre de tâches.
  - O(n) : durée d'exécution de l'ordonnanceur égale à n, le nombre de tâches.
  - O(n.log(n))…

# Conception d'un SE : le tick timer

- Les ordonnanceurs des SE modernes sont en O(1).
- L'ordonnanceur comme la manipulation de la liste de TCB est critique. Le processeur ne peut pas être interrompu (interruptions dévalidées).
- On a donc un zone critique de code non interruptible !



# Conception d'un SE : le tick timer



# FONCTIONNALITES GENERALES D'UN SE MULTITACHE



## Fonctionnalités d'un SE moderne

- Un SE multitâche multiutilisateur « moderne » implémente différentes fonctionnalités pour fournir des services aux applications :
  - Gestion du processeur.
  - Gestion de la mémoire.
  - Gestion des E/S.
  - Gestion des systèmes de fichiers.
  - Gestion de la concurrence.
  - Gestion de la protection.

# Gestion du processeur

- Le SE doit gérer l'attribution du processeur aux différentes tâches ou processus. On emploiera le terme de processus par la suite…
- C'est l'algorithme d'ordonnancement de l'ordonnanceur qui gère cette fonctionnalité.



# Gestion de la mémoire

- Le SE doit gérer l'allocation de la mémoire vive aux différents processus en utilisant les circuits matériels de la plateforme.
- Comme la mémoire physique réelle est insuffisante, la gestion mémoire se fait selon le principe de la mémoire virtuelle.
- A un instant donné, on ne chargera en mémoire vive que les parties utiles (pages) de code et de données d'un processus…

# Gestion des E/S

- Le SE doit gérer l'allocation de la mémoire vive aux différents processus en utilisant l'accès aux périphériques.
- Il gère le lien entre les appels système de haut niveau et les accès bas niveau.
- C'est le pilote de périphérique qui assure ce lien.



# Gestion des systèmes de fichiers

- La mémoire vive du SE est volatile.
- Il convient de stocker vers une mémoire de masse (disque dur) les données d'un processus.
- Dans l'embarqué, la mémoire de masse peut se réduire à de la mémoire FLASH.
- L'accès aux données s'appuie sur la notion de fichiers et de système de fichiers.

# Gestion de la concurrence

- Plusieurs processus sont en mémoire. Ils peuvent communiquer entre eux pour échanger des données.
- Le SE gère la synchronisation de l'accès aux données partagées pour assurer leur cohérence.
- Le SE offre donc des mécanismes de communication et de synchronisation entre processus…



# Gestion de la protection

- Le SE offre des mécanismes de protection aux fichiers et aux ressources physiques en introduisant des droits d'accès.
- Il faut en outre protéger le SE et la plateforme des programmes utilisateurs…

# FONCTIONNALITES AVANCEES D'UN SE MULTITACHE



## Mode utilisateur. Mode noyau

- Un processus s'exécute en mode utilisateur (*NIX) (*user mode*).
- Les actions possibles dans ce mode sont restreintes afin de protéger la plateforme.
- Au niveau du jeu d'instructions du processeur, on a accès à un ensemble réduit du jeu d'instructions complet (on ne peut pas « jouer » avec les interruptions).

# Mode utilisateur. Mode noyau

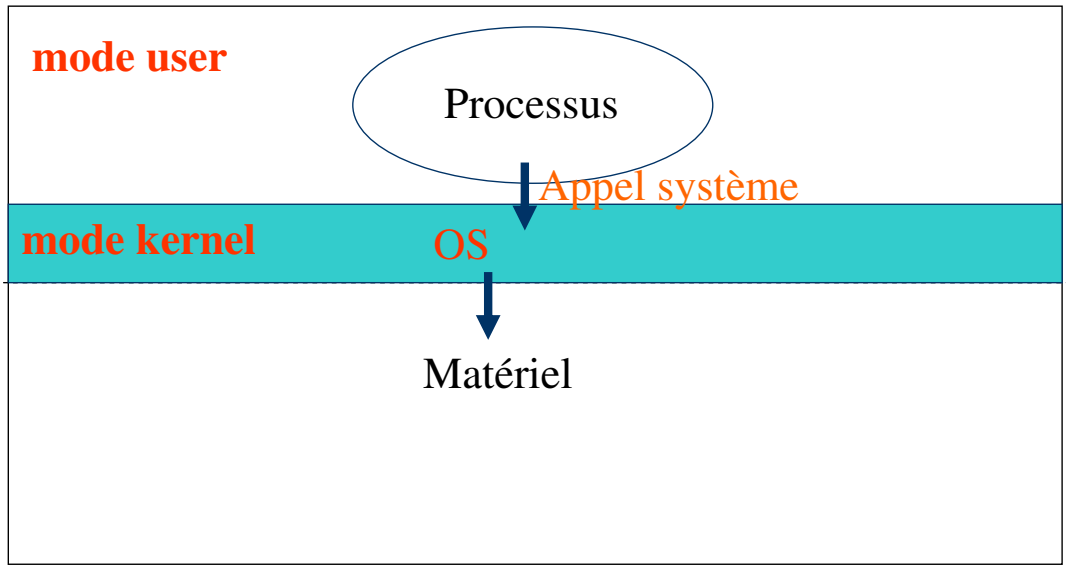
- Quand le processus exécute une primitive du SE, on dit qu'il exécute un appel système.
- Le processus passe dans le mode superviseur ou mode noyau (*NIX) (*kernel mode*).
- Le code exécuté sous la responsabilité du processeur a accès à l'ensemble du jeu d'instructions du processeur.



# Mode utilisateur. Mode noyau

- Généralement, le processeur possède au moins ces 2 modes d'exécution. Par exemple, pour le processeur 68000, on a le bit S dans le registre d'état SR :
  - S=1 : mode superviseur.
  - S=0 : mode utilisateur.
- L'exécution d'un appel système provoque l'exécution d'une instruction de TRAP qui permet de passer intrinsèquement dans le mode noyau :
  - 68000 : instruction assembleur TRAP.
  - X86 : instruction assembleur INT (INT 0x80 sous Linux).

# Mode utilisateur. Mode noyau

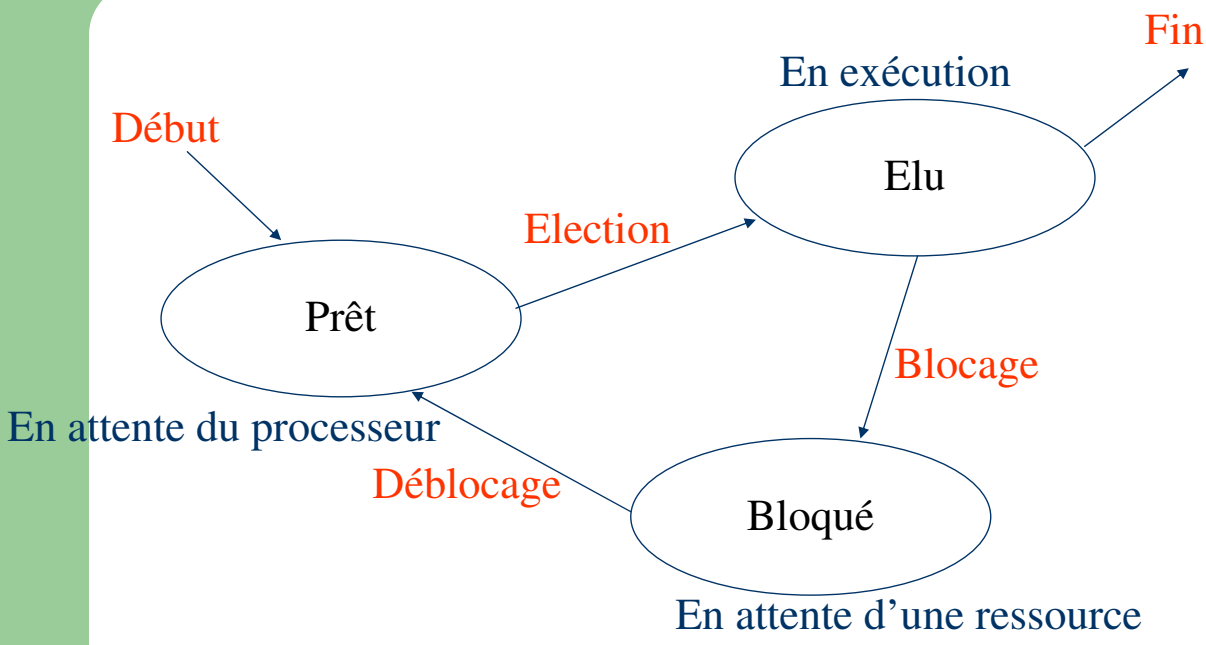





# Etats d'un processus

- On a vu que l'ordonnanceur élit un processus pour le prochain *tick* suivant l'algorithme d'ordonnancement utilisé.
- Cela dépend aussi de l'état dans lequel se trouvent les processus.
- Un processus peut avoir différents états :
  - Prêt (*ready*) : il est prêt à être exécuté.
  - Elu (*running*) : il est en cours d'exécution.
  - Bloqué (*pending*) : il est bloqué sur une ressource : opération d'E/S, appel système bloquant…

# Etats d'un processus

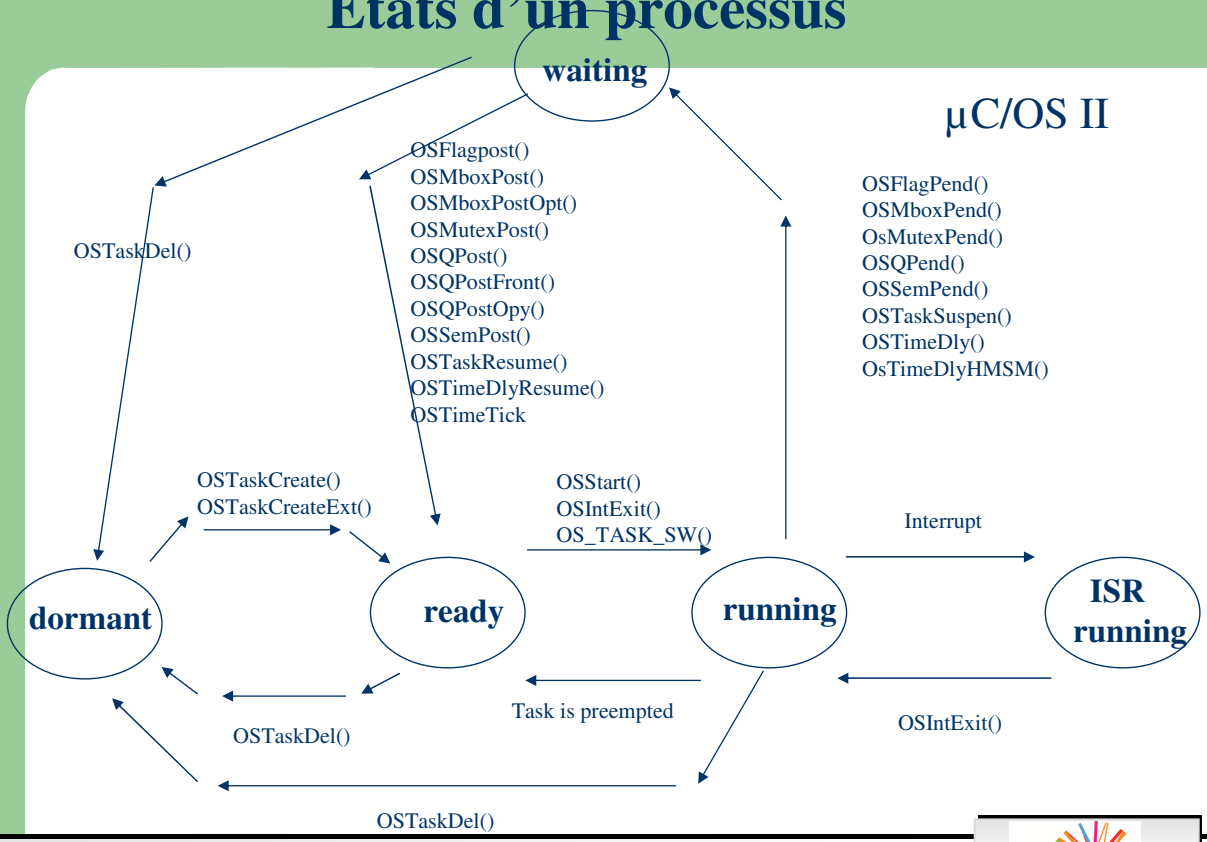





# Etats d'un processus

- Dans les SE modernes, il peut avoir plus d'états.
- On pendra l'exemple d'un SE Temps Réel : µC/OS II.

# Etats d'un processus






# Etats d'un processus

- On a besoin aussi d'avoir un SE réactif.
- Par exemple, pour un SE Temps Réel, il faut toujours donner l'accès au processeur à la tâche de plus forte priorité prête.
- Cela impose au SE d'être préemptif.
- On distingue donc :
  - Le SE avec un ordonnancement non préemptif.
  - Le SE avec un ordonnancement préemptif.

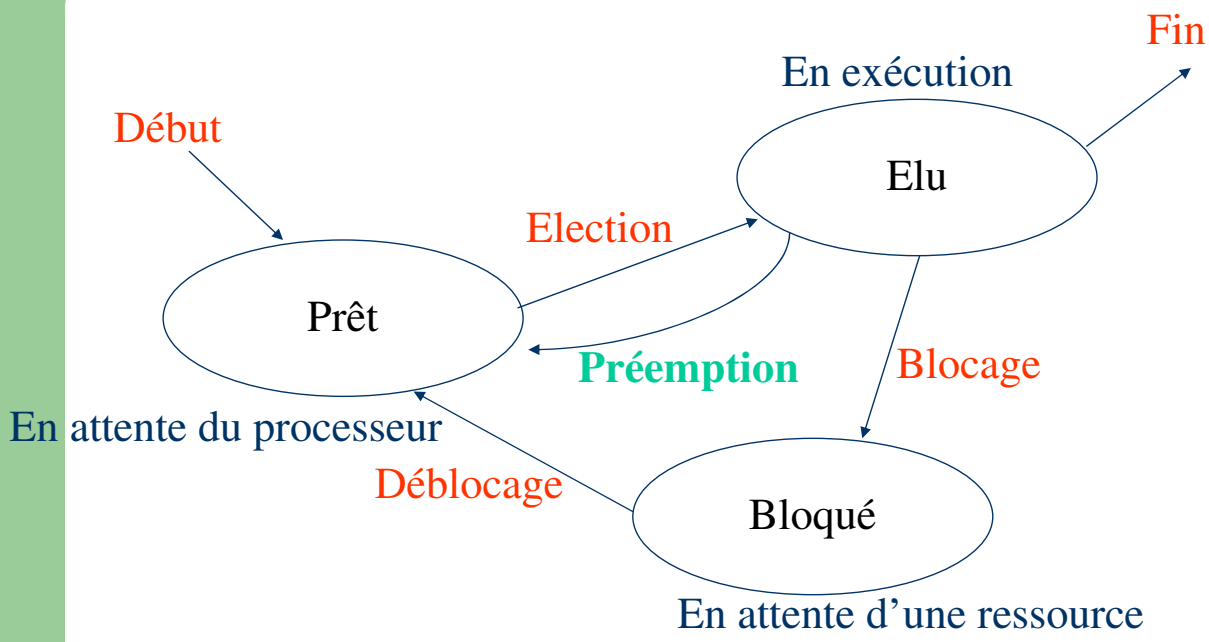
# Etats d'un processus

- Dans le cas d'un **SE non préemptif**, le processus dans l'état élu libère le processeur s'il a fini de s'exécuter, s'il décide de lui-même de passer dans l'état prêt ou s'il se bloque.

- Dans le cas d'un **SE préemptif**, le processus dans l'état élu libère le processeur s'il a fini de s'exécuter, s'il décide de lui-même de passer dans l'état prêt ou s'il se bloque ou **si le processeur est réquisitionné**.



# Etats d'un processus



# Etats d'un processus

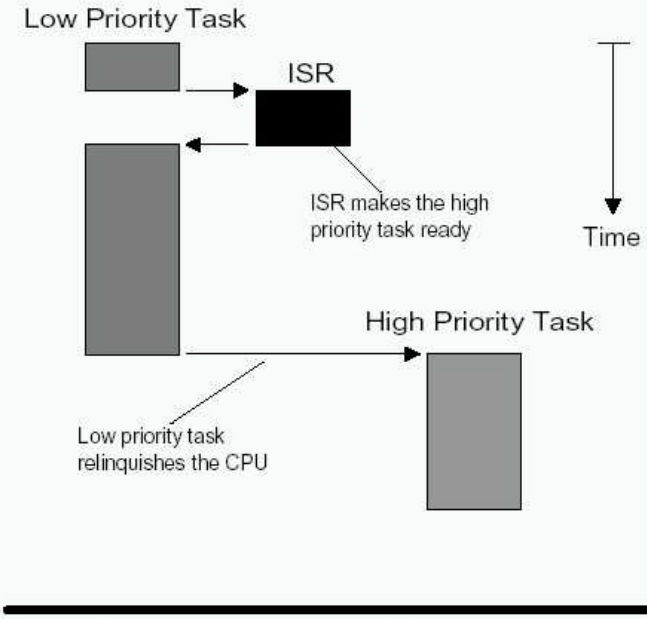



Figure 2. Non-preemptive scheduling



# Etats d'un processus

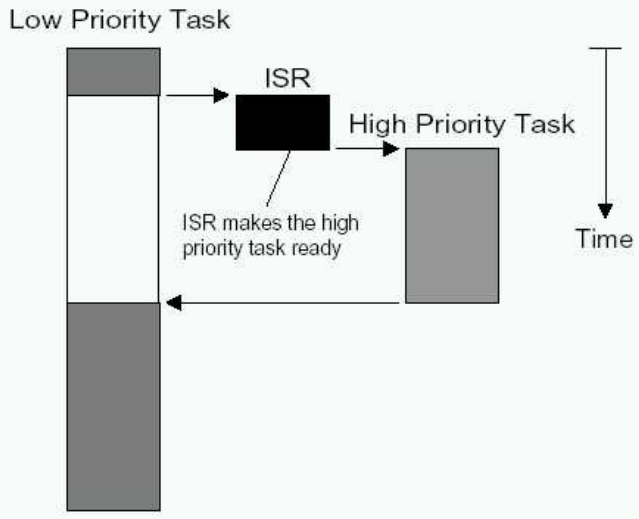



Figure 3. Preemptive scheduling

# Etats d'un processus

- Pour avoir un SE préemptif, il est nécessaire d'appeler explicitement l'ordonnanceur (fonction *schedule()* sous Linux) :
  - A la fin de tout appel système bloquant.
  - En retour d'interruption.



# Mémoire virtuelle

- On doit protéger la mémoire :
  - Plusieurs processus se partagent la mémoire.
  - Le matériel offre certains mécanismes de protection gérés par le système d'exploitation.
- On a introduit la notion de mémoire virtuelle :
  - On n'est plus limité par la taille de la mémoire physique.
  - On peut concevoir des programmes sans limitation de taille manipulant des quantités de données importantes en mémoire centrale.
  - Le SE fournit des mécanismes de gestion de la mémoire.

# Mémoire virtuelle

- La mémoire physique va être découpée en partitions de taille fixe :
  - Une partition = une page.
  - La mémoire est paginée. Gestionnaire mémoire par pagination.
  - Exemples de SE : *NIX, Windows NT, XP.
- La mémoire physique va être découpée en partition de taille variable :
  - Une partition = un segment.
  - La mémoire est segmentée. Gestionnaire mémoire par segmentation.
  - Exemple de SE : MSDOS.



# Mémoire virtuelle

- La conversion d'une adresse virtuelle en adresse physique :
  - Est obtenue par remplacement du numéro de page virtuelle dans l'adresse virtuelle par le numéro de page physique correspondant.
  - Est obtenue en utilisant une MMU (*Memory Management Unit*).
- Il est nécessaire de gérer des structures de données permettant la gestion de la mémoire :
  - Une table des pages logiques (pages virtuelles).
  - Une table des pages physiques (effectivement présentes en mémoire).

# Mémoire virtuelle

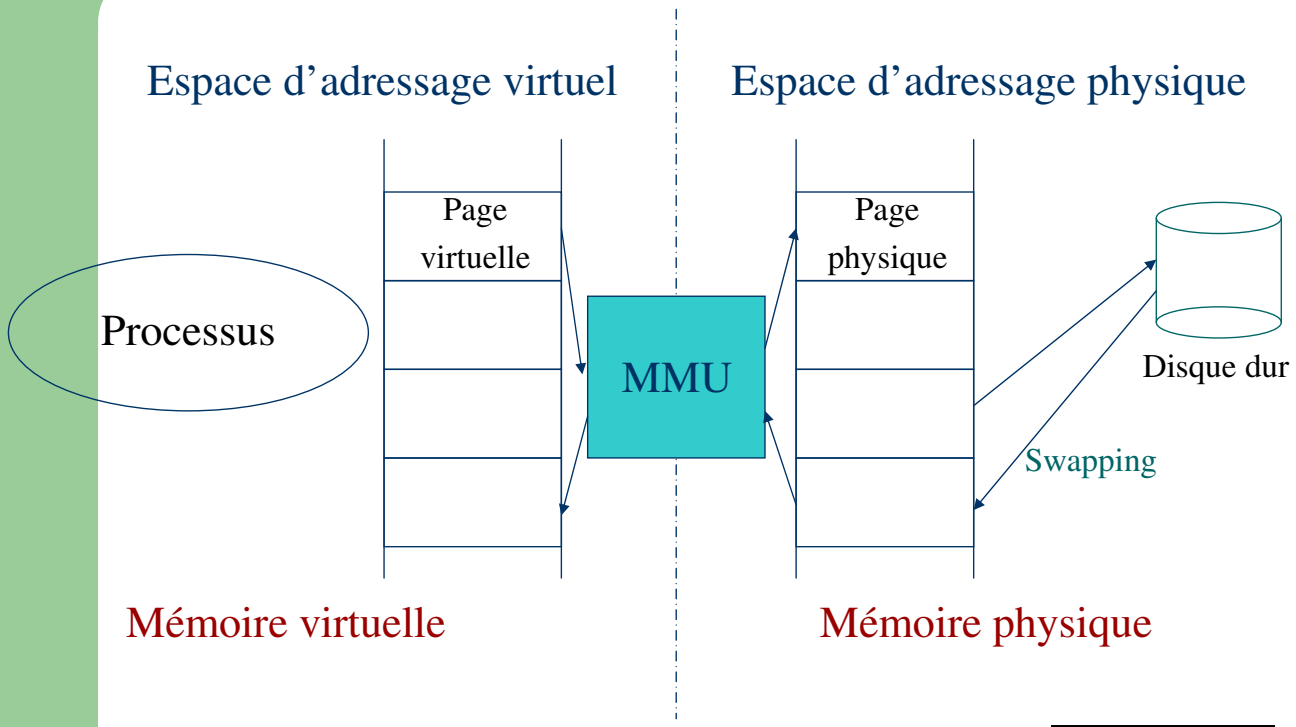
- Quand on n'a pas assez de mémoire physique pour l'ensemble des processus, on va libérer des pages physiques en les « swappant » sur une mémoire de masse secondaire comme un disque dur.
- On met donc en œuvre le *swapping*.
- Par la technique de mémoire virtuelle, chaque processus a l'impression d'avoir 4 Go de mémoire chacun dans le cas d'un processeur 32 bits !



# Mémoire virtuelle

- La MMU est un circuit électronique qui est intégré dans le processeur aujourd'hui.
- Elle apporte un niveau de protection matérielle et évite qu'un processus A aille écrire dans les données d'un processus B. Cela pose la question de comment le faire proprement. Si ce n'est pas le cas, on a l'occurrence d'un « *segmentation violation* » sous *NIX.

# Mémoire virtuelle






# Mémoire virtuelle

- La logique de décodage de la MMU prend un peu de temps.
- Dans le cas de l'embarqué, il est fréquent que la MMU soit dévalidée dans le portage du SE sur le système électronique.
- Dans ce cas, il y a confusion entre adresse logique et adresse physique.

# Mémoire virtuelle

- Avoir une MMU :
  - Assure une meilleure fiabilité du système.
  - Pour accéder à un périphérique, on doit écrire le pilote que l'on incorpore dans le SE.
  - Il est néanmoins possible d'accéder à un périphérique en mode user. Dans le cas de Linux, on écrit un pilote en mode user (*user mode driver*) en utilisant */dev/mem* par exemple... Un exemple : la bibliothèque svgalib.

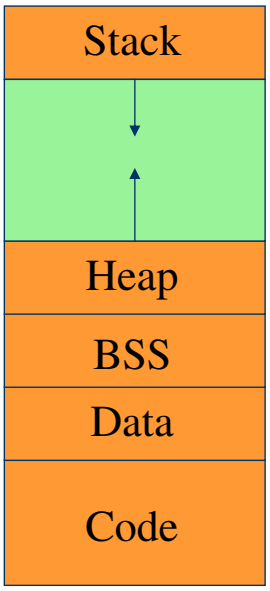


# Mémoire virtuelle

- Ne pas avoir de MMU ou la dévalider :
  - Adresse logique = adresse physique.
  - Permet d'être plus rapide.
  - C'est souvent le cas dans l'embarqué.
  - Le système peut être moins fiable car on accède directement à l'espace d'adressage.
  - Pour accéder à un périphérique, il n'y a plus de pilote à écrire. Néanmoins, il vaut mieux en écrire un pour garder l'homogénéité du SE.

# Mémoire virtuelle

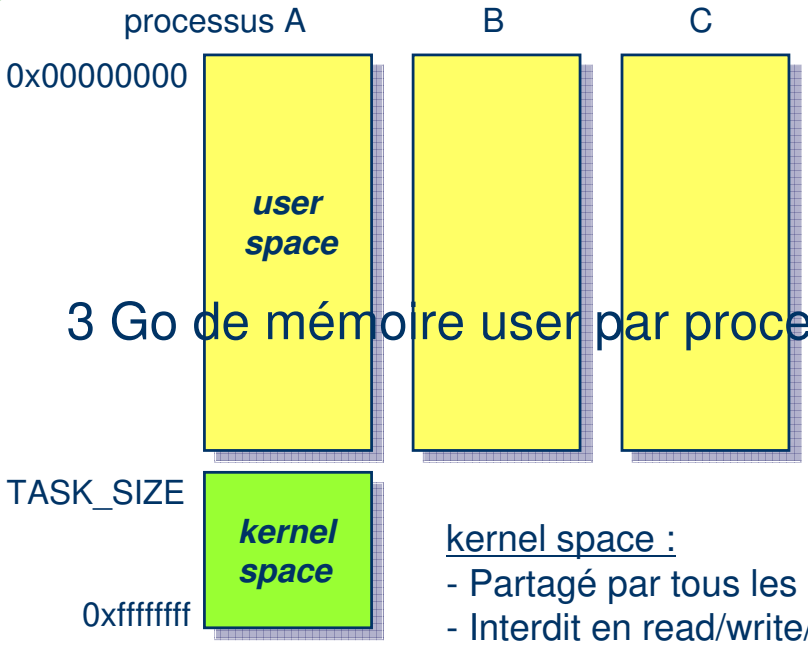
- Pile (*stack*) :
  - Retours d'appel de fonctions et variables locales aux fonctions.
  - Variables locales et paramètres.
- Tas (*heap*) :
  - Allocation dynamique de variables.
- Code (*Text*) :
  - Code objet en lecture seulement.
- Données :
  - Data : données initialisées.
  - BSS (*Bloc Starting by Symbol*) : données non initialisées.




Processus *NIX en mémoire



# Mémoire virtuelle




3 Go de mémoire user par processus

kernel space :
- Partagé par tous les processus
- Interdit en read/write/execute en user mode

**TASK_SIZE=0xc0000000 pour x86**
**0xbf000000 pour ARM**

# Mémoire virtuelle

**$ cat /proc/<PROCESS_ID>/maps**

```
00240000-003ae000 r-xp 00000000 08:01 1018138    /lib/libc-2.9.so
003ae000-003b0000 r--p 0016e000 08:01 1018138    /lib/libc-2.9.so
003b0000-003b1000 rw-p 00170000 08:01 1018138    /lib/libc-2.9.so
08048000-080fb000 r-xp 00000000 08:01 1171471    /bin/bash
080fb000-08100000 rw-p 000b2000 08:01 1171471    /bin/bash
08100000-08105000 rw-p 08100000 00:00 0
09b25000-09b67000 rw-p 09b25000 00:00 0          [heap]
b7de8000-b7fe8000 r--p 00000000 08:01 1826818    /usr/lib/locale/locale-archive
b7fe8000-b7fea000 rw-p b7fe8000 00:00 0
b7ffe000-b8000000 rw-p b7ffe000 00:00 0
b8000000-b8007000 r--s 00000000 08:01 2261245    /usr/lib/gconv/gconv-modules.cache
bf8f2000-bf907000 rw-p bffeb000 00:00 0          [stack]
. . .
```

Address Range — file offset — inode — file name

r: read
w: write
x: execute
s: shared
p: private (copy on write)

device major:minor



# Interpréteur de commandes

- Avec un SE moderne, on a généralement accès à un interpréteur de commandes ou *shell* :
  - C'est un processus utilisateur.
  - Il permet l'exécution des programmes.
  - Il permet de manipuler des fichiers.
  - Il permet d'accéder aux périphériques du système.
  - Il permet l'automatisation de procédures à l'aide de fichiers de commandes. Il existe donc des langages de programmation pour le *shell*.

# LES SE DE TYPE *NIX



# UNIX

- Le système UNIX créé en 1969 (Ken Thompson et Dennis Ritchie) est un système d'exploitation multitâche multiutilisateur. Il est indissociable du langage C.
- Il a plusieurs interpréteurs de commandes ainsi qu'un grand nombre de commandes et de nombreux utilitaires…
- C'est un SE très portable : il est possible de le mettre en oeuvre sur une multitude de plateformes matérielles.
- UNIX a une sécurité élevée et a une très grande connectivité réseaux.

# Linux

- Le système Linux créé en 1991 (Linus Torvald) est un système d'exploitation multitâche multiutilisateur. Il est UNIX like.
- Conçu initialement pour fonctionner sur plateforme *PC i386*, Linux a été porté sur « toutes » plateformes possibles.
- C'est un Logiciel Libre sous licence GPL !
- Il existe différents distributions Linux :
  - Slackware (1993), RedHat (1994), Fedora (2003), Debian, SuSe, Knoppix, Mandriva, Ubuntu, Damn Small Linux…



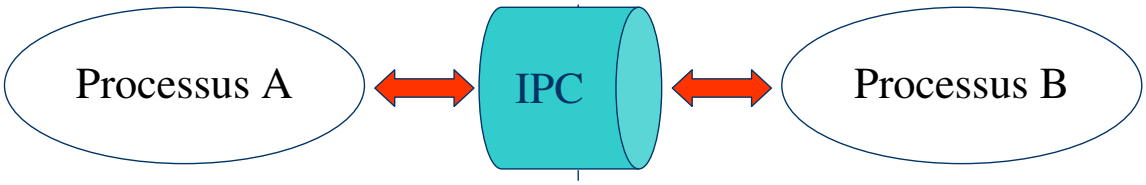
# COMMUNICATIONS INTERPROCESSUS : FORK, EXEC, SIGNAUX ET TUBES

# Introduction

- Nous allons considérer un SE de type *NIX.
- Nous allons lister quels sont les moyens de communication entre 2 processus locaux : communication interprocessus ou IPC (*Inter Processus Communications*).



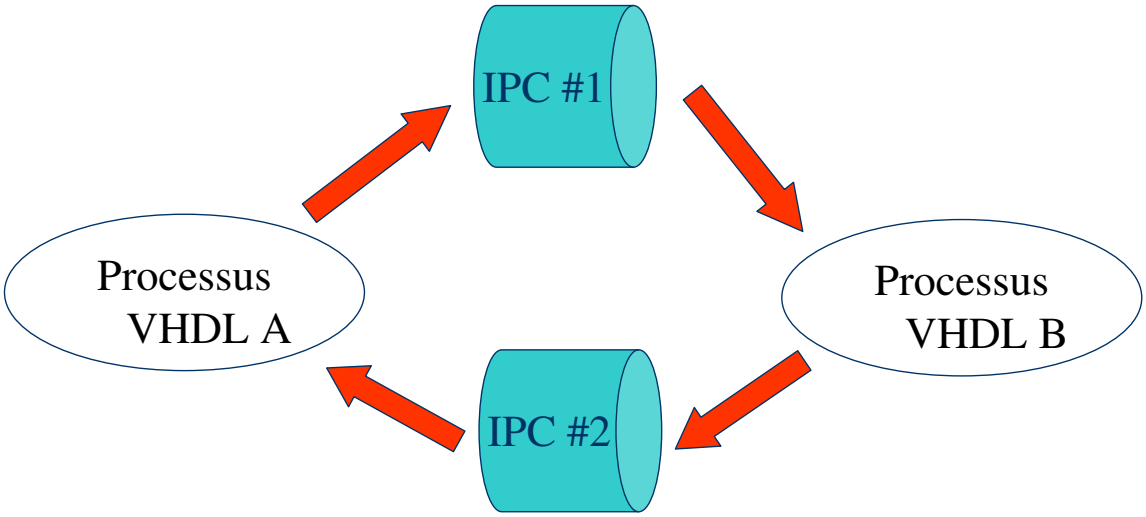
# Introduction




Communication bidirectionnelle

# Introduction

Analogie avec le langage VHDL :

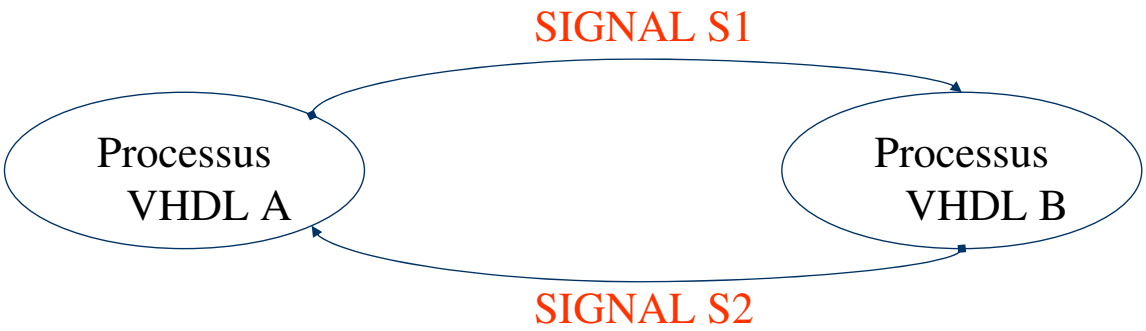



Quel est l'IPC utilisé en VHDL ?



# Introduction

Analogie avec le langage VHDL :




Le signal !

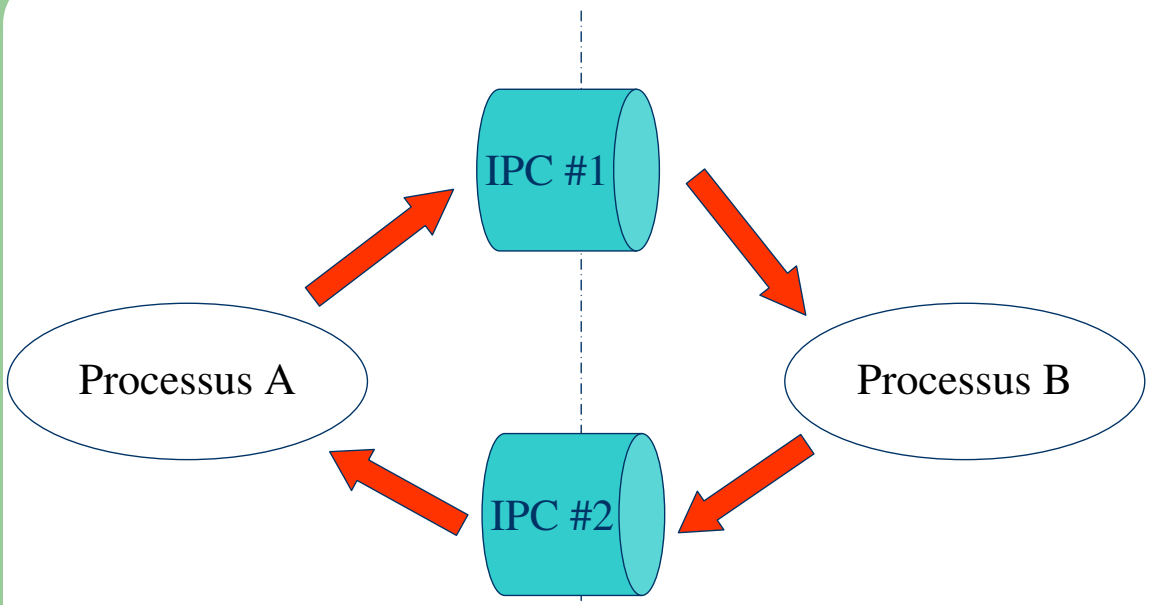
# Introduction

Analogie avec le langage VHDL :

- Le mécanisme de communication IPC entre 2 processus VDHL est le signal.
- Un simulateur VHDL est un SE en soi : chaque processus VHDL peut être comparé à un processus d'un SE, un signal VHDL entre 2 processus VHDL correspondrait à un IPC du SE !
- Il est alors possible d'imaginer de simuler du code VHDL avec cette analogie :
  - Processus VHDL = processus du SE
  - Signal VHDL = IPC du SE (file de messages)
- Exemple : SystemC (bibliothèque C++ de simulation haut niveau de VHDL)



# Introduction



# Introduction

- Deux processus locaux peuvent communiquer en utilisant les mécanismes de communication suivants :
  - Les fichiers.
  - Les signaux.
  - Les tubes (*pipe*) et les tubes nommés FIFO (*named pipe*).
  - Les IPC UNIX System V (ou IPC tout court) :
    - Files de messages.
    - Mémoire partagée.
    - Sémaphores.



# Introduction

- Nous allons aussi regarder :
  - Le mécanisme de création d'un processus à partir d'un processus : la « fourche » (*fork*).
  - Le mécanisme de recouvrement : *exec*.

# fork() et exec()

- L'appel système *fork()* permet de créer un nouveau processus fils à partir d'un processus père par duplication.
- Afin de distinguer le père du fils, *fork()* retourne :
  - -1 : en cas d'erreur de création.
  - 0 : pour le procesus fils.
  - > 0 : le *pid* (*Process IDentifier*) du processus fils au processus père.
- Les appels système *getpid()* et *getppid()* permettent d'identifier un processus et son père.



# fork() et exec()

```
#include <unistd.h>

pid_t fork(void) // crée un nouveau processus
pid_t getpid(void) // donne le pid du processus
pid_t getppid(void) // donne le pid du père du processus
```

# fork() et exec()

```
#include <stdio.h>
#include <unistd.h>
int main(void){
int pid;

pid = fork();
if (pid > 0) {
printf("processus pere: %d-%d-%d\n", pid, getpid(), getppid());
}
if (pid == 0) {
printf("processus fils: %d-%d-%d\n", pid, getpid(), getppid());
}
if (pid < 0){
printf("Probleme de creation par fork()\n"); }
return 0;
}
```

A l'exécution :

```
% ./fork
processus pere: 7121-7120-7070
processus fils: 0-7121-1
```



# fork() et exec()

- Le processus fils hérite de beaucoup d'attributs du père comme les descripteurs de fichiers ouverts (*file descriptor*) mais n'hérite pas :
  - De l'identification de son père.
  - Des temps d'exécution qui sont initialisés à 0.
  - De la priorité du père. La sienne est initialisée à la valeur standard.
  - …
- Le fils travaille sur les données du père s'il les accède seulement en lecture. Si le fils accède en écriture à une donnée du père, celle-ci est recopiée dans l'espace local du fils.

# fork() et exec()

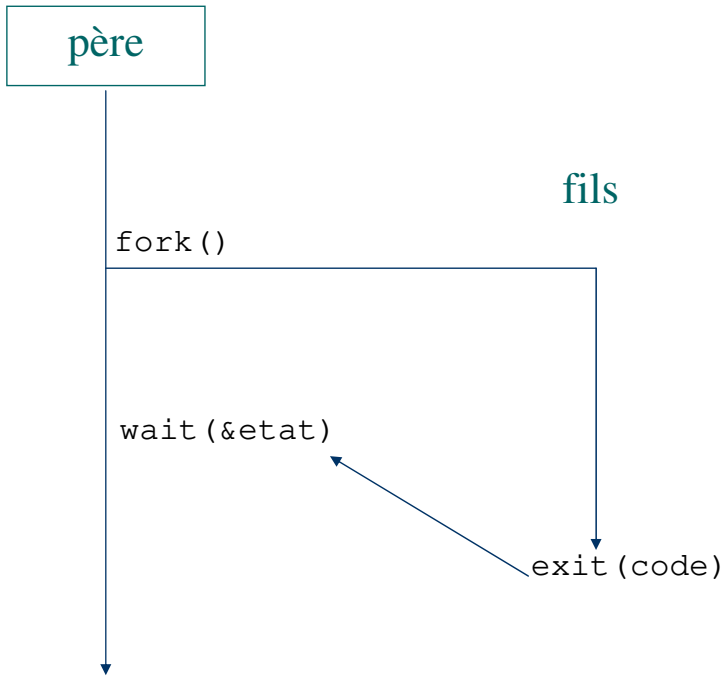
- Les appels système *wait()* et *waitpid()* permettent l'élimination des processus « zombis » et la synchronisation d'un processus sur la terminaison de ses descendants avec récupération des informations relatives à cette terminaison.
- Elles provoquent la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine.
- L'appel système *waitpid()* permet de sélectionner un processus particulier parmi les processus fils (*pid*).



# fork() et exec()

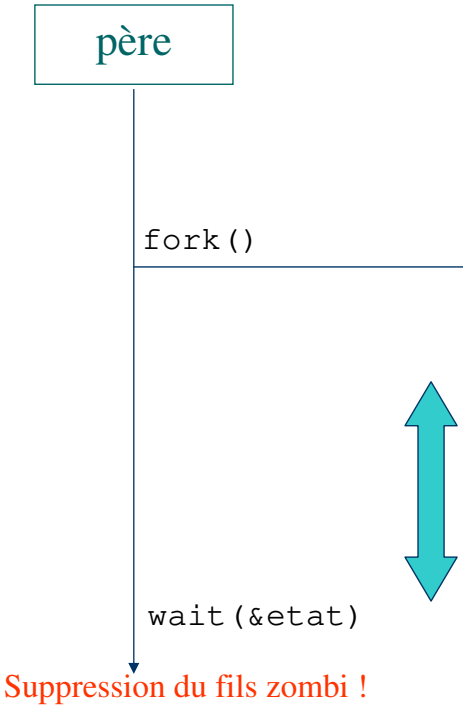
```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *etat);
pid_t waitpid(pid_t pid, int *etat, int options);
```

# fork() et exec()






# fork() et exec()




Processus zombi

Le fils n'a plus son code ni ses données en mémoire. Seules les infos utiles pour le père sont conservées.

Suppression du fils zombi !

# fork() et exec()

- Un processus qui meurt devient un zombi jusqu'au *wait() :*
  - Si le père meurt avant son fils, il est adopté par un autre processus (processus *init* de *pid* égal à 1 en général).
  - Si le père n'attend pas la mort de son fils alors son fils reste un zombi.
- Sur certains SE, les zombis ne peuvent pas être supprimés !



# fork() et exec()

- Les appels système de recouvrement permettent de charger en mémoire à la place du processus fils un autre programme provenant d'un fichier binaire.
- Il existe 6 primitives de type *exec().*
- Les arguments sont transmis :
  - Sous forme d'un tableau (ou vecteur, d'où le *v*) contenant les paramètres : *execv(), execvp(), execve().*
  - Sous forme d'une liste (d'où le *l*) : arg0, arg1, …, argN, NULL : *execl(), execlp(), execle().*
  - La lettre finale *p* ou *e* est pour *path* ou *environment*.

# fork() et exec()

```
#include <stdio.h>
#include <unistd.h>
int res;
int main(void) {

printf("Avant exec\n");

if (fork()==0) {
   res = execl("/bin/ls", "NULL" , 0);
   printf("%d\n", res);     // -1 en cas de probleme
   exit(res);               // Terminaison fils
}
wait(NULL);                 // Pere qui attend le fils
printf("Apres exec\n");
return 0;
}
```

A l'exécution :

```
% ./exec
Avant exec
bin  Bureau  Desktop  PCB  pub
   research  src  training
Apres exec
```



# Les signaux

- Un signal ressemble à une interruption logicielle.
- La réception d'un signal interrompt le traitement en cours du processus et provoque l'exécution automatique de la fonction associée au signal (par analogie, l'ISR en cas d'occurrence d'une interruption).
- L'association entre le numéro du signal et la fonction est réalisée par l'appel système *signal()*.
- La fonction ne sera pas exécutée une deuxième fois si le traitement du signal n'est pas réactivé par un nouvel appel à la fonction *signal( )*.

# Les signaux

- Un signal est envoyé par un processus, reçu par un autre processus (éventuellement le même) et est transporté par le noyau.

- Quand un processus reçoit un signal :
  - Il interrompt son traitement en cours.
  - Il exécute la fonction de traitement du signal.
  - Il reprend l'exécution du traitement interrompu.



# Les signaux

- Il existe différents comportements associés à la réception d'un signal :
  - Un comportement par défaut est défini : terminaison anormale du processus (ex : violation de mémoire, division par zéro).
  - Le processus peut ignorer le signal.
  - Le processus peut redéfinir par une fonction de traitement spécifique son comportement à la réception d'un signal.

# Les signaux

- Origine des signaux :

Frappe de caractères.

| touche | signal |
|---|---|
| CTRL-C | SIGINT |
| CTRL-\ | SIGQUIT |
| CTRL-Z | SIGSTP |

Par l'utilisateur depuis le terminal vers le processus en avant-plan

Erreur de programme.

Violation de mémoire : SIGSEGV

Division par zéro : SIGFPE

Par le noyau vers le processus concerné

Commande *NIX *kill.*

Appels système : *kill(), alarm().*

Par un processus quelconque vers un processus quelconque si autorisé



# Les signaux

```
$ kill -l
```

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      33) SIGRTMIN    34) SIGRTMIN+1
35) SIGRTMIN+2  36) SIGRTMIN+3  37) SIGRTMIN+4  38) SIGRTMIN+5
39) SIGRTMIN+6  40) SIGRTMIN+7  41) SIGRTMIN+8  42) SIGRTMIN+9
43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13
47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

# Les signaux

```
1  SIGHUP      terminaison du processus leader
2  SIGINT      frappe d'interruption (CTRL-C)
3  SIGQUIT     frappe de quit (CTRL-\)
4  SIGILL      instruction illégale
6  SIGABRT     problème matériel
7  SIGBUS      erreur sur le bus
8  SIGFPE      erreur arithmétique
9  SIGKILL     signal de terminaison. Ne peut être ignoré ou
               redéfini
10 SIGUSR1     signal utilisateur
11 SIGSEGV     violation écriture mémoire
12 SIGUSR2     signal utilisateur
13 SIGPIPE     écriture sur un tube non ouvert en lecture
14 SIGALRM     fin de temporisateur
15 SIGTERM     terminaison normale d'un processus
17 SIGCHLD     terminaison (arrêt) d'un fils
18 SIGCONT     continuation d'un processus arrêté par SIGSTOP
19 SIGSTOP     signal de suspension d'exécution de processus
20 SIGTSTP     frappe du caractère de suspension sur le
   clavier (CTRL-Z)
```



# Les signaux

- Envoi d'un signal à un processus :

```
#include <signal.h>
int kill(pid_t pid,int sig)
   pid : > 0 : pid de processus destinataire
   sig : le signal à envoyer
```

- Envoi du signal SIGALRM au processus en cours dans *n* s :

```
#include <unistd.h>
unsigned alarm (unsigned n)
```

# Les signaux

- Signal à intercepter et à traiter :

```
#include <signal.h>
void *signal(int signum, void (*handler));
   signum : le signal à intercepter
   handler : pointeur sur une fonction qui gère le signal
             SIG_DFL : comportement par défaut
             SIG_IGN : ignorer le signal
```

- Endort le processus appelant jusqu'à ce qu'il reçoive un signal :

```
#include<unistd.h>
int pause(void);
```



# Les signaux

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void it_fils(){
printf("\t Interruption :
kill SIGGINT\n");
kill(getpid(),SIGINT);
}

void fils(){
signal(SIGUSR1,it_fils);
printf("Looping !\n");
while(1)
   pause();
}
```

```
int main(){
int pid ;

if ((pid=fork())== 0)
   fils();
else {
   sleep(3);
   printf("kill
   (pid,SIGUSR1)\n") ;
   kill (pid,SIGUSR1) ;
}
return 0 ;
}
```

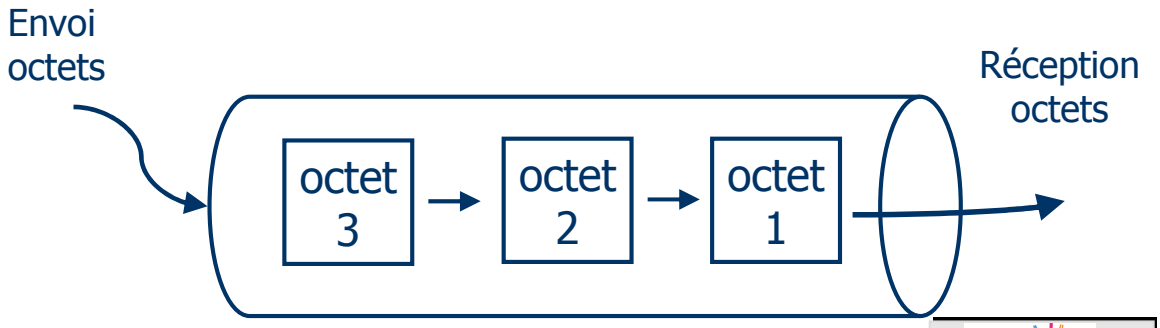
A l'exécution :

```
% ./sig
Looping !
kill (pid,SIGUSR1)
Interruption : kill SIGINT
```

# Les tubes et les tubes nommés

- Les tubes sont un autre moyen de communication entre 2 processus locaux.

- On distingue :
  - Les tubes ordinaires (*pipe*).
  - Les tubes nommés (*named pipe*) qui possèdent une entrée dans le système de fichiers.






# Les tubes et les tubes nommés

- Un tube est traité comme un fichier ouvert.

- Un tube est cependant un fichier particulier :
  - Il n'a pas de nom dans l'arborescence des fichiers.
  - Il est connu par deux descripteurs de fichier : un en lecture, un en écriture.

# Les tubes et les tubes nommés

- On lit/écrit dans un tube comme on lit/écrit dans un fichier séquentiel :
  - Utilisation des appels système *read()* et *write()*.
  - Les données ne sont pas structurées : le tube ne contient qu'un flux d'octets non structurés.
- Un tube est comme un « tuyau » avec une entrée et une sortie :
  - Plusieurs processus peuvent y écrire des données.
  - Plusieurs processus peuvent y lire des données.



# Les tubes et les tubes nommés

- Le fonctionnement d'un tube est de type *First In First Out* :
  - On lit les octets dans l'ordre dans lequel ils ont été écrits dans le tube.
  - Il n'est pas possible de lire un octet sans voir lu tous ceux qui le précèdent.
  - Un octet lu par un processus ne pourra jamais être relu dans le tube par un autre processus.
- Les droits d'accès du tube sont du même genre que les droits d'accès *NIX sur les fichiers (rwx, rwx, rwx).

# Les tubes et les tubes nommés

- Si un tube est vide ou si tous les descripteurs susceptibles d'y écrire sont fermés, l'appel système *read()* renvoie la valeur 0 (fin de fichier atteinte).

- Un processus qui écrit dans un tube qui n'a plus de processus lecteurs (tous les descripteurs de fichier des processus susceptibles de lire sont fermés) reçoit le signal SIGPIPE.



# Les tubes et les tubes nommés

- Création :

```
#include <unistd.h>
int pipe(int p_desc[2])
    p_desc[0] pour la lecture
    p_desc[1] pour l'écriture
```

# Les tubes et les tubes nommés

- E/S :

```
write(p_desc[1], buf_ecrire, nbre_octets)
read(p_desc[0], buf_lire, nbre_octets)
```



# Les tubes et les tubes nommés

```
#include <stdio.h>
#include <unistd.h>

main(){
int i, ret, p[2];
char c;

pipe(p);

write(p[1], "AB", 2);

for (i=0; i<2; i++) {
   read(p[0], &c, 1);
   printf("valeur lue: %c\n", c);
}
}
```

A l'exécution :

```
% ./pipe
valeur lue: A
valeur lue: B
```

# Les tubes et les tubes nommés

- Un tube ordinaire est adapté pour les communications entre 2 processus parents. Ce n'est pas adapté pour des communications entre 2 processus indépendants (préférer le tube nommé).
- Pour des communications bidirectionnelles entre un processus père et un processus fils, il faut 2 tubes.
- Un processus qui lit un tube vide est bloqué : c'est le comportement par défaut de *read()*.
- Un processus qui écrit dans un tube plein (4 Ko) est bloqué : c'est le comportement par défaut de *write()*.



# Les tubes et les tubes nommés

- Un tube ordinaire n'existe pas dans le système de fichiers et n'est accessible uniquement qu'aux processus d'une même filiation.
- Les tubes nommés ou FIFO ont une entrée dans le système de fichiers et sont utilisables entre processus indépendants.
- Aucune donnée n'est enregistrée sur le disque dur, les données du tube nommés restant en mémoire.

# Les tubes et les tubes nommés

```
#include <unistd.h>

int mknod(const char *nom_fich, mode_t mode, dev_t dev

int mkfifo (const char *pathname, mode_t mode);

int unlink (const char * path);
```

- On utilisera ensuite les appels système classiques *read(), write()…*



# Les tubes et les tubes nommés

```
#include<sys/stat.h>
#include<sys/types.h>
#include<stdio.h>

main () {
char *chemin = "ma_fifo";

printf("Creation d'une FIFO : ");
if ((mkfifo (chemin, 0600))==-1) {
        printf("BAD\n");
        exit(-1);
}
else
        printf("OK\n");
exit(0);
}
```

A l'exécution :

```
% ./fifo
Creation d'une FIFO : OK
% ls -l ma_fifo
prw-------   1 kadionik ens_e          0
   Mar 13 12:49 ma_fifo
```

# Les tubes et les tubes nommés

- Il est possible de créer un tube nommé avec la commande *NIX *mkfifo* :

```
% mkfifo toto
% ls -l toto
/net/ens/kadionik/src
prw-r--r--   1 kadionik ens_e          0 Mar 13 13:10 toto
% rm toto
% mkfifo fifo
% echo "coucou" >> fifo &
[1] 9699
% cat < fifo
coucou
[1]+  Done                    echo "coucou" >>fifo
%
```



# COMMUNICATIONS INTERPROCESSUS : IPC SYSTEM V

# Introduction

- Les IPC UNIX System V (ou IPC tout court) recouvrent 3 mécanismes de communications interprocessus :
  - Files de messages.
  - Mémoire partagée.
  - Sémaphores.



# Les sémaphores

- Un sémaphore est une variable protégée (ou un type de donnée abstrait) et constitue la méthode utilisée couramment pour restreindre l'accès à des ressources partagées (par exemple un espace de stockage) dans un environnement de programmation concurrente.
- Le sémaphore a été inventé par Edsger Dijkstra.

# Les sémaphores

- On manipule un sémaphore de façon **atomique** : la fonction de manipulation d'un sémaphore ne pourra être interrompue ou préemptée par quoi que ce soit.
- C'est un temps mort de non réponse du SE (section critique pour le SE).
- 3 opérations atomiques de manipulation d'un sémaphore existe :
  - Initialisation d'un sémaphore.
  - Prise du sémaphore.
  - Libération du sémaphore



# Les sémaphores

```
Init(sem, n)
   disable_interrupt
   valeur[sem] = n
   enable_interrupt

P(sem)
   disable_interrupt    (ordonnanceur dévalidé !)
   valeur[sémaphore] = valeur[sémaphore] - 1
   si (valeur[sémaphore] < 0) alors
              étatProcessus = Bloqué
              mettre processus en file d'attente
   finSi
   invoquer l'ordonnanceur
   enable_interrupt
```

# Les sémaphores

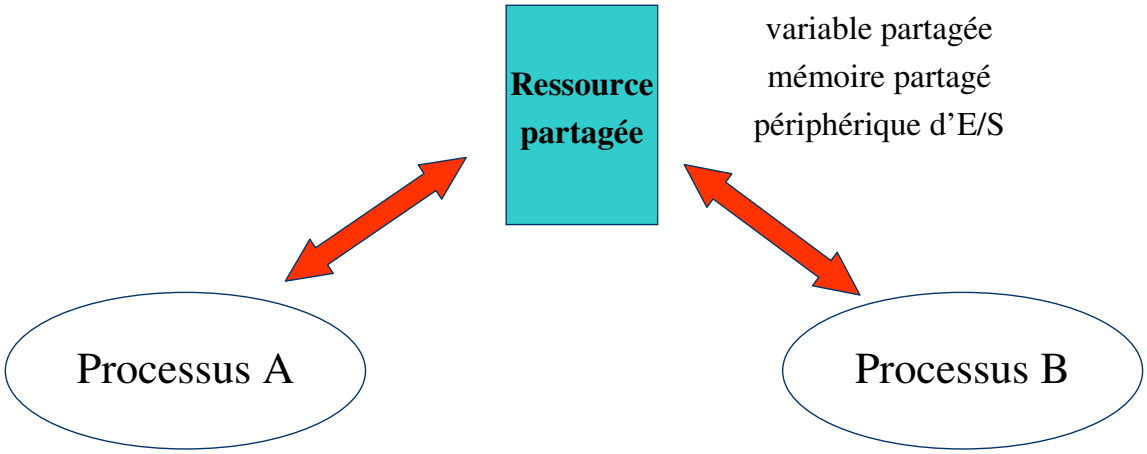
```
V(sem)
   disable_interrupt
   valeur[sémaphore] = valeur[sémaphore] + 1
   si (valeur[sémaphore] == 0) alors
              extraire processus de file d'attente
              étatProcessus = Prêt
   finSi
   invoquer l'ordonnanceur
   enable_interrupt
```



# Les sémaphores

- On parle de sémaphore à compteur quand le sémaphore est initialisé à une valeur > 1.
- On parle de sémaphore binaire quand le sémaphore est initialisé à 1. On parle aussi de *mutex* (POSIX).
- Le sémaphore permet de gérer 2 cas de figure complexes :
  - Accès exclusif à une ressource partagée et basé sur le principe de l'exclusion mutuelle : variable partagée, mémoire partagé ou périphérique d'E/S.
  - Synchronisation de processus.

# Les sémaphores




Accès exclusif à une ressource partagée



# Les sémaphores

```
Init(sem, 1)
```

```
. . .
P(sem)
Accès ressource partagée
V(sem)
. . .
```

```
. . .
P(sem)
Accès ressource partagée
V(sem)
. . .
```

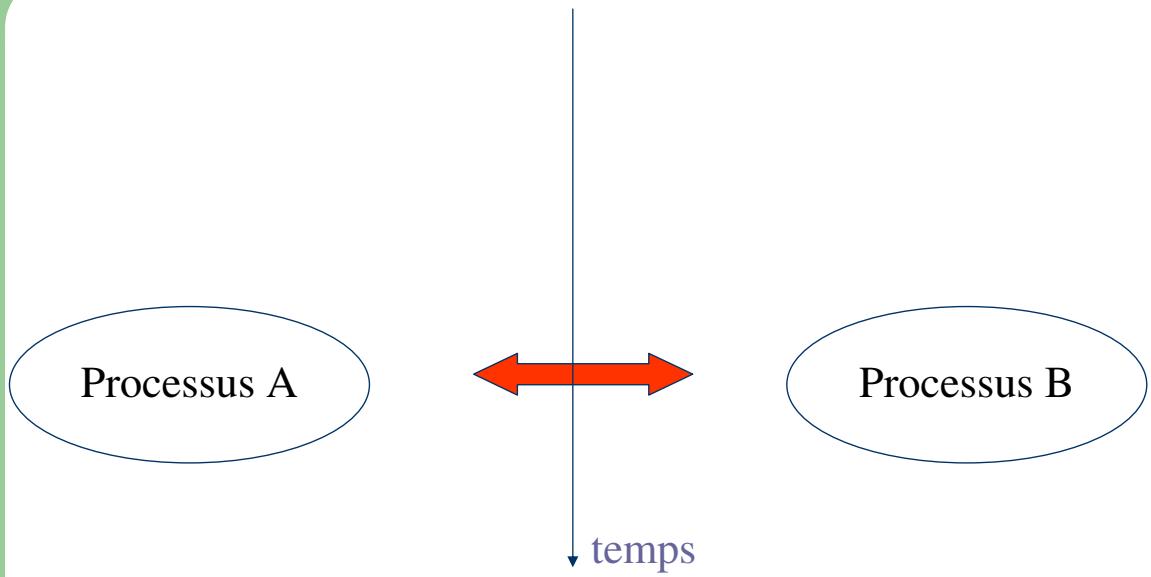
Processus A

Processus B

Accès exclusif à une ressource partagée

- Le sémaphore binaire est initialisé par un seul processus à **1**.

# Les sémaphores

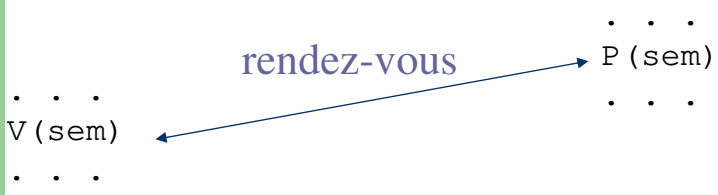



Synchronisation de processus : le rendez-vous



# Les sémaphores

```
Init(sem, 0)
```





Synchronisation de processus : le rendez-vous

- Le sémaphore binaire est initialisé par un seul processus à **0**.

# Manipulation des objets IPC

- Les objets IPC sont :
  - Files de messages.
  - Segments de mémoire partagée.
  - Sémaphores.
- A chaque objet IPC sera associée une clé d'identification de 32 bits.
- Cette identification définie dans le source du programme est analogue à un nom de fichier : on crée ou on cherche un objet IPC à partir d'une valeur de clé donnée…



# Manipulation des objets IPC

- On ne peut pas prévoir si un autre programmeur n'a pas par hasard utilisé la même clé pour un autre objet IPC.
- On pourra donc utiliser une fonction spéciale *ftok()* pour créer une clé unique à partir d'un nom de fichier existant dans le système de fichiers et d'un numéro de projet.

```
# include <sys/ipc.h>
key_t  ftok(const char *pathname, int proj_id);
```

# Manipulation des objets IPC

- Flags :
  - IPC_CREAT : crée un objet IPC s'il n'existe pas.
  - IPC_NOWAIT : ne pas attendre.
  - …
- Commandes :
  - IPC_RMID : suppression de l'objet.
  - IPC_SET : mise à jour des attributs de l'objet.
  - IPC_STAT : lecture des attributs de l'objet.
- Permissions : analogues à celles du système de fichiers *NIX.



# Manipulation des objets IPC

- On a les commandes *NIX suivantes pour manipuler les objets IPC :
  - Commande *ipcs* : liste des objets IPC existants.
  - Commande *ipcrm* : suppression d'un objet IPC.

```
% ipcs
T          ID      KEY         MODE         OWNER    GROUP
Message Queues:
q         800    0x4252c6a0 --rw-------    quidam       t2
q          11    0x421aa794 --rw-------    quidam       i1
Shared Memory:
m        1500    0x6f26     --rw-rw-rw-     tang       t2
Semaphores:
%
```

# Objet IPC : file de messages

- Contrairement aux tubes, une file de messages contient des données structurées composées :
  - D'un type de message.
  - D'un message de longueur variable.
- A la différence des tubes, il est possible d'attendre uniquement des messages d'un type donné : s'il n'y a que des messages d'un autre type, le processus sera bloqué par défaut.
- Comme les tubes, les files de messages sont gérées selon le principe *First In First Out*.



# Objet IPC : file de messages

- L'attente est par défaut bloquante mais comme pour les tubes, il est possible d'altérer le comportement par défaut en positionnant le flag IPC_NOWAIT. Le processus recevra un code d'erreur lui indiquant s'il a ou non lu un message (-1 avec *errno* égal à EAGAIN).
- Comme pour les tubes, les files de messages ont des droits d'accès de type *NIX.

# Objet IPC : file de messages

- Création :

```
int msgget(key_t key, int flg)
   retour = msq_id
```

- Emission :

```
int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int
   flg);

struct msgbuf {
long mtype;
char *mtexte;
}
```



# Objet IPC : file de messages

- Réception :

```
int msgrcv(int msqid, struct msgbuf *msgp,int msgsz, long
   msgtyp, int flg);
   msgtyp = 0 : premier message lu
   msgtyp > 0 : premier message de type msgtyp
   msgtyp < 0 : premier message avec un type ≤ |msgtyp|
```

- Opérations de contrôle :

```
int msgctl(int f_id, int cmde, struct msqid_ds *structure)
   cmde = IPC_RMID, IPC_STAT, IPC_SET
   msqid_ds : structure définie dans <sys/msg.h> qui donne
   des informations sur la file comme les droits d'accès,
   dernier processus écrivain…
```

# Objet IPC : file de messages

- On peut définir son propre type de message. Il faut respecter les consignes suivantes :
  - Le premier membre de la structure doit être un *long*.
  - Les autres membres en nombre quelconque, peuvent avoir n'importe quel type de base sauf le type pointeur.

```
typedef struct {
   long type;
    int tab[64];
   float x, y;
} mon_msgbuf ;
```



# Objet IPC : file de messages

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

extern int errno;

main()
{
key_t cle=0x99990000;
int msgid;
```

# Objet IPC : file de messages

```
printf("Creation file message : ");
if((msgid=msgget(cle, IPC_CREAT|0666)) == -1) {
        printf("BAD. Cause errno=%d\n", errno);
        exit(-1);
}
printf("OK. msgid=%d\n", msgid);

exit(0);
}
```

A l'exécution :

```
% ./msg
Creation file message : OK. msgid=908
% ipcs
IPC status from <running system> as of Fri Mar 13 17:19:08
   MET 2009
T         ID      KEY        MODE        OWNER    GROUP
Message Queues:
q        908    0x99990000 --rw-rw-rw- kadionik    ens_e
```



# Objet IPC : file de messages

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>
#include <string.h>

key_t cle = 0x99990001;
#define TYPE_MESS 1
struct ordre {
long type;
char texte[50]; // nom de l'action
int nombre;     // nombre action a vendre
} message;

int longueur = sizeof(message) - sizeof(long);
```

# Objet IPC : file de messages

```
main(int argc, char *argv[]) {
int msgid;

if((msgid=msgget(cle, 0600|IPC_CREAT)) == -1) {
        perror("msgget");
        exit(-1);
}
```



# Objet IPC : file de messages

```
// Passage d'ordre d'un client
if(strcmp(argv[1], "client") == 0) {
        message.type = TYPE_MESS;  // Ordre de vente
        message.nombre = atoi(argv[2]);
        sprintf(message.texte, "Vendre %d actions zorglub",
  message.nombre);
        if(msgsnd(msgid, &message, longueur, 0) == -1) {
                perror("msgsnd");
                exit(-1);
        }
}
```

# Objet IPC : file de messages

```
// Traitement par l'agent de change
if(strcmp(argv[1], "agent") == 0) {
        if(msgrcv(msgid, &message, longueur, TYPE_MESS, 0)
  == -1) {
                perror("msgrcv");
                exit(-1);
        }

 printf("Reception type message %d\n", message.type);
        printf("Ordre recu : %s\n", message.texte);
}

exit(0);
}
```



# Objet IPC : file de messages

A l'exécution :

```
% ./bourse client 200
% ./bourse client 1
% ./bourse client 100000
% ./bourse agent
Reception type message 1
Ordre recu : Vendre 200 actions zorglub
% ./bourse agent
Reception type message 1
Ordre recu : Vendre 1 actions zorglub
% ./bourse agent
Reception type message 1
Ordre recu : Vendre 100000 actions zorglub
```

# Objet IPC : mémoire partagée

- Un segment de mémoire partagée est une zone mémoire contigüe que pourra utiliser un ou plusieurs processus pour y manipuler des données.
- Un processus crée un segment de mémoire partagée et/ou demande à utiliser un segment déjà créé (*shmget()*).
- Ce processus ou un autre demande au SE d'initialiser un pointeur sur cette zone (*shmat()* ).



# Objet IPC : mémoire partagée

- Ce pointeur est initialisé à une adresse de mémoire virtuelle différente pour chaque processus qui sera traduite en adresse physique identique pour tous les processus utilisant cette zone mémoire partagée.
- L'accès à des variables partagées mappées dans la zone mémoire partagée nécessite d'utiliser des sémaphores pour en gérer l'intégrité lors d'accès concurrents par les processus.
- Il existe des droits d'accès de type *NIX sur un segment de mémoire partagée.

# Objet IPC : mémoire partagée

- Création du segment :

```
# include <sys/shm.h>
int shmget(key_t key, int size, int flg)
   retour = shm_id
```

- Attachement du segment :

```
int shmat(int shmid, const void *shmadr, int flg);
   adr = 0 : choix de l'adresse par le système
   d'exploitation
```



# Objet IPC : mémoire partagée

- Détachement du segment :

```
int shmdt(const void *shmadr)
   retour = 0 ou -1
   Libération de l'espace mémoire uniquement lors du dernier
   détachement
```

- Opérations de contrôle :

```
 int shmctl(int shm_id,int cmde, struct shmid_ds *structure)
   cmde = IPC_RMID, IPC_STAT, IPC_SET, SHM_(UN)LOCK
   (verrouillage en mémoire = pas de swap autorisé)
```

# Objet IPC : mémoire partagée

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>

main(){
int pid, shmid;
char *mem, *mem1;
key_t key = 0x99990000;

if((shmid=shmget(key, 128, IPC_CREAT|IPC_EXCL|SHM_R|SHM_W)) < 0) {
        perror("shmget");
        exit(-1);
}
printf("Identificateur du segment=%d\n", shmid);
```



# Objet IPC : mémoire partagée

```
pid =fork();

if(pid == 0){           // processus fils
        if((mem=shmat(shmid, 0, 0)) == (void *) -1) {
                perror("shmat fils");
                exit(-1);
        }
        strcpy(mem,"Message pour le pere");
        shmdt(mem);
        exit(0);
}
```

# Objet IPC : mémoire partagée

```
else {                          //processus père
        if((mem=shmat(shmid, 0, 0)) == (void *) -1) {
                perror("shmat pere");
                exit(-1);
        }
        wait(0);
        printf("message reçu de fils : %s\n", mem); fflush(stdout);
        shmdt(mem);
}
shmctl(shmid, IPC_RMID, NULL);
exit(0);
}
```

A l'exécution :

```
% ./shm
Identificateur du segment=5101
message reçu de fils : Message pour le pere
```



# Objet IPC : sémaphore

- Il est possible avec l'IPC sémaphore de créer des sémaphores à compteur (donc des sémaphores binaires).
- On utilisera les sémaphores pour gérer correctement l'accès à un segment mémoire partagée…

# Objet IPC : sémaphore

- Création d'un sémaphore :

```
int semget(key_t clé,int nsems,int semflg) :
   retour = semid
```

- Opérations sur le sémaphore :

```
int semop (int semid, struct sembuf *spos, int nsops)


struct sembuf {
   u_short sem_num;
   short sem_op;
   short sem_flg;
}
```



# Objet IPC : sémaphore

- Opérations de contrôle du sémaphore :

```
int semctl(int semid,int semnum,int cmd, union semun arg)
    cmde = IPC_RMID, IPC_STAT, IPC_SET,GETVAL…
```

- Sémaphore binaire :
  - `P(sem) : semop(sem_num, -1, 0);`
  - `V(sem) : semop(sem_num, 1, 0);`

# Objet IPC : sémaphore

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

extern int errno;

main() {
key_t cle=0x99990000;
int semid;
struct semid_ds buf;

printf("Creation sem : ");
```



# Objet IPC : sémaphore

```
if((semid=semget(cle, 10, IPC_CREAT|0660)) == -1) {
        perror("BAD.\n");
        exit(-1);
}
printf("OK. semid=%d\n", semid);

exit(0);
}
```

A l'exécution :

```
% ./sem
Creation sem : OK. semid=32769
```

# CONCLUSION



# Conclusion

- Nous avons pu voir les éléments constitutifs d'un SE en partant du matériel jusqu'aux mécanismes logiciels de base mis en œuvre.
- Ce n'est bien sûr qu'une première étape pour approfondir le fonctionnement d'un SE…
- Nous avons pu étudier les principaux mécanismes de communication interprocessus sous *NIX, ce qui vous permettra de concevoir des logiciels plus élaborés qu'un simple programme autonome.
- Nous verrons la mise en œuvre des IPC sous *NIX en TP…

# BIBLIOGRAPHIE



# Bibliographie


- Architecture des machines et systèmes informatiques. A. Cazas et J. Delacroix. Editions Dunod
- UNIX. Programmation et communication. J.M. Rifflet. Editions Mc Graw Hill
- Programmation système en C sous Linux : Signaux, processus, threads, IPC et sockets. C. Blaess. Editions Eyrolles