Cours programmation système & réseaux

David SALLÉ (d4v1d.s4ll3@gmail.com)

Version du document : v0.3
Date : 29/09/2017

Table des matières

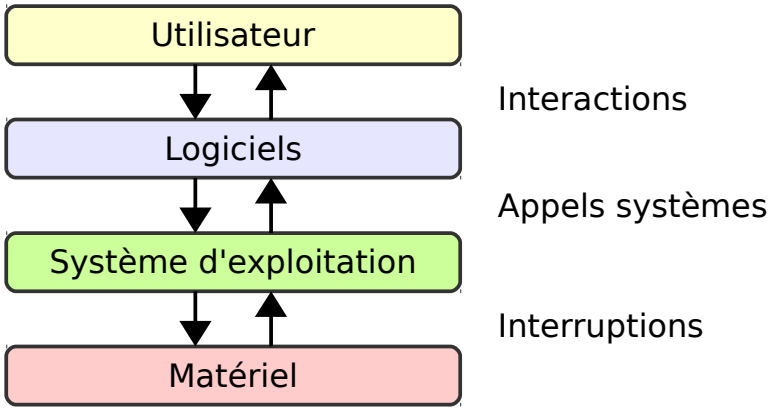
1 – Introduction....4
1.1 – Définitions....4
1.2 – Un peu d'histoire....5
1.3 – Rôles....6
1.4 – Composants....7
1.5 – Familles....7
1.6 – Types de noyau....8
1.7 – Marché....8
1.8 – Multi-tâches....9
2 – Processus lourd/léger....10
2.1 – Qu'est ce qu'un processus....10
2.2 – Etats....11
2.3 – Différence processus lourd/léger....12
2.4 – Programmation multi-tâches....13
2.4.1 – Processus lourd (fork)....13
2.4.2 – Processus léger (pthread_create API C/POSIX)....14
2.4.3 – Processus léger (thread API C++11)....14
3 – Ordonnancement....15
3.1 – Ordonnanceur....15
3.2 – Commutation de contexte....15
3.3 – Coopératif VS préemptif....15
3.4 – Algorithme d'ordonnancement....16
3.5 – Priorité....16
3.6 – Exemple en C++....17
3.7 – Les interruptions....17
3.8 – Les entrées/sorties....19
4 – Communication inter-processus....20
4.1 – Introduction....20
4.2 – Tube....20
4.3 – Files de messages....20
4.4 – Mémoire partagée....21
4.5 – Et pour les threads....22
4.6 – Autres techniques....22
5 – Accès concurrents....23
5.1 – Définitions....23
5.2 – Sémaphore et mutex....24
5.3 – Autres techniques de synchronisation....27
6 – Règles d'échange de données....28
6.1 – Introduction et problématique....28
6.2 – Modèle producteur/consommateur....29
6.2.1 – Solution à 1 case (1 producteur et 1 consommateur)....29
6.2.2 – Solution à N case (1 producteur et 1 consommateur)....29
6.2.3 – Solution à N case (X producteurs et Y consommateurs)....30
6.3 – Modèle lecteur/rédacteur....31
7 – Spécificités temps-réel....32
7.1 – Définition....32
7.2 – Domaine d'utilisation....32
7.3 - Caractéristiques....33

7.4 – Exemples....33
8 – Spécificités embarqué....34
8.1 – Définition....34
8.2 – Enjeux économiques....34
8.3 – Firmware....35
8.4 – Contraintes d'IHM....35
9 – Programmation réseau....36
9.1 – Les sockets....36
9.1.1 - Généralités....36
9.1.2 – Modes de communication....36
9.2 – Rappels sur les réseaux....36
9.2.1 – Sockets VS modèle OSI....36
9.2.2 – Encapsulation....37
9.2.3 – Modèle client/serveur....37
9.3 – Les sockets POSIX....38
9.3.1 – Panorama des fonctions....38
9.3.2 – Client TCP....39
9.3.3 – Serveur TCP....40
9.3.4 – Client UDP....41
9.3.5 – Serveur UDP....41
9.3.6 – Gestion des erreurs....42
9.3.7 – Informations sur le correspondant....42
10 – Ressources....43
10.1 – Bibliographie....43
10.2 – Webographie....43

1 – Introduction

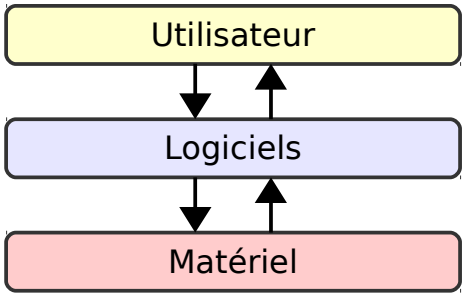
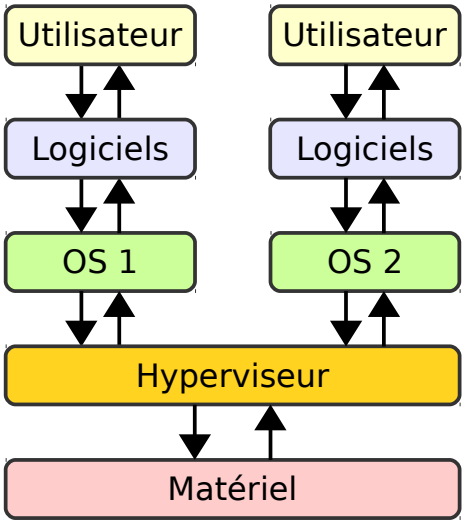
1.1 – Définitions

Le **système d'exploitation** (noté SE ou OS, abréviation du terme anglais Operating System), est chargé d'assurer la liaison entre les ressources matérielles, l'utilisateur et les logiciels (traitement de texte, jeu vidéo, ...).



Le système d'exploitation en dissociant les logiciels et le matériel, mais aussi en proposant une interface conviviale (IHM) permet à l'utilisateur au accès relativement facile à une machine très complexe.

D'autres situations sont possibles :

« Bare metal »	Virtualisation
 Ici pas de système d'exploitation, c'est le logiciel qui se charge de tout. On trouve ce cas de figure dans avec les microcontrôleurs et les périphériques trop simple pour nécessiter un OS.	 Ici une couche supplémentaire, l'hyperviseur, est ajoutée pour virtualiser le matériel et ainsi permettre de faire fonctionner plusieurs OS en même temps sur une même machine physique

1.2 – Un peu d'histoire

Quelques rappels historiques

Dates	Descriptions
Années 1950	La **préhistoire**. Accès direct à la machine pour le technicien qui souhaite exécuter son programme (pile de cartes perforées). Ce dernier doit réserver l'ordinateur pour son usage pendant un certain temps. Le résultat est alors imprimé sur du papier. *Un programme dans l'ordinateur (le moniteur) permet de lancer, arrêter les programme : l'ancêtre des systèmes d'exploitation*
Années 1960	Pour optimiser l'utilisation des ordinateurs très chers à cette époque, le **traitement par lot** apparaît. Un technicien spécialisé est chargé de récupérer les programmes des utilisateurs, de les exécuter sur la machines les uns après les autres et de fournir les résultats imprimés après coup aux usagers. *Le programme dans l'ordinateur (le moniteur) se perfectionne alors pour gérer ce travail en série (temps max, tampons E/S, protection du moniteur)*
Années 1970	C'est l'époque du **temps partagé** où plusieurs utilisateurs vont pouvoir utiliser l'ordinateur en même temps en se voyant allouer des quantums de temps assez courts mais très souvent. De nouvelles innovations matérielles vont permettre de mieux gérer la mémoire (pagination/segmentation) et aussi de stocker les programmes sur des disques. *Le programme dans l'ordinateur (système d'exploitation) doit à présent gérer la mémoire et le disque pour pouvoir y stocker les données de chaque utilisateur. Il doit en outre faire en sorte que chacun puisse exécuter son programme à tour de rôle sur un même processeur. Il permet également à chaque utilisateur via un terminal d'intéragir avec lui. (ex : UNIX)*
Années 1980	L'ordinateur se démocratise et entre chez le particulier. Celui-ci a besoin d'un interface simple et conviviale pour pouvoir utiliser la machine. L'**IHM graphique** avec des icônes, des fenêtres, des boutons voit le jour. Celle-ci peut se gérer au travers d'un clavier/souris et d'un écran. *Le programme dans l'ordinateur (système d'exploitation) doit alors gérer en plus cet environnement graphique et ses nouveaux périphériques. (ex : Apple II, Macintosh)*
Années 1990	La puissance des machines continue de suivre la célèbre loi de Moore. Le **web** fait son apparition avec un nouveau logiciel : le navigateur. *Le systèmes d'exploitation évoluent sans grosse révolution, mais deviennent plus stable et plus polyvalent. Microsoft affine ses versions de Windows tandis que Linux, un clone d'UNIX libre fait son apparition.*
Années	Les **appareils mobiles** (tabletPC, PDA, smartphone, tablette tactile)

Dates	Descriptions
2000	commencent à se démocratiser. Sur les ordinateurs « classiques » les processeurs multi-coeurs permettent de gagner en puissance en contournant la limite physique de soudure des transistors. *De nouveaux systèmes d'exploitation voient le jour. Ils sont adaptés à ces nouveaux périphériques mobiles et ses nouveaux composants :processeur, mémoire, connexion sans fil, écran tactile, ... (ex : Android, iOs, Windows xxx)*
Années 2010	L'informatique **ubiquitaire** ou **pervasive** envahit notre quotidien avec ses objets connectés (maison, transport, santé, ...) *Les systèmes d'exploitation s'adaptent aux contraintes de ces nouveaux appareils*

1.3 – Rôles

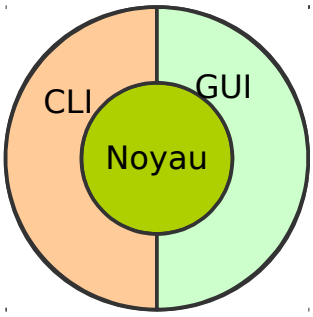
Selon son environnement le système d'exploitation peut avoir en charge :

- Gestion du **processeur** : le système d'exploitation est chargé de gérer l'allocation du processeur entre les différents programmes grâce à un ordonnanceur.
- Gestion de la **mémoire vive** : le système d'exploitation est chargé de gérer l'espace mémoire alloué à chaque application et à lui même. Grâce au MMU (Memory Management Unit) ces zones sont cloisonnées pour éviter qu'un programme écrive dans la zone d'un autre. En cas d'insuffisance de mémoire physique, le système d'exploitation peut créer une zone mémoire sur le disque dur.
- Gestion des **entrées/sorties** : le système d'exploitation permet d'unifier et de contrôler l'accès des programmes aux ressources matérielles par l'intermédiaire des pilotes
- Gestion de l'exécution des **applications** : le système d'exploitation est chargé de la bonne exécution des applications en leur affectant les ressources nécessaires à leur bon fonctionnement. Il permet à ce titre de «tuer» une application ne répondant plus correctement.
- Gestion des **droits** : le système d'exploitation est chargé de la sécurité liée à l'exécution des programmes en garantissant que les ressources ne sont utilisées que par les programmes et utilisateurs possédant les droits adéquats (lecture, écriture, exécution, ...).
- Gestion des **fichiers** : le système d'exploitation gère la lecture et l'écriture dans le système de fichiers et les droits d'accès aux fichiers par les utilisateurs et les applications.

1.4 – Composants

Les différents composants d'un système d'exploitation :

- Le **noyau** (en anglais kernel) représentant les fonctions fondamentales du système d'exploitation telles que la gestion de la mémoire, des processus, des fichiers, des entrées-sorties principales, et des fonctionnalités de communication.
- Le **shell** (en anglais shell, traduisez «coquille» par opposition au noyau) permettant la communication avec le système d'exploitation par l'intermédiaire de :
  - **CLI** (Command Line Interface) : interpréteur de commandes permettant à l'utilisateur via des commandes et un langage de script de dialoguer avec l'OS. Très souvent utilisé sur les serveurs pour sa puissance et ses faibles ressources consommées
  - **GUI** (Graphical User Interface) : interface graphique qui avec son pointeur, ses fenêtres, ses icônes, ses boutons permet à l'utilisateur une manipulation aisée et conviviale de la machine



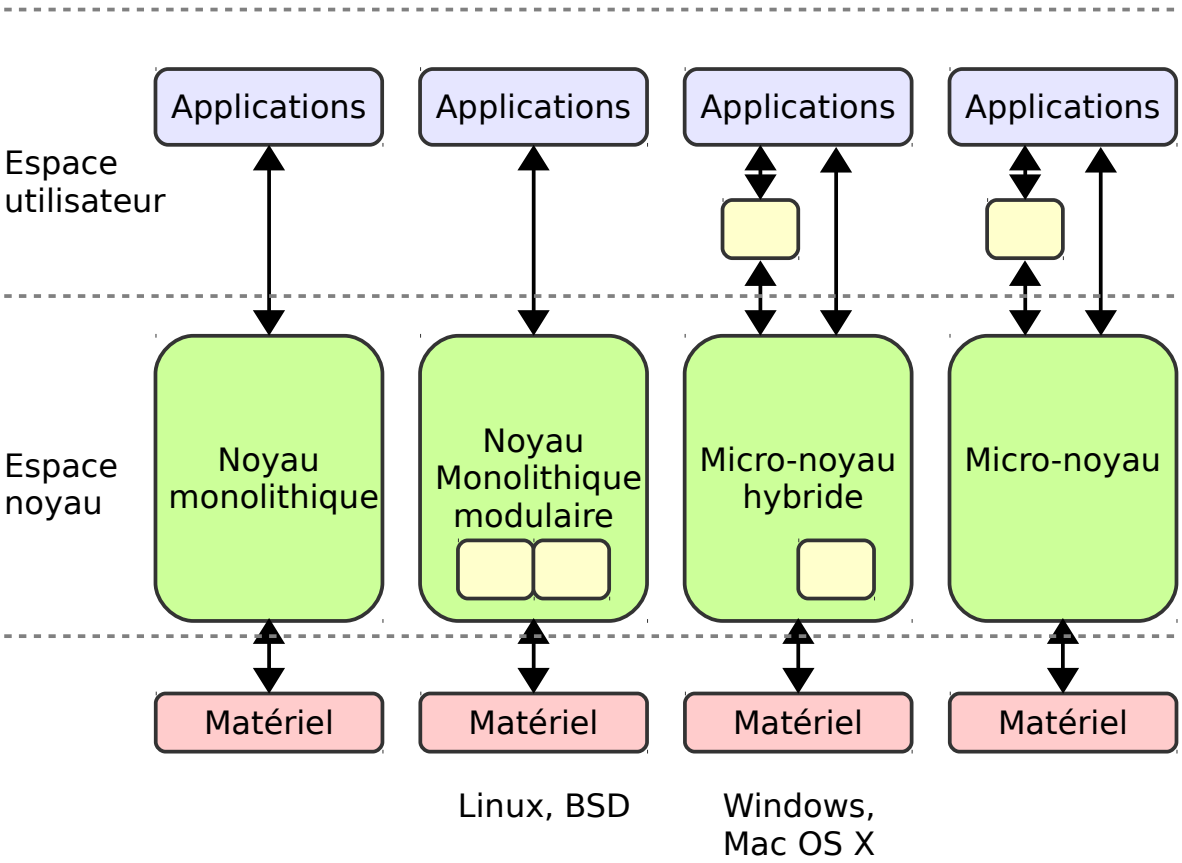
1.5 – Familles

Les différentes familles de système d'exploitation selon le système visé :

- Système à **temps partagé** : c'est le cas des OS « classiques » (Windows, Linux, OS X) où chaque processus pourra à tour de rôle utiliser le processeur et les autres ressources matérielles
- Système **embarqué** : ce sont des systèmes d'exploitation prévus pour fonctionner sur des machines de petite taille, (PDA, sondes spatiales, robot, ordinateur de bord de véhicule, etc.). Ainsi, une caractéristique essentielle des systèmes embarqués est leur gestion avancée de l'énergie et leur capacité à fonctionner avec des ressources limitées.
- Système **temps-réel** : essentiellement utilisés dans l'industrie, sont des systèmes dont l'objectif est de garantir que les opérations seront effectuées en respectant des délais stricts quelles que soient les conditions d'utilisation (charge du système, ...). Ces systèmes sont aussi très souvent embarqués

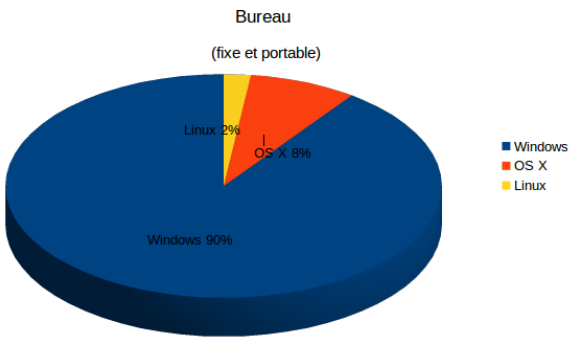
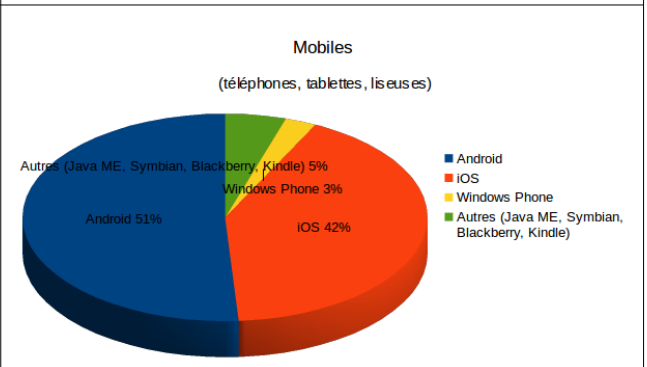
1.6 – Types de noyau

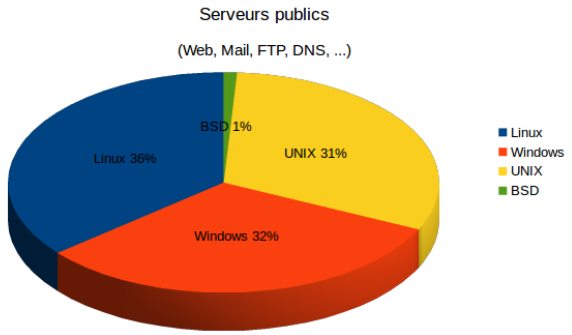
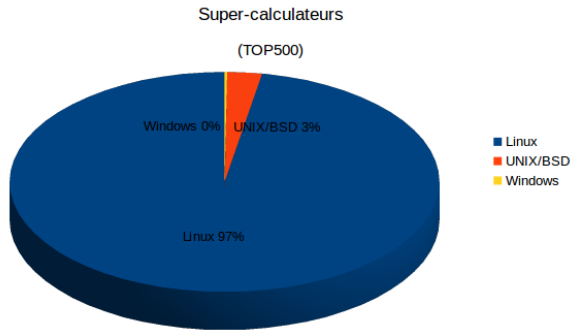
Différents types de noyaux pour les systèmes d'exploitation « classiques » et temps-réel.



1.7 – Marché

Quelques éléments statistiques de l'utilisation des systèmes d'exploitation selon l'usage

PC de bureau	Appareils mobiles
	

Serveurs (publics)	Super-calculateurs
	

1.8 – Multi-tâches

Lorsque l'on aborde la programmation multi-tâches, on a souvent à gérer plusieurs types de problèmes :

- **création/arrêt** des tâches en parallèle (chapitre 2)
- **ordonnancement** des tâches (chapitre 3)
- **communication** de données entre tâches (chapitre 4)
- **synchronisation** des travaux entre tâches (chapitre 5)
- **gestion** des ressources partagées (chapitre 5)

Nous verrons quelques « recettes » prêtes à l'emploi pour les 2 dernières problématiques (chapitre 6) ainsi que les spécificités des systèmes embarqués (chapitre 7) et temps réel (chapitre 8)

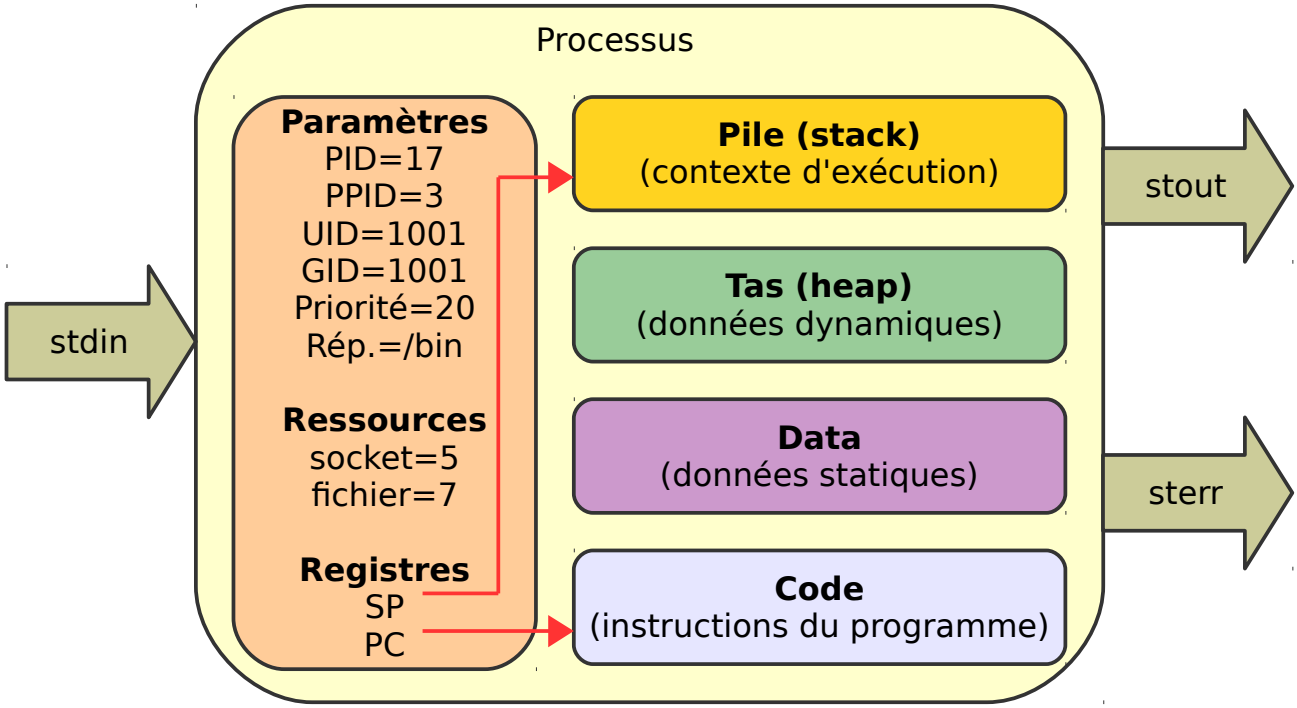
2 – Processus lourd/léger

2.1 – Qu'est ce qu'un processus

Un processus est un programme en cours d'exécution par un ordinateur. Il est constitué :

- d'un ensemble d'**instructions** à exécuter (stockées en RAM)
- d'un espace d'adressage en **mémoire vive** pour travailler
- de **ressources** (fichiers ouverts, socket réseaux, ...)
- d'un **environnement** (PID, répertoire, utilisateur, droits, priorité, temps, processus parent, variables d'environnement, état registres CPU, ...)
- des **flux** d'entrée (stdin) et de sortie (stdout, stderr)

Schéma (simplifié) d'un processus en mémoire vive après avoir été lancé :



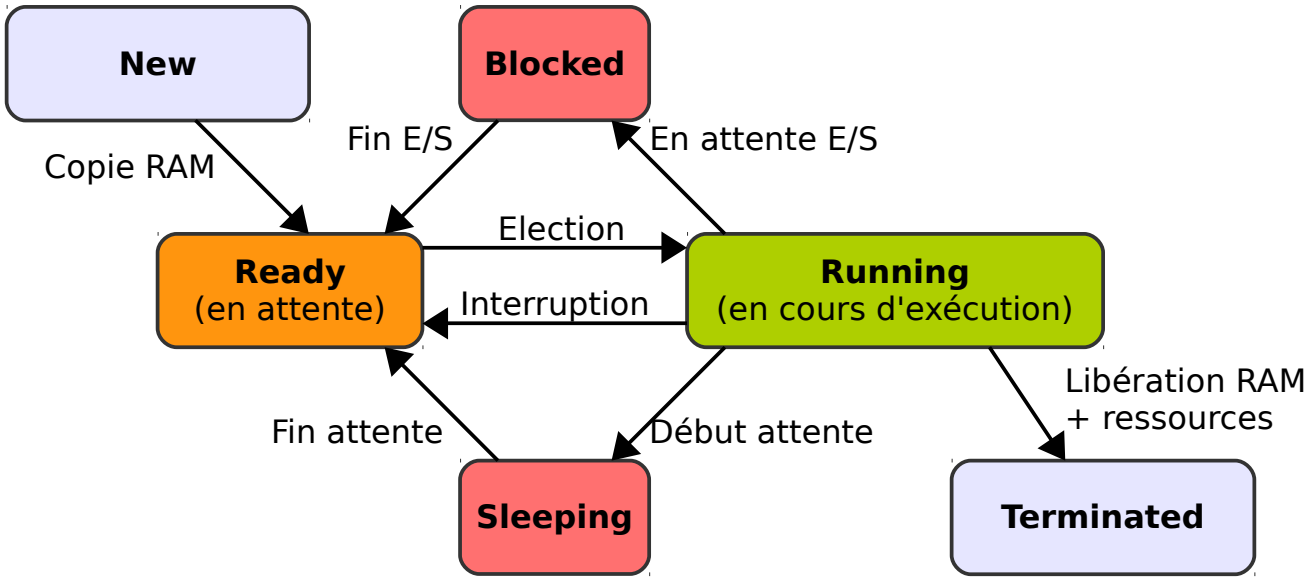
En détail :

- **pile** : variables locales aux fonctions, paramètres, résultats et adresses de retour des sous-programmes
- **tas** : variables allouées dynamiquement par pointeur avec un new par exemple
- **data** : variables globales du programme (celles ayant une valeur autre que 0 ou NULL, puis celles non initialisée ou à 0 : BSS)
- **code** : instructions du programme en langage machine

2.2 – Etats

Un processus peut prendre plusieurs états tout au long de sa « vie » :

- **new** : le processus a juste été lancé
- **ready** : complètement copié en RAM il attend que l'ordonnanceur lui donne la main sur le processeur
- **running** : le processus est en cours d'exécution sur le processeur
- **blocked** : le processus est bloqué par la lecture d'une entrée (cin clavier, read sur une socket ou un fichier)
- **sleeping** : le processus dort (ex : appel de la fonction sleep())
- **terminated** : le processus s'est terminé normalement ou a été stoppé et ses ressources libérées



Il existe d'autres états plus anecdotiques :

- **swapped** : le processus est transféré de la RAM en mémoire virtuelle (donc sur le disque dur)
- **zombie** : cas particulier où un processus fils se termine alors que sont père est déjà terminé. Devenu orphelin, le système le place en état de zombie pour pouvoir récupérer ses ressources (mémoire, fichiers...)

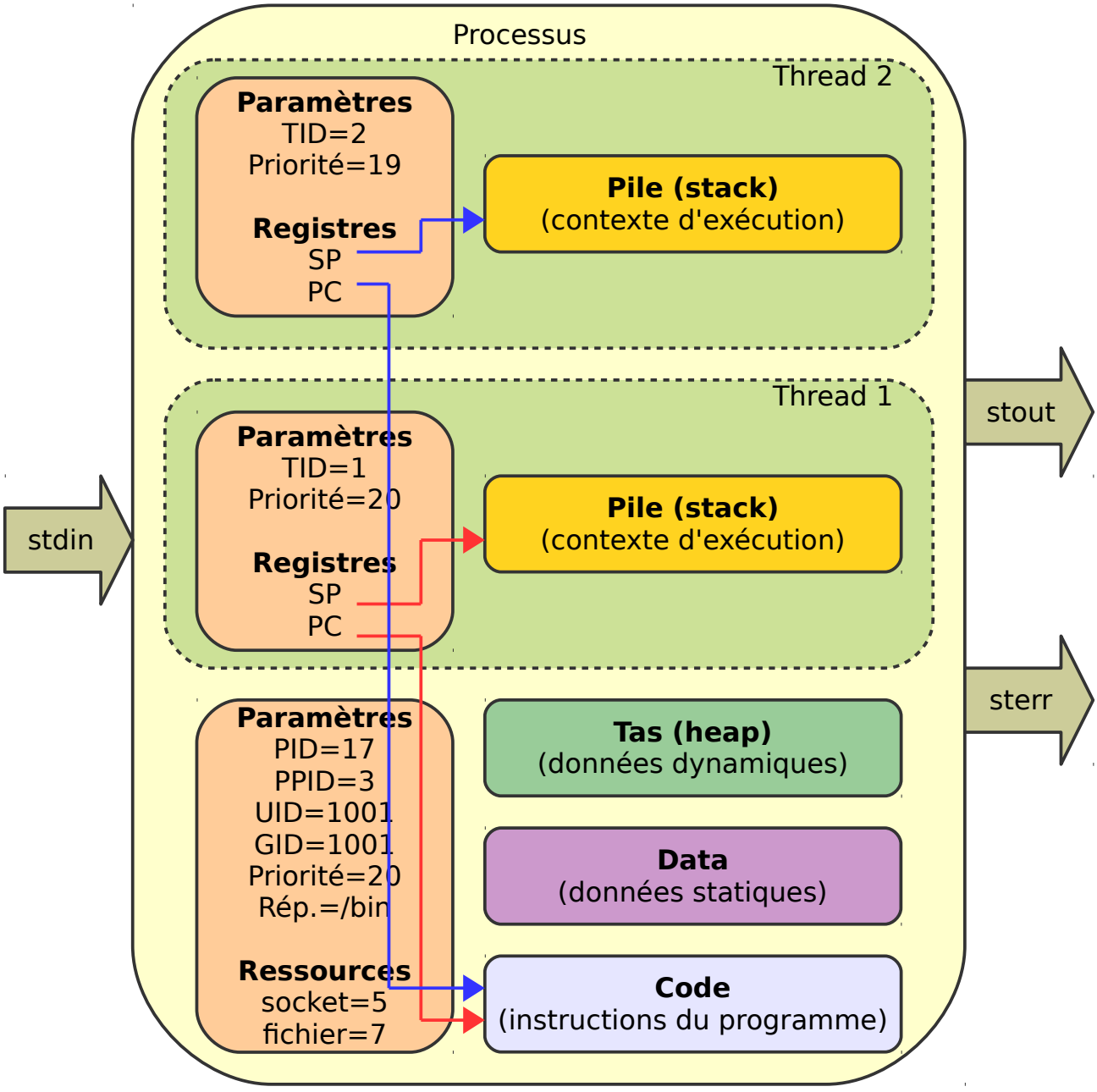
Le passage d'un processus à l'autre sur le processeur est appelé **commutation de contexte** et est présenté au chapitre 3.2

2.3 – Différence processus lourd/léger

Les processus légers aussi appelés **threads** ont été créés pour faciliter le développement d'applications multi-tâches.

Processus lourd	Processus léger
• Plus long a créer • Commutation de contexte plus longue • Communication inter-processus complexe via IPC (voir chapitre 4)	• Plus rapide à créer • Commutation de contexte plus rapide • Communication inter-processus simple car partageant une zone mémoire commune

Schéma d'un processus lourd, contenant 2 processus légers :



2.4 – Programmation multi-tâches

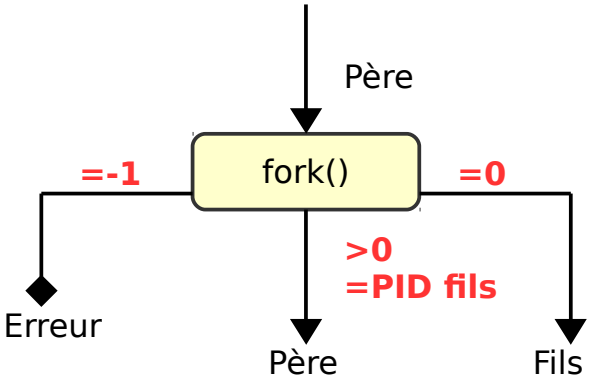
L'objectif des 3 programmes suivants est d'exécuter simultanément 2 tâches :

- une tâche qui affiche « 1 » toutes les secondes (10 fois)
- une tâche qui affiche « 2 » toutes les 2 secondes (5 fois)

2.4.1 – Processus lourd (fork)

La création d'un processus lourd s'effectue avec la fonction **fork()**.

Un processus créateur (le père) par un appel à la fonction fork() créé un nouveau processus (le fils) qui est une copie exacte de lui même (code et data). Les 2 processus s'exécute ensuite de manière concurrente.



Exemple de code source

```c
#include <unistd.h> // pour fork()
#include <stdlib.h> // pour exit()
#include <stdio.h>  // pour printf()

int main()
{
    int pid; // on peut aussi déclarer pid_t pid

    printf( "Je suis le pere\n" );

    switch ( pid=fork() )
    {
        case -1:
            printf("Echec du fork !!!");
            exit(2);

        case 0: // on est dans le processus fils
            printf("Je suis le fils\n");
            exit(0);

        default: // on est dans le processus père
            sleep(1) ;
            printf("Je suis toujours le pere\n");
            break;
    }
    return 0;
}
```

2.4.2 – Processus léger (pthread_create API C/POSIX)

Exemple de code source

```
// COMPILATION : g++ ex2.cpp -o ex2 -lpthread
#include <pthread.h>    // pour pthread_create() et pthread_join()

// Fonction qui fait quelque chose
void *faire_quelque_chose(void *parametre)
{
        // ...
}

// Point d'entrée du programme
int main()
{
        // La thread
        pthread_t ma_thread;

        // Creation de la thread avec une fonction à embarquer
        pthread_create(&ma_thread, NULL, faire_quelque_chose, NULL);

        // Attente de la fin de chaque thread
        pthread_join(ma_thread, NULL);

        // ...
}
```

2.4.3 – Processus léger (thread API C++11)

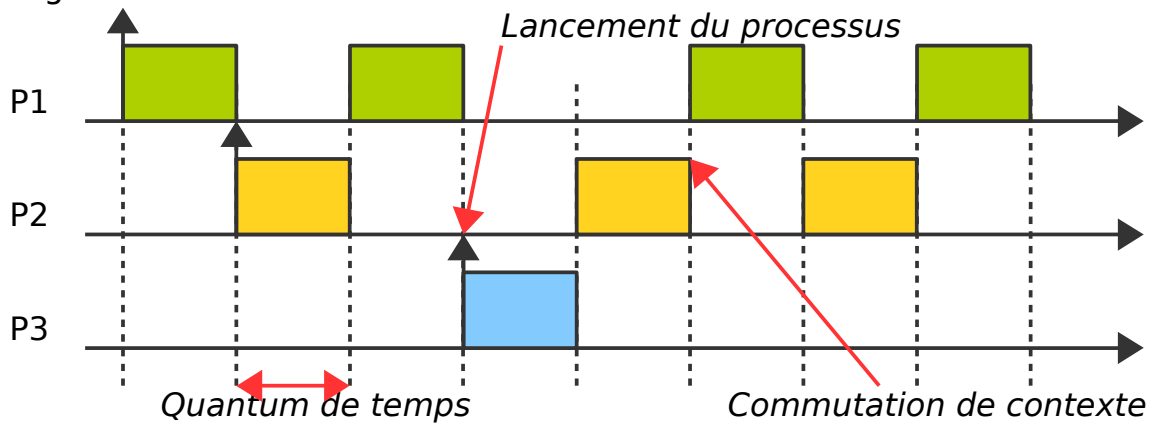
Exemple de code source

```
// COMPILATION : g++ ex3.cpp -o ex3 -pthread -std=c++11
#include <iostream>     // pour cout
#include <thread>       // pour objet thread

// Espace de nom
using namespace std;

// Fonction qui fait quelque chose
void faire_quelque_chose()
{
        // ...
}

// Point d'entrée du programme
int main()
{
        // Creation de la thread avec une fonction à embarquer
        thread ma_thread(faire_quelque_chose);

        // Attente de la fin de la thread
        ma_thread.join();

        // ...
}
```

3 – Ordonnancement

3.1 – Ordonnanceur

Au coeur d'un système d'exploitation se trouve l'**ordonnanceur** (scheduler in english) qui comme un chef d'orchestre gère l'utilisation du processeur par les programmes.



Son rôle est crucial car il y a souvent plus de processus à exécuter que de processeurs. En allouant à chaque processus un **quantum de temps** (ex : 10 ms sur Linux), l'utilisateur à l'illusion que plusieurs programmes fonctionnent simultanément. On parle aussi de **pseudo-parallèlisme**.

3.2 – Commutation de contexte

La commutation de contexte consiste à remplacer un processus P1 travaillant sur le processeur par un autre processus P2.

Pour bien faire il faut :

1. sauvegarder l'état d'un processus P1 quelque part en mémoire (ex : pile du processus)
2. retrouver le contexte de P2 (ex : depuis la pile de P2)
3. restaurer le contexte de P2 sur le processeur
4. reprendre l'exécution de P2 juste après sa dernière instruction

3.3 – Coopératif VS préemptif

L'ordonnanceur peut fonctionner selon 2 modes :

- **collaboratif** : les programmes doivent alors prévoir de passer eux-mêmes la main aux autres
- **préemptif** : l'ordonnanceur peut à tout moment couper un processus dans sont travail et le remplacer par un autre (avec la commutation de contexte)

3.4 – Algorithme d'ordonnancement

L'ordonnanceur utilise un algorithme d'ordonnancement afin de choisir quel processus pourra utiliser le processeur.

Algorithme	Description
FIFO (First In First Out)	Premier arrivé, premier servi
SJF (Shortest Job First)	Le processus le plus court en premier
Round Robin (tourniquet)	Chaque processus aura le processeur pendant un certain quantum de temps.
RMS (Rate Monotonic S.)	Le processus qui a la plus petite période à la plus grande priorité
A priorité fixe	Le processus le plus prioritaire a le processeur

Les systèmes d'exploitation Windows, Mac OS X et Linux utilisent des combinaisons plus ou moins complexes de ces algorithmes en mode **préemptif** (ex : Multi-level-feedback, Completely Fair Scheduler)

3.5 – Priorité

A chaque processus, il est possible d'attribuer une **priorité** de sorte qu'un processus qualifié de plus important soit exécuté en premier ou plus souvent que les autres ou à l'inverse uniquement quand le processeur n'a rien à faire.

3.6 – Exemple en C++

Exemple de code permettant de changer à la fois de politique d'ordonnancement et de priorité pour une thread.

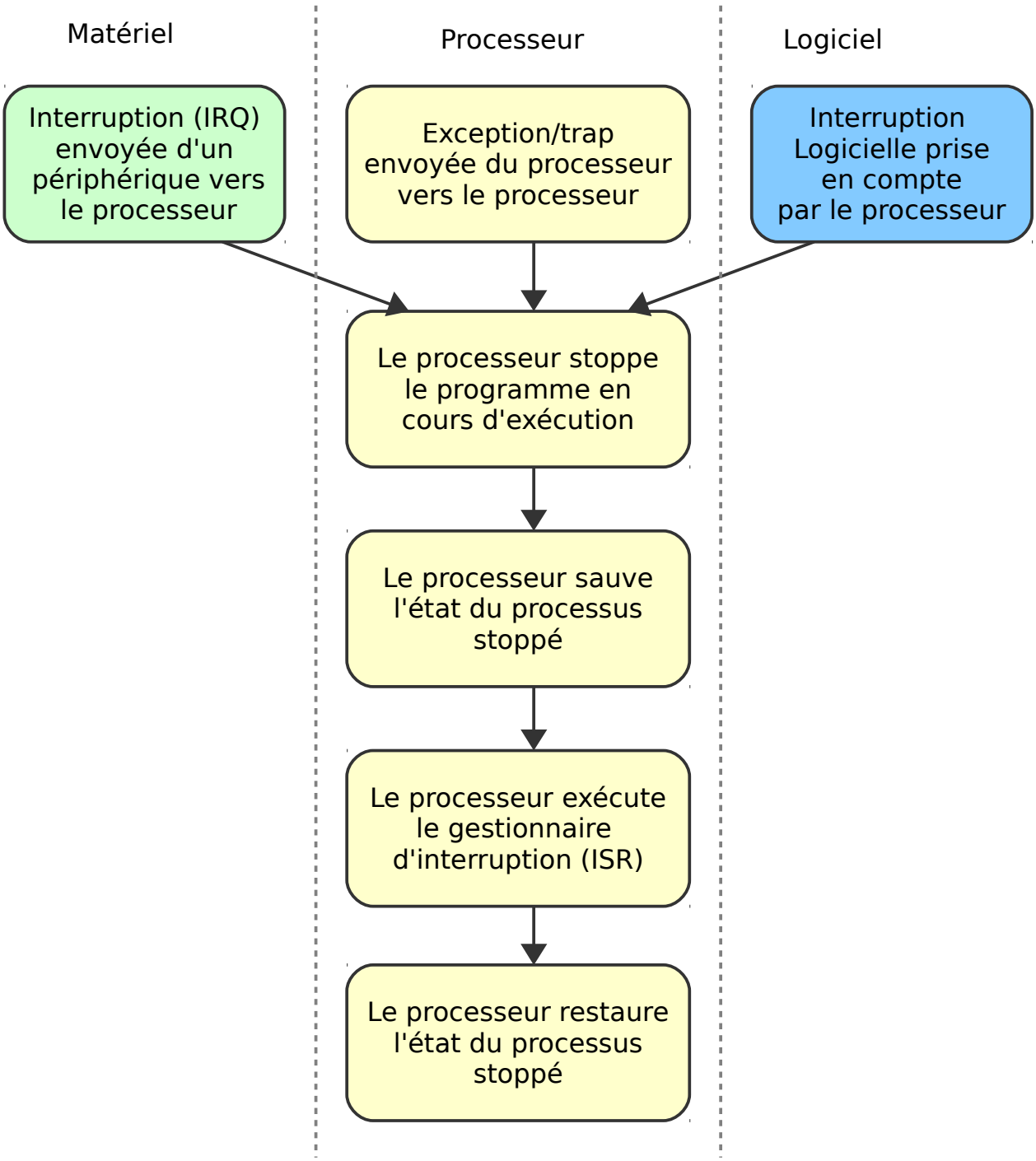
```
// Variables
struct sched_param parametre_ordonnanceur;
pthread_t ma_thread;
pthread_attr_t attributs_thread;


// Initialisation des attributs de la thread
pthread_attr_init(&attributs_thread);

// Configuration des attributs de la thread pour qu'ils puissent être modifiés
pthread_attr_setinheritsched(&attributs_thread, PTHREAD_EXPLICIT_SCHED);

// Changement de politique d'ordonnancement pour la thread
pthread_attr_setschedpolicy(&attributs_thread, SCHED_FIFO);

// Changement de priorité (priorité max pour SCHED_FIFO)
parametre_ordonnanceur.sched_priority = 10;

// Application des nouveaux attributs à la thread
pthread_attr_setschedparam(&attributs_thread, &parametre_ordonnanceur);

// Création de la thread
pthread_create(&ma_thread, &attributs_thread, faire_quelque_chose, NULL);

// Attente de la fin du travail de la thread
pthread_join(ma_thread, NULL);
```

3.7 – Les interruptions

En plus de la gestion des tâches, l'ordonnanceur doit souvent prendre en compte les interruptions qui peuvent être de 2 types

- **logicielle** : un événement extérieur au programme se produit (timer, signal envoyé par l'OS, ...) ou le programme souhaite faire un appel système
- **interne** processeur : gestion d'erreurs (division par zéro, faute mémoire) aussi appelée « exception » ou « trap » en anglais
- **matérielle** : communication avec le matériel via des IRQ pour gérer les entrées/sorties de manière plus efficace que la scrutation active (polling)

Principe de fonctionnement lorsqu'une interruption survient :



Idéalement le code du gestionnaire d'interruption doit être le plus court possible surtout pour une interruption qui survient très souvent.

Certaines interruptions sont masquables (non prises en compte), d'autres pas comme les « watchdog timers » par exemple

Sur les ordinateurs modernes un composant spécifiques, l'APIC gère les interruptions matérielles.

3.8 – Les entrées/sorties

Les **périphériques d'entrées/sorties** permettent à un système informatique d'interagir avec l'extérieur, utilisateur compris.

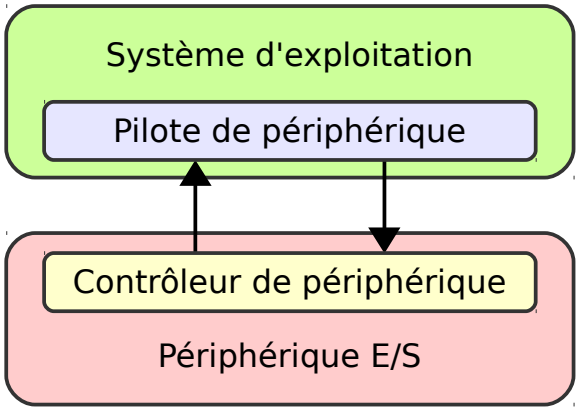
Ces périphériques sont nombreux, variés, avec des caractéristiques et fonctionnements très différents.

Périphérique	E/S	Volume données	Régularité
Souris	Entrée	Faible	Non
Clavier	Entrée	Faible	Non
Ecran	Sortie	Important	Oui
Ecran tactile	Entrée/sortie	Important	Oui/Non
Disque dur	Entrée/sortie	Important	Oui
Port parallèle	Entrée/sortie	Faible	Non
Port RS232	Entrée/sortie	Faible	Non
Carte réseau	Entrée/sortie	Variable	Non

Afin de gérer ces différents périphériques le système d'exploitation dispose de plusieurs possibilités

- Entrées/sorties programmées (l'OS gère tout et attend les réponses : lecture + traitement + écriture)
- Interruptions (voir chapitre précédent)
- Accès direct à la mémoire (DMA pour gros volume de données HDD, CDROM)

Système d'exploitation et périphériques E/S utilisent souvent des intermédiaires afin de faciliter la communication et le travail entre eux.



Sur UNIX tout périphérique est fichier (/dev) et manipulable avec les fonctions : `open(), read(), write() close()`

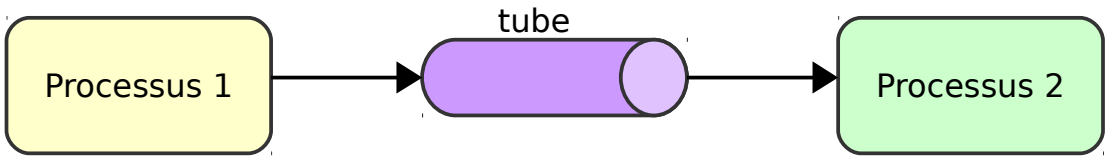
4 – Communication inter-processus

4.1 – Introduction

Comme énoncé au chapitre 2.3, les processus lourds qui ne disposent pas de zone mémoire communes (comme les processus légers), nécessitent l'utilisation d'outils spécifiques (IPC : Inter Processus Communication) pour communiquer entre eux.

4.2 – Tube

Un **tube** (pipe in english) est un mécanisme de communication unidirectionnel permettant le transfert d'un flot continu de caractères. Les caractères sont lus dans l'ordre dans lequel ils ont été écrits dans le tube (FIFO).



Le processus P1 envoie des données à l'entrée du tube afin que le processus 2 à l'autre bout du tube puisse les lire.

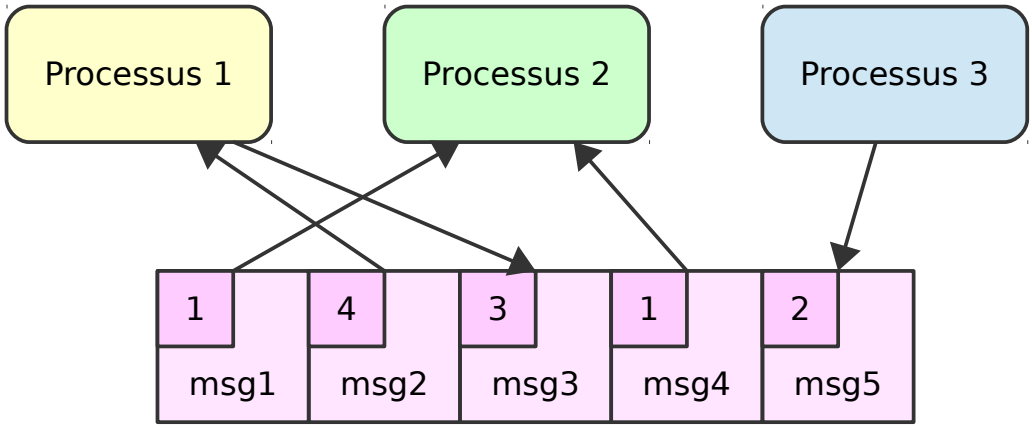
On distingue 2 types de tubes :

- tubes **anonymes** : communication entre 2 processus de la même famille (ex : processus père et fils)
- tubes **nommés** : communication entre 2 processus quelconques

Fonction de l'API : `pipe(), mkfifo(), read(), write(), dup2(), close()`

4.3 – Files de messages

Les **files de messages** (messages queue in english) permettent comme avec des boîtes aux lettres d'échanger des données entre plusieurs processus.



File de messages

Chaque message est composé :

- d'un type (entier strictement positif)
- d'un corps (des données de n'importe quel type et/ou taille)

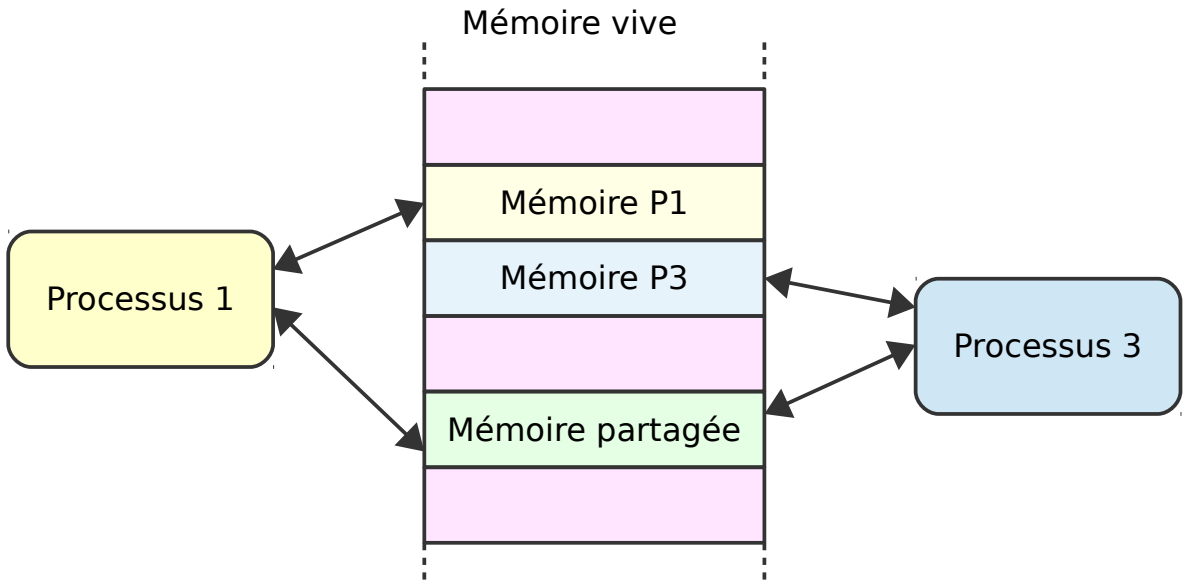
Dans l'exemple précédent, on trouve :

- Le processus 1 lit le message de type 4 : msg2
- Le processus 1 écrit le message de type 3 : msg3
- Le processus 2 lit les messages de type 1 : msg1 et msg4
- Le processus 3 écrit le message de type 2 : msg5

Fonction de l'API POSIX : `mq_open(), mq_send(), mq_receive(), mq_close() et mq_unlink()`

4.4 – Mémoire partagée

La **mémoire partagée**, comme son nom l'indique consiste à définir une zone de la mémoire accessible en lecture/écriture entre plusieurs processus.



Dans cette illustration le processus P1 a bien son espace mémoire dédié, le processus 3 également. Mais ils partagent également une zone mémoire commune où chacun pourra venir lire et écrire des données à l'aide d'un pointeur.

Fonction de l'API POSIX : `shm_open(), mmap(), munmap(), close() et shm_unlink()`

4.5 – Et pour les threads

Une simple variable globale par exemple pourra permettre d'échanger des données entre plusieurs threads. Attention toutefois à la protection et la synchronisation des données (voir chapitre suivant)

4.6 – Autres techniques

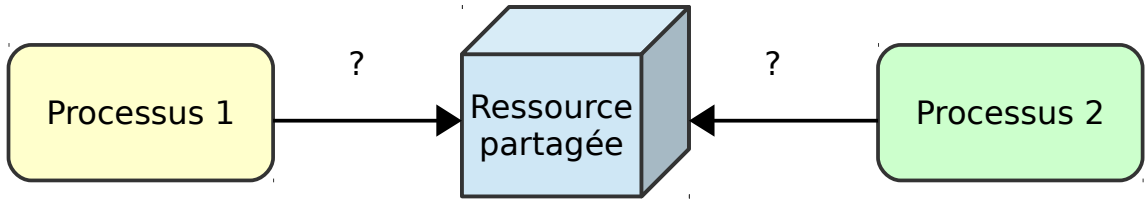
Fichier (avec verrous), base de données, sockets

Techniques	Caractéristiques
Tube	Unidirectionnel, caractères seulement, taille limitée, obligation de synchronisation
File de messages	Bidirectionnel, tout type de données, taille illimitée, pas de problème de synchronisation, possibilité de trier les données
Mémoire partagée	Bidirectionnel, tout type de données, taille illimitée, pas de problème de synchronisation, très rapide, nécessité de protéger les accès concurrents (sémaphore, mutex)
Signal	Bidirectionnel, pas de données
Socket	Bidirectionnel, tout type de données, taille illimitée, problème de synchronisation, à distance via une réseau
Fichier	B-directionnel, tout type de données, taille illimitée, pas de problème de synchronisation, lent, nécessité de protéger les accès concurrents (verrous)
Base de données	Bidirectionnel, tout type de données, taille illimitée, pas de problème de synchronisation, données structurées, à distance via un réseau, installation d'un logiciel/librairie en plus

5 – Accès concurrents

5.1 – Définitions

Une situation problématique ... où 2 processus veulent accéder en même temps à une ressource partagée.



Tout d'abord quelques définitions :

- **ressource partagée** : variable, zone mémoire, matériel partagée par plusieurs processus
- **accès concurrents** : les 2 processus peuvent être en accès concurrents pour accéder à la ressource partagée (ou pas d'ailleurs)
- **exclusion mutuelle** : si la ressource partagée n'est accessible qu'à un seul processus à un instant T, alors on parlera d'exclusion mutuelle entre les 2 processus. (ex : si P1 utilise la ressource partagée, alors P2 est automatiquement exclus)
- **section critique** : portion de code contenant une suite d'instructions ne devant pas être interrompues lors de l'utilisation d'une ressource partagée

Puis quelques problématiques soulevées par cette situation :

- **synchronisation** : P1 a terminé d'utiliser la ressource partagée, comment P2 peut-il en être informé ? Doit-il attendre obligatoirement ?
- **protection** : comment être certain que les opérations effectuées sur la ressource partagée se déroulent correctement ? Complètement ? ...

Le chapitre suivant présente une solution à ces problématiques en informatique.

5.2 – Sémaphore et mutex

Les **sémaphores** sont des objets informatiques utilisés pour synchroniser des processus (ou threads) entre eux. Ils constituent aussi une solution pour résoudre le problème d'exclusion mutuelle, et permettent en particulier de régler les conflits d'accès concurrents de processus distincts à une même ressource partagée.

Un sémaphore est constitué d'un compteur et de 3 opérations :

Opération	Description
Init(S, valeur)	Initialisation du compteur du sémaphore à une certaine valeur

P(S)	Prise du sémaphore qui se traduit par l'algorithme :
	DEBUT
	  Désactivation des interruptions
	  SI (compteur == 0) **ALORS**
	  bloquer le processus
	  placer le processus dans une file d'attente
	  SINON
	  compteur = compteur – 1
	  FINDESI
	  Rétablissement des interruptions
	FIN

V(S)	Vente du sémaphore qui se traduit par l'algorithme :
	DEBUT
	  Désactivation des interruptions
	  compteur = compteur + 1
	  SI la file d'attente n'est pas vide **ALORS**
	  libérer le premier processus ce la file
	  FINDESI
	  Rétablissement des interruptions
	FIN

Ces opérations P(S) et V(S) sont indivisibles ou **atomiques** : elles sont totalement ininterruptibles et ont toujours lieu séquentiellement sur le processeur.

Le **mutex** est un sémaphore particulier qu'on a initialisé à 1. Il fonctionne alors comme une sorte de verrou qu'on peut ouvrir et fermer.

Exemple d'utilisation :

```
Init(S, 1)                      // Initialisation du sémaphore à 1
...
P(S)                            // Prise du sémaphore
lire(ressource_partagee)        // Code de la section critique
ecrire(ressource_partagee)      // Code de la section critique
V(S)                            // Vente du sémaphore
...
```

Fonction de l'API POSIX : `sem_init()`, `sem_post()`, `sem_wait()`, `sem_destroy()`

Exemple de code source avec les sémaphores POSIX :

```c
// Les librairies
#include <pthread.h>    // pour pthread_create() et pthread_join()
#include <semaphore.h>  // pour sem_init(), sem_wait(), sem_post() et
sem_destroy()

// Variables globales
sem_t mon_semaphore;


// Fonction embarquée dans un thread
void *faire_quelque_chose(void *parametre)
{
    // Prise du sémaphore : P()
    sem_wait(&mon_semaphore);

    // ... Travail sur la ressource partagée

    // Vente du sémaphore : V()
    sem_post(&mon_semaphore);
}


// Point d'entrée du programme
int main()
{
    // Identifiants des threads
    pthread_t t1;

    // Initialisation du sémaphore à 1
    sem_init(&mon_semaphore, 0, 1);

    // Creation des 2 threads avec chacune une fonction à embarquer
    pthread_create(&t1, NULL, faire_quelque_chose, NULL);

    // Attente de la fin de chaque thread
    pthread_join(t1, NULL);

    // Destruction du sémaphore
    sem_destroy(&mon_semaphore);

    // Fin du programme
    return 0;
}
```

Exemple de code source avec les mutex POSIX :

```c
// Les librairies
#include <pthread.h>    // pour pthread_create(), pthread_join(),
pthread_mutex_xxx


// Variables globales
pthread_mutex_t mon_mutex;


// Fonction embarquée dans un thread
void *faire_quelque_chose(void *parametre)
{
    // Verrouillage du mutex : P()
    pthread_mutex_lock(&mon_mutex);

    // ... Travail sur la ressource partagée

    // Déverrouillage du mutex : V()
    pthread_mutex_unlock(&mon_mutex);
}


// Point d'entrée du programme
int main()
{
    // Identifiants des threads
    pthread_t t1;

    // Initialisation du mutex
    pthread_mutex_init(&mon_mutex, NULL);

    // Creation des 2 threads avec chacune une fonction à embarquer
    pthread_create(&t1, NULL, faire_quelque_chose, NULL);

    // Attente de la fin de chaque thread
    pthread_join(t1, NULL);

    // Destruction du mutex
    pthread_mutex_destroy(&mon_mutex);

    // Fin du programme
    return 0;
}
```

5.3 – Autres techniques de synchronisation

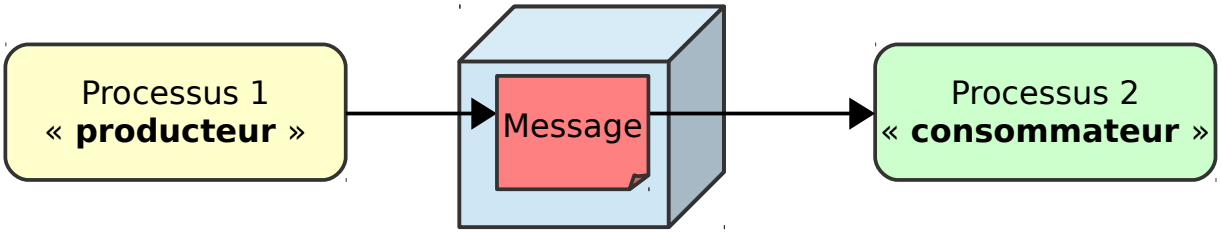
condition, atomic (C++11), synchronized (JAVA)

6 – Règles d'échange de données

6.1 – Introduction et problématique

La problématique majeure des applications multi-tâches tourne très souvent autour du partage d'information (ici appelé **message**) entre

- un/des processus « **producteur** » de données
- un/des processus « **consommateur** » de données



Tampon d'échange

Ci-dessous un exemple de problématique qui peut être étendu à de nombreuses situations

Producteur	Consommateur
• Produire un message *(ex : lecture capteur, résultat calcul, réception réseau, ...)* • Ecrire le message dans le tampon	• Lire le message dans le tampon • Consommer le message *(ex : affichage IHM, écriture fichier/BDD, actionneur, ...)*

Les problèmes de synchronisation qui se posent alors :

- le producteur veut déposer un nouveau message alors que le consommateur est en train de lire l'ancien
- le consommateur veut lire un message alors que le producteur est en train d'en écrire un nouveau
- le producteur dépose un message alors que le consommateur n'a pas eu le temps de lire le précédent
- le consommateur retire un message alors que le producteur n'a rien déposé

Sachant que pour corser les choses, il peut y avoir *plusieurs* producteurs et *plusieurs* consommateurs qui écrivent et lisent *plusieurs* messages ...

Il existe alors des modèles/recettes qui permettent de régler ces problèmes à l'aide de sémaphores et de mutex :

- modèle **producteur/consommateur**
- modèle **lecteur/rédacteur**

6.2 – Modèle producteur/consommateur

6.2.1 – Solution à 1 case (1 producteur et 1 consommateur)

Cette solution est la plus couramment utilisée. Elle correspond par exemple à la lecture d'une information sur un capteur par un processus qui est ensuite exploitée par un autre.

Les éléments en présence :

- 1 processus producteur
- 1 processus consommateur
- une zone tampon en mémoire avec 1 case pour y stocker les données

Les éléments de la solution :

- 1 sémaphore nommé SEMVIDE et initialisé à 1
- 1 sémaphore nommé SEMPLEIN et initialisé à 0

Et l'organisation chez les 2 processus

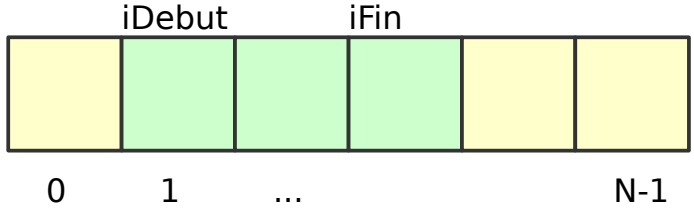
Producteur	Consommateur
Produire un message *(ex : lecture capteur, résultat calcul, réception réseau, ...)* **P(SEMVIDE)** Ecrire le message dans le tampon **V(SEMPLEIN)**	**P(SEMPLEIN)** Lire le message dans le tampon **V(SEMVIDE)** Consommer le message *(ex : affichage IHM, écriture fichier/BDD, actionneur, ...)*

6.2.2 – Solution à N case (1 producteur et 1 consommateur)

Cette solution est très similaire à la précédente mais offre plus de souplesse de fonctionnement grâce à son tampon à N cases.

Les éléments en présence :

- 1 processus producteur
- 1 processus consommateur
- une zone tampon en mémoire avec N cases pour y stocker les données



Les éléments de la solution :

- 1 sémaphore nommé SEMVIDE et initialisé à N
- 1 sémaphore nommé SEMPLEIN et initialisé à 0

Et l'organisation chez les 2 processus

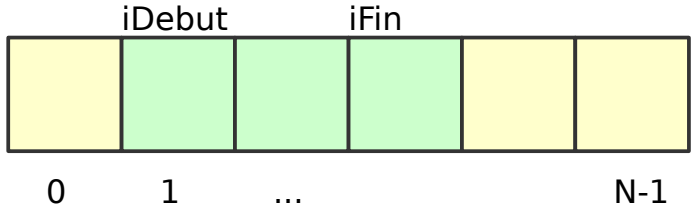
Producteur	Consommateur
Produire un message *(ex : lecture capteur, résultat calcul, réception réseau, ...)* **P(SEMVIDE)** tampon[iDebut] = message iDebut = (iDebut + 1) mod N **V(SEMPLEIN)**	**P(SEMPLEIN)** message = tampon[iFin] iFin = (iFin + 1) mod N **V(SEMVIDE)** Consommer le message *(ex : affichage IHM, écriture fichier/BDD, actionneur, ...)*

6.2.3 – Solution à N case (X producteurs et Y consommateurs)

Cette solution similaire aux précédentes est plus rare.

Les éléments en présence :

- X processus producteur
- Y processus consommateur
- une zone tampon en mémoire avec N cases pour y stocker les données



Les éléments de la solution :

- 1 sémaphore nommé SEMVIDE et initialisé à N
- 1 sémaphore nommé SEMPLEIN et initialisé à 0
- 1 mutex PRODUCTEX
- 1 mutex CONSOMMATEX

Et l'organisation chez les 2 processus

Producteur	Consommateur
Produire un message *(ex : lecture capteur, résultat calcul, réception réseau, ...)* **P(SEMVIDE)** **P(PRODUCTEX)** tampon[iDebut] = message iDebut = (iDebut + 1) mod N **V(PRODUCTEX)** **V(SEMPLEIN)**	**P(SEMPLEIN)** **P(CONSOMMATEX)** message = tampon[iFin] iFin = (iFin + 1) mod N **V(CONSOMMATEX)** **V(SEMVIDE)** Consommer le message *(ex : affichage IHM, écriture fichier/BDD, actionneur, ...)*

6.3 – Modèle lecteur/rédacteur

Cette dernière solution est souvent utilisée dans les logiciels de base de données puisqu'elle permet :

- une lecture ne peut pas avoir lieu en même temps qu'une écriture dans le fichier
- plusieurs lectures peuvent avoir lieu en même temps
- une seule écriture à un instant donné

Les éléments en présence :

- X processus lecteur
- Y processus rédacteur
- 1 fichier où stocker les données

Les éléments de la solution :

- 1 sémaphore nommé LECTEX et initialisé à 1
- 1 sémaphore nommé REDACTEX et initialisé à 1
- 1 compteur initialisé à 0

Et l'organisation chez les 2 processus

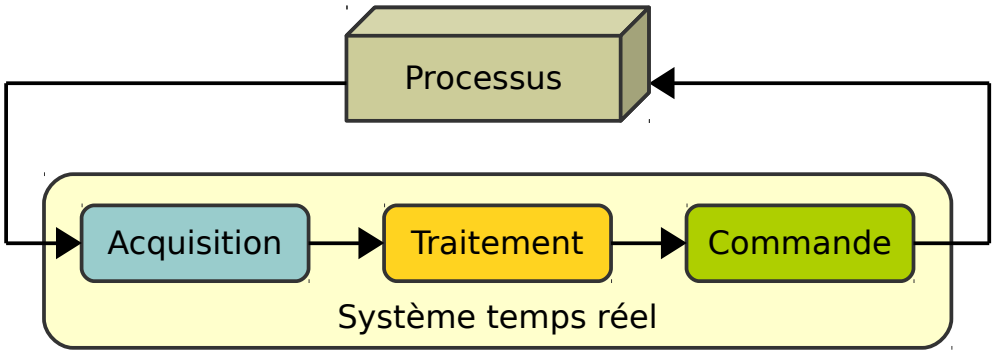
Rédacteur	Lecteur

P(REDACTEX)
Ecrire dans le fichier
V(REDACTEX)
	P(LECTEX)
	compteur = compteur + 1
	SI (compteur = 1) **ALORS**
	  P(REDACTEX)
	FINDESI
	V(LECTEX)
	Lire dans le fichier
	P(LECTEX)
	compteur = compteur - 1
	SI (compteur = 0) **ALORS**
	  V(REDACTEX)
	FINDESI
	V(LECTEX)

7 – Spécificités temps-réel

7.1 – Définition

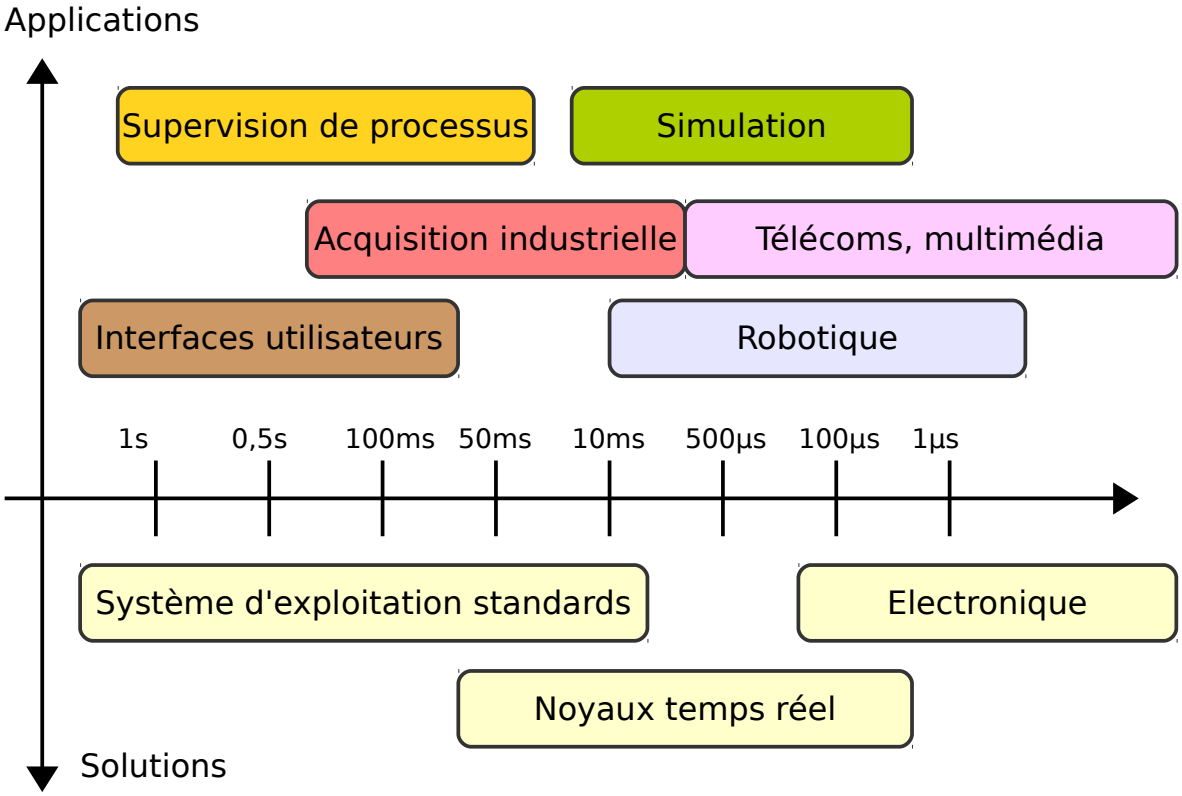
Un système d'exploitation temps réel (**RTOS** in english : Real Time Operating System) est un système d'exploitation spécifiquement conçu pour répondre à des contraintes temporelles fortes.



Le temps « *acquisition* + *traitement* + *commande* » doit obligatoirement être inférieur à une certaine valeur et ce quelque soit le contexte. La justesse du résultat est aussi importante que le délai pour l'obtenir.

En pratique on peut tolérer certains retards (**temps réel mou**) pour certains systèmes (ex : multimédia), et pas pour d'autres (**temps réel dur**) comme les centrales nucléaires, les avions, les missiles, ...

7.2 – Domaine d'utilisation



7.3 - Caractéristiques

Caractéristiques des systèmes d'exploitation temps-réel par rapport aux systèmes d'exploitation « classiques » :

Item	OS temps réel	OS classique
Temps de réponse	Garanti	Non garanti
Architectures supportées	Nombreuses	Peu (sauf Linux, BSD)
Algorithme d'ordonnancement	Grand choix.	Peu ou pas de choix
Gestion priorité	Fine et exacte	Indicative
Temps interruption	Faible et garanti	Grand et non garanti
Temps commutation de contexte	Faible et garanti	Grand et non garanti
Choix coopératif/préemptif	Oui	Non
Nombre de tâches à gérer	Peu	Beaucoup
Masquage d'interruption	Possible	Difficile
Programme dans l'espace noyau	Oui	Difficile

7.4 – Exemples

Quelques exemples de systèmes d'exploitation temps réel :

- **FreeRTOS**
- **Micrium uC/OS-II** ou **III**
- Wind River **VxWorks**
- **Texas Instrument RTOS**
- **QNX**
- **RTX** (extension Windows)

8 – Spécificités embarqué

8.1 – Définition

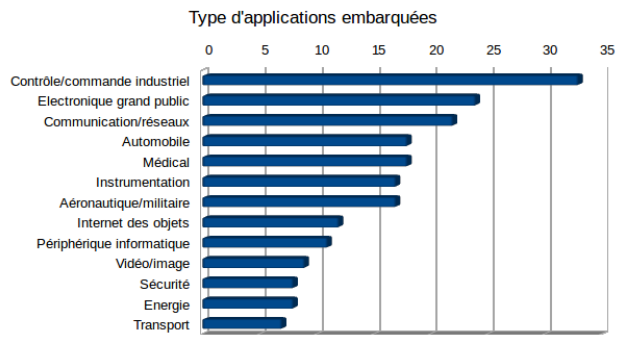
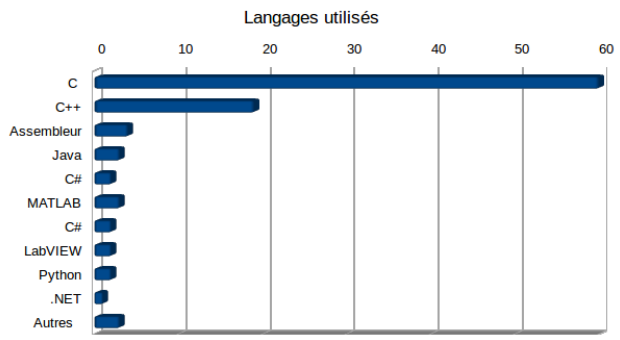
Un système **embarqué** est un système électronique et informatique autonome aux ressources limitées (puissance, espace, énergie, ...).

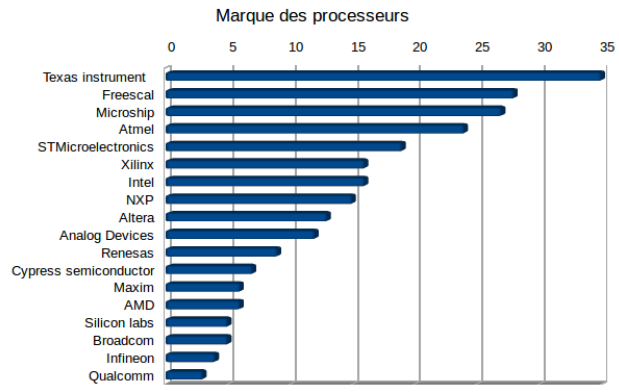
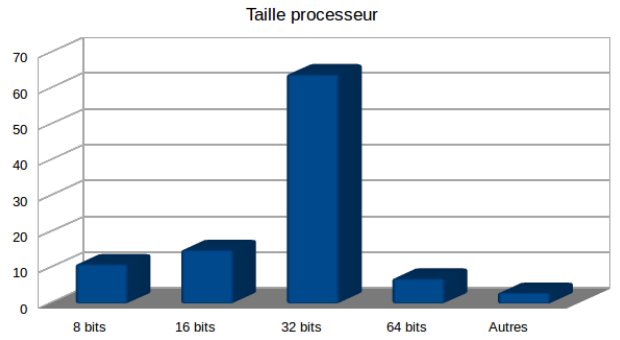
On les retrouve dans beaucoup d'objets de notre quotidien : véhicules, téléphones, TV, box internet, ... et cela devrait encore continuer avec l'internet des objets.

8.2 – Enjeux économiques

Transverses aux différents secteurs économiques, en 2013, les systèmes embarqués représentent en France un marché évalué à **73 milliards d'euros**, soit 3,7% du PIB. En termes d'emploi, la filière regroupe 1,3% des actifs, soit 387.500 personnes (*source : Wikipédia*)

Quelques autres données sur le secteur de l'embarqué en image :

Type d'application	Langages utilisés
	

Marque processeur	Type processeur
	

8.3 – Firmware

Un **firmware** est un programme spécifique chargé d'aider le processeur (ou d'autres périphériques) à se lancer au démarrage d'un appareil (ex : BIOS d'un ordinateur).

En effet comme le processeur lit les instructions depuis la mémoire, il faut bien qu'un programme spécifique prépare la mémoire avec les instructions ua bon endroit permettant aussi au processeur d'être opérationnel automatiquement au démarrage de la machine.

8.4 – Contraintes d'IHM

Longtemps les systèmes embarqués étaient autonomes sans affichage ou alors configurable via un réseau (HTTP, SNMP, telnet). Mais de plus en plus les besoins d'IHM se font sentir sur ces appareils.

Contraintes sur les périphériques de sortie

- LEDS
- afficheur LCD
- Ecrans réduits

Contraintes sur les périphériques d'entrée

- Boutons
- Télécommandes
- Joystick
- Ecran tactiles
- Reconnaissance vocale

9 – Programmation réseau

9.1 – Les sockets

9.1.1 - Généralités

Les **sockets** sont des liens bidirectionnels de communication qui permettent de réaliser des applications réseaux indépendamment du matériel et du système d'exploitation utilisé.

Il existe différentes implémentations des sockets et ce dans de multiples langages dont en C/C++ :

- sockets **POSIX** (Unix, Linux, BSD, OSX...)
- sockets **Winsock** (Windows)

9.1.2 – Modes de communication

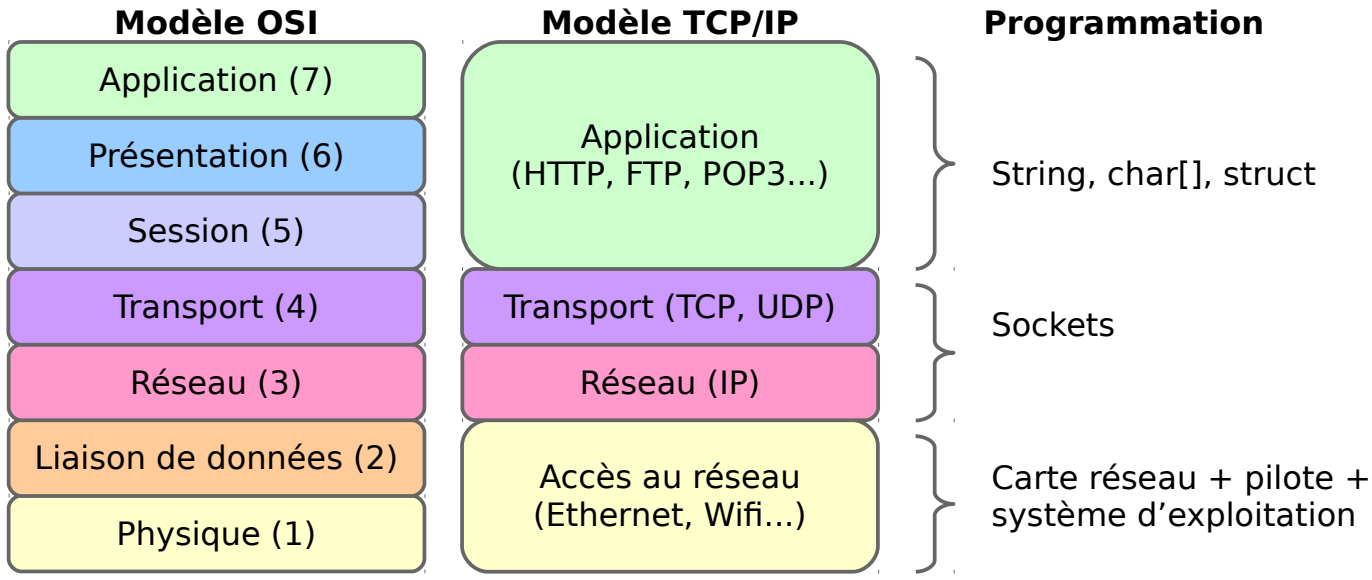
On distingue ainsi deux modes de communication :

Mode	*Connecté*	*Non connecté*
Comparaison	Téléphone	Carte postale
Modèle OSI	TCP	UDP
Avantage	Fiabilité	Rapidité

9.2 – Rappels sur les réseaux

9.2.1 – Sockets VS modèle OSI

Situation des sockets dans le modèle OSI et TCP/IP :

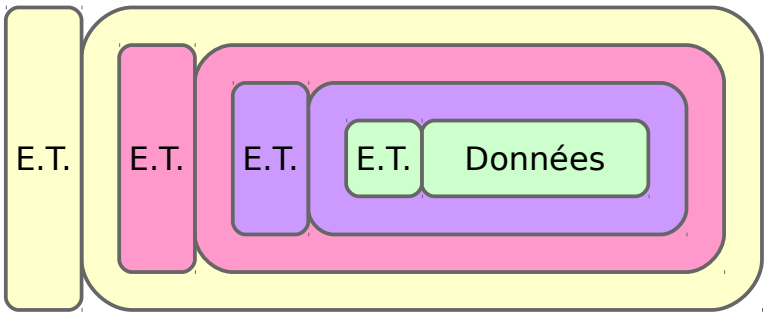


9.2.2 – Encapsulation

Pour chaque couche, on trouve :

- en-tête
- données

Chaque couche se trouve encapsulée dans la couche inférieure.

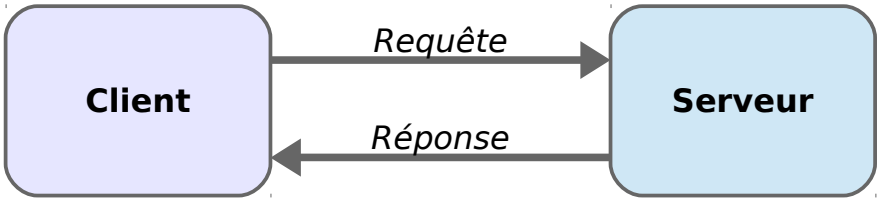


Exemple avec un paquet capturé par Wireshark :

```
00 07 e9 19 f6 5a 00 50 bf a1 52 03 08 00 45 10
00 3e df 7e 40 00 40 06 6c e7 52 f4 89 7b d4 1b
3f 03 82 f2 00 15 b5 98 96 56 2a 84 2e d0 80 18
16 d0 2c 82 00 00 53 54 4f 52 20 72 65 63 65 74
74 65 5f 66 61 72 63 69 2e 74 78 74 0d 0a
```

9.2.3 – Modèle client/serveur

Schéma du modèle client/serveur :

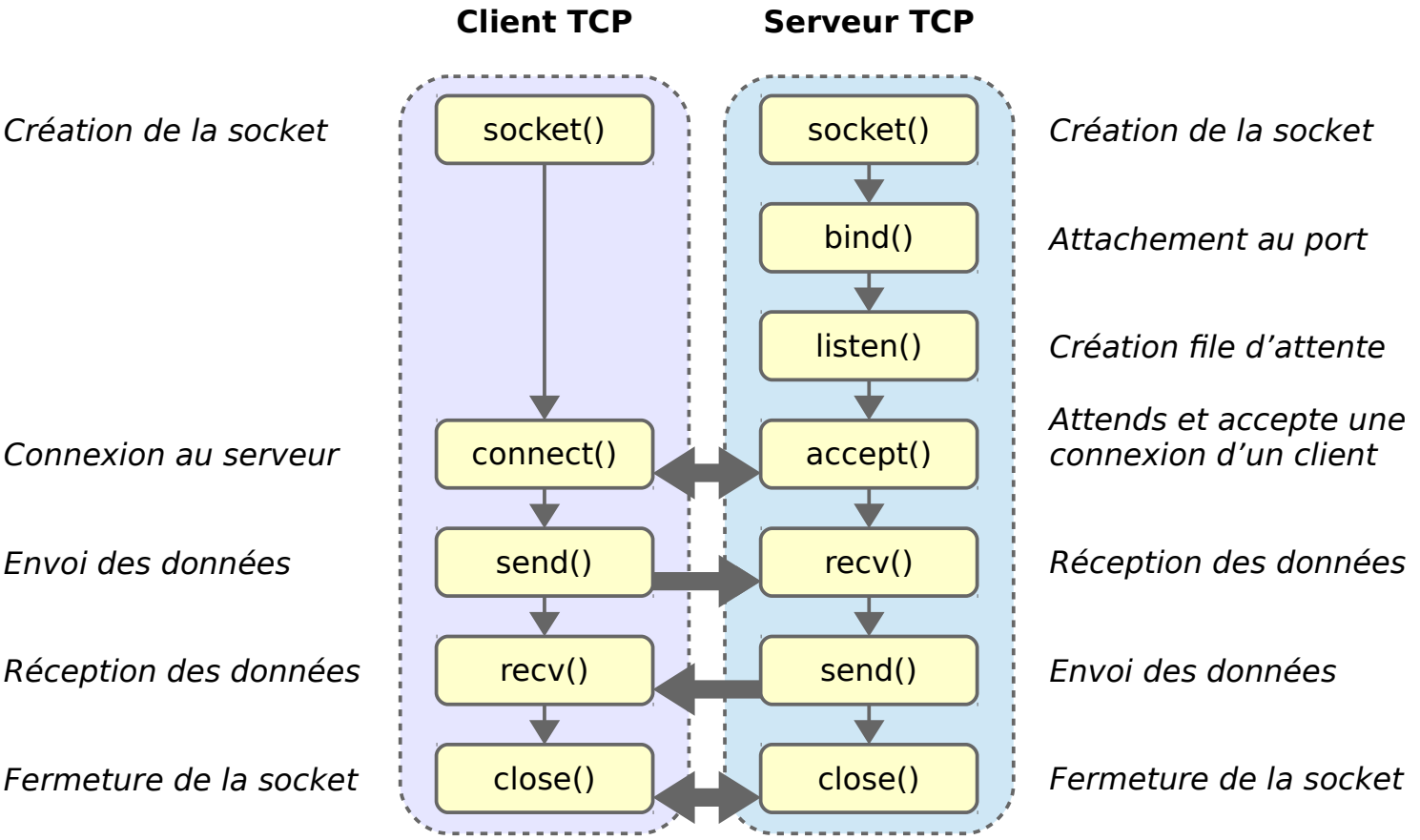


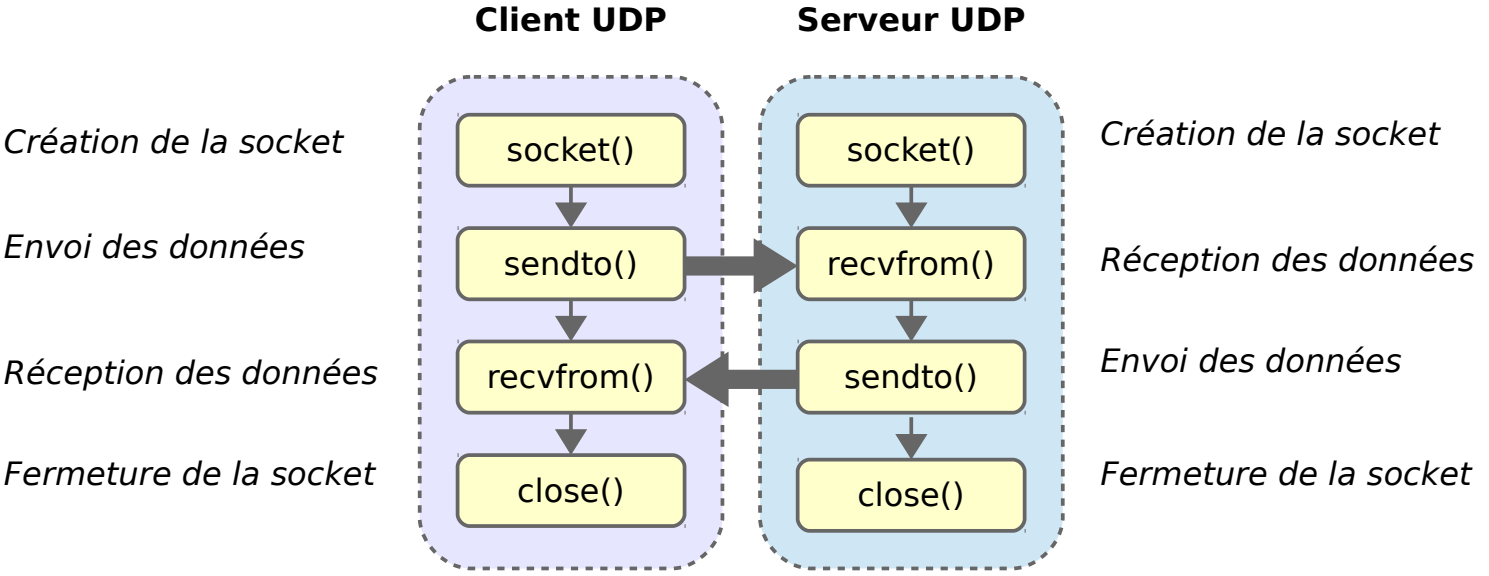
Le client prend l'initiative de la connexion et adresse une requête au serveur. Ce dernier, la traite et retourne ensuite une réponse au client.

9.3 – Les sockets POSIX

9.3.1 – Panorama des fonctions

La programmation réseau avec les sockets POSIX utilise un dizaine de fonctions avec chacune un rôle bien spécifique.





9.3.2 – Client TCP

Exemple complet commenté d'un client TCP :

```cpp
// Inclusions des librairies nécessaires
#include <iostream>
#include <unistd.h>
#include <cerrno>
#include <sys/un.h>
#include <arpa/inet.h>

// Espace de nom utilisé
using namespace std;


// Point d'entrée du programme
int main()
{
    // Création de la socket en mode connecté (TCP)
    int sd = socket(AF_INET, SOCK_STREAM, 0);

    // Configuration des paramètres de connexion
    struct sockaddr_in adresse;
    adresse.sin_family = AF_INET;
    adresse.sin_port = htons(1664);
    adresse.sin_addr.s_addr = inet_addr("127.0.0.1");

    // Connexion au serveur TCP
    connect(sd, (struct sockaddr *)&adresse, sizeof(struct sockaddr));

    // Envoi d'une requête au serveur
    string requete = "Hello world!";
    send(sd, requete.c_str(), requete.size(), 0);

    // Lecture de la réponse du serveur pour affichage
    char bufferReception[1024];
    memset(bufferReception, 0x00, 1024);
    int nbOctets = recv(sd, bufferReception, sizeof(bufferReception), 0);
    string reponse(bufferReception);
    cout << reponse << endl;

    // Fermeture de la socket
    close(sd);

    // Fin du programme
    return 0;
}
```

9.3.3 – Serveur TCP

Exemple complet et commenté d'un serveur TCP :

```cpp
// Librairies nécessaires
#include <iostream>
#include <unistd.h>
#include <cerrno>
#include <sys/un.h>
#include <arpa/inet.h>

// Espace de nom utilisé
using namespace std;

// Point d'entrée du programme
int main()
{
    // Création de la socket serveur
    int sd_serveur = socket(AF_INET, SOCK_STREAM, 0);

    // Configuration de la socket, notamment le port d'écoute
    struct sockaddr_in cfg_serveur;
    cfg_serveur.sin_family = AF_INET;
    cfg_serveur.sin_addr.s_addr = htonl(INADDR_ANY);
    cfg_serveur.sin_port = htons(1664);

    // Attachement de la socket au port défini
    bind(sd_serveur, (struct sockaddr *)&cfg_serveur, sizeof(cfg_serveur));

    // Création une file d'attente de connexion
    listen(sd_serveur, 5);

    while(1)
    {
        // Dès qu'un nouveau client se connecte à notre serveur,
        // une nouvelle socket est créée pour gérer le client
        int sd_client = accept(sd_serveur, NULL, NULL);

        // Réception de la requête du client
        char buffer[1024];
        memset(buffer, 0x00, 1024);
        int nbOctets = recv(sd_client, buffer, sizeof(buffer), 0);
        string reponse(buffer);
        cout << reponse << endl;

        // Envoi de la réponse au client
        string requete = "Hello world!";
        send(sd_client, requete.c_str(), requete.size(), 0);

        // Fermeture de la socket client
        close(sd_client);
    }

    // Fermeture de la socket serveur
    close(sd_serveur);

    // Fin du programme
    return 0;
}
```

9.3.4 – Client UDP

Exemple commenté d'un client UDP:

```
// Inclusions des librairies nécessaires
#include <iostream>
#include <unistd.h>
#include <cerrno>
#include <sys/un.h>
#include <arpa/inet.h>

// Espace de nom utilisé
using namespace std;

// Point d'entrée du programme
int main()
{
        // Création du socket en mode non connecté (UDP)
        int sd = socket(AF_INET, SOCK_DGRAM, 0);

        // Configuration des paramètres de connexion
        struct sockaddr_in cfg_serveur;
        cfg_serveur.sin_family = AF_INET;
        cfg_serveur.sin_port = htons(1664);
        cfg_serveur.sin_addr.s_addr = inet_addr("127.0.0.1");

        // Envoi d'une requête
        string requete = "Hello world!";
        sendto(sd, requete.c_str(), requete.length(), 0, (struct sockaddr
*)&cfg_serveur, sizeof(cfg_serveur));

        // Lecture de la réponse du serveur pour affichage
        char buffer[1024];
        memset(buffer, 0x00, 1024);
        recvfrom(sd, buffer, sizeof(buffer), 0, NULL, NULL);
        string reponse(buffer);
        cout << reponse << endl;

        // Fermeture de la socket
        close(sd);

        // Fin du programme
        return 0;
}
```

9.3.5 – Serveur UDP

Le code source du serveur UDP sera le même que précédemment en inversant juste l'ordre d'appel des fonctions :

- recvfrom()
- sendto()

9.3.6 – Gestion des erreurs

Chaque fonction rencontrée précédemment retourne la valeur -1 en cas d'erreur ou de problème.

Il est vivement recommandé de vérifier le résultat de chaque fonction en utilisant la fonction strerror comme ci-dessous :

```
#include <cerrno>

// Connexion au serveur TCP
int test = connect(sd, (struct sockaddr *)&adresse, sizeof(struct sockaddr));
if (test < 0)
{
    cerr << "Erreur avec connect() : " << strerror(errno) << endl;
    return -1;
}
```

9.3.7 – Informations sur le correspondant

Le serveur peut obtenir des informations sur le client qui s'est connecté (en TCP) ou qui lui a envoyé un paquet (en UDP), notamment :

- son adresse IP
- son port (TCP ou UDP)

Exemple d'utilisation avec la fonction accept() en TCP :

```
struct sockaddr_in cfg_client;
socklen_t lg = sizeof(cfg_client);
int sd_client = accept(sd_serveur, (struct sockaddr *)&cfg_client, &lg);
cout << "Nouvelle connexion d'un client :" << endl;
cout << "  + IP => " << inet_ntoa(cfg_client.sin_addr) << endl;
cout << "  + port => " << htons(cfg_client.sin_port) << endl;
```

On pourra procéder à l'identique avec les 2 derniers paramètres de la fonction recvfrom().

10 – Ressources

10.1 – Bibliographie

Quelques livres pour aller plus loin :

- **Systèmes d'exploitation** – A. Tanenbaum – Pearson – 2008
- **Système temps-réel de contrôle/commande** – F. Cottet et E. Grolleau – Dunod – 2005

10.2 – Webographie

Quelques liens pour aller plus loin :

- https://fr.wikipedia.org/wiki/Liste_des_syst%C3%A8mes_d'exploitation (Liste des systèmes d'exploitation)
- https://interstices.info/jcms/nn_72288/la-naissance-des-systemes-d-exploitation (histoire de la naissance des OS)
- http://kos.enix.org/ (Comment créer son propre OS)
- ...