Programmation Système Multi-tâche

Thierry Vaira

BTS SN Option IR

v.1.1 - 20 mars 2018

Sommaire

Qu'est-ce que le multitâche ?

- Un système d'exploitation est **multitâche** (*multitasking*) s'**il permet d'exécuter, de façon apparemment simultanée, plusieurs programmes informatiques** (*processus*).
- La simultanéité apparente est le résultat de l'**alternance rapide d'exécution des processus présents en mémoire** (notion de **temps partagé et de multiplexage**).
- Le passage de l'exécution d'un processus à un autre est appelé **commutation de contexte**.
- Ces commutations peuvent être initiées par les programmes eux-mêmes (**multitâche coopératif**) ou par le système d'exploitation (**multitâche préemptif**). Tous les systèmes utilisent maintenant le multitâche préemptif.

Remarques

- Le **multitâche** n'est pas dépendant du **nombre de processeurs** présents physiquement dans l'ordinateur : un système multiprocesseur (ou multi-cœur) n'est pas nécessaire pour exécuter un système d'exploitation multitâche.
- Le **multitâche coopératif** n'est plus utilisé (cf. Windows 3.1 ou MAC OS 9).
- Unix et ses dérivés, Windows et MAC OS X sont des systèmes basés sur le **multitâche préemptif**.

À quoi ça sert ?

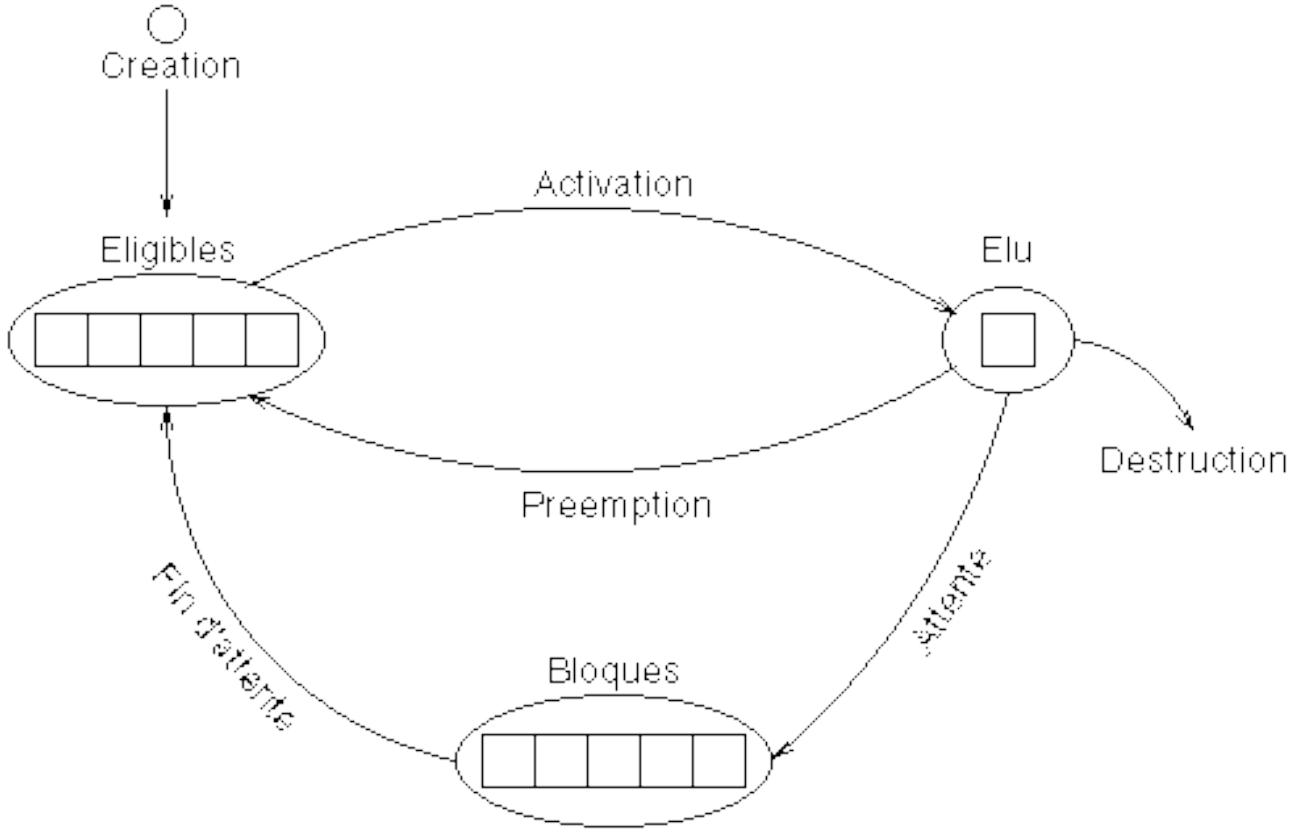
- Le multitâche permet de **paralléliser les traitements** par l'exécution simultanée de programmes informatiques.
- Exemples de besoins :
  - permettre à plusieurs utilisateurs de travailler sur la même machine.
  - utiliser un traitement de texte tout en naviguant sur le Web.
  - transférer plusieurs fichiers en même temps.
  - améliorer la conception : écrire plusieurs programmes simples, plutôt qu'un seul programme capable de tout faire, puis de les faire coopérer pour effectuer les tâches nécessaires.

Comment ça marche ?

- La **préemption** est la capacité d'un système d'exploitation multitâche **à suspendre un processus au profit d'un autre**.
- Le **multitâche préemptif** est assuré par l'**ordonnanceur** (*scheduler*), un service de l'OS.
- L'**ordonnanceur** distribue le **temps du processeur entre les différents processus**. Il peut aussi interrompre à tout moment un processus en cours d'exécution pour permettre à un autre de s'exécuter.
- Une **quantité de temps définie** (*quantum*) est attribuée par l'**ordonnanceur** à chaque processus : les processus ne sont donc pas autorisés à prendre un temps non-défini pour s'exécuter dans le processeur.

L'ordonnancement en action

Dans un ordonnancement (statique à base de priorités) avec préemption, un processus peut être préempté (remplacé) par n'importe quel processus plus prioritaire qui serait devenu prêt.



Synthèse

- Multitâche : exécution en parallèle de plusieurs tâches (*processus* ou *threads*).
- Commutation de contexte : passage de l'exécution d'un processus à un autre.
- Multitâche préemptif : mode de fonctionnement d'un système d'exploitation multitâche permettant de partager de façon équilibrée le temps processeur entre différents processus.
- Ordonnancement : mécanisme d'attribution du processeur aux processus (blocage, déblocage, élection, préemption).

Définitions

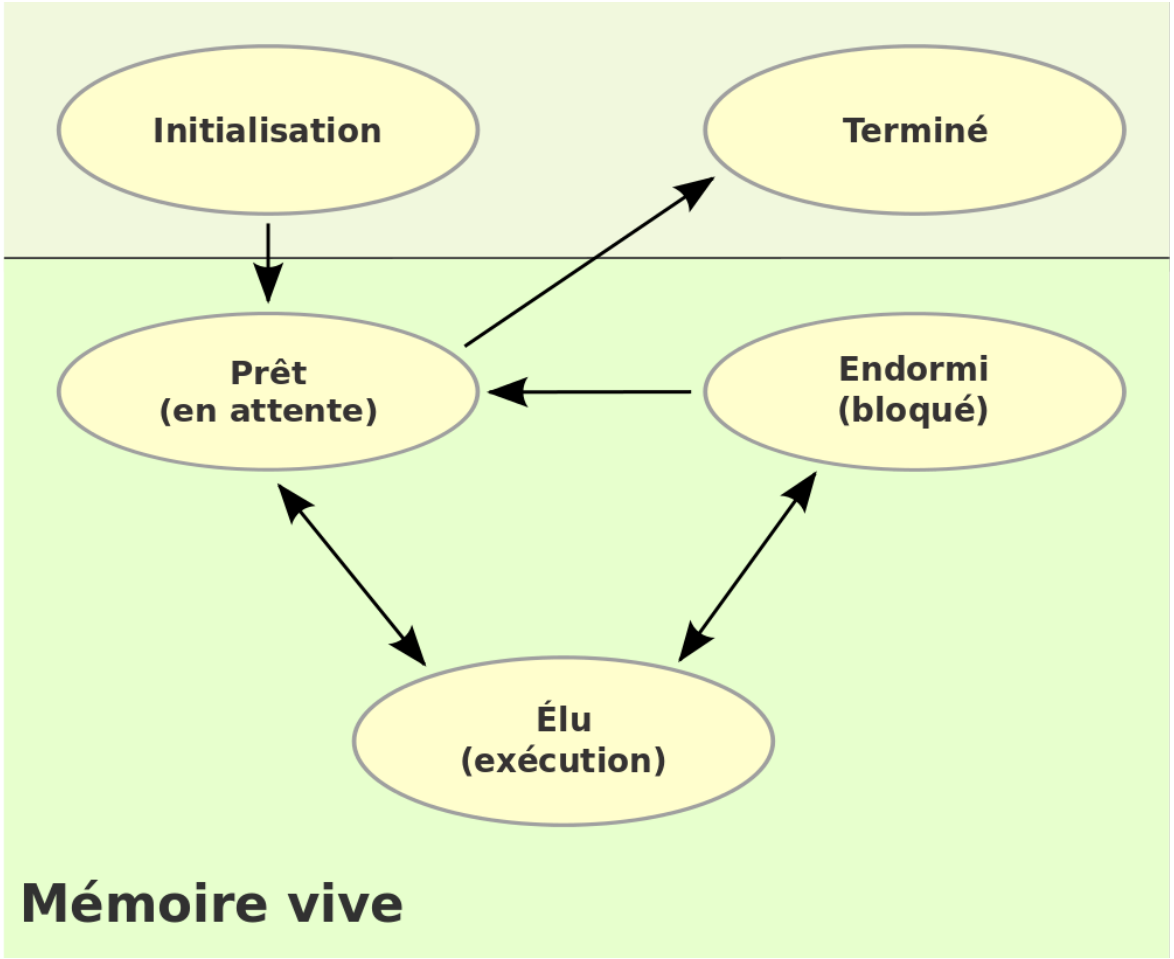
- Un **processus** (*process*) est un **programme en cours d'exécution** par un système d'exploitation.
- Un **programme** est une **suite d'instructions permettant de réaliser un traitement**. Il revêt un caractère statique.
- Une **image** représente l'**ensemble des objets (code, données, ...) et des informations (états, propriétaire, ...)** qui peuvent donner lieu à une exécution dans l'ordinateur.
- Un **processus** est donc l'**exécution d'une image**. Le processus est l'aspect dynamique d'une image.
- Un **fil d'exécution** (*thread*) est l'**exécution séquentielle d'une suite d'instructions**.

Rôle du système d'exploitation

- *Rappel : un système d'exploitation multitâche est capable d'exécuter plusieurs processus de façon quasi-simultanée.*
- Le **système d'exploitation** est chargé d'**allouer les ressources** (mémoires, temps processeur, entrées/sorties) nécessaires aux processus et d'**assurer son fonctionnement isolé** au sein du système.
- Un des rôles du système d'exploitation est d'**amener en mémoire centrale l'image mémoire d'un processus avant de l'élire et de lui allouer le processeur**. Le système d'exploitation peut être amené à sortir de la mémoire les images d'autres processus et à les copier sur disque. Une telle gestion mémoire est mise en oeuvre par un algorithme de **va et vient appelée aussi *swapping***.
- Il peut aussi fournir une **API** pour **permettre leur gestion et la communication inter-processus (IPC)**.

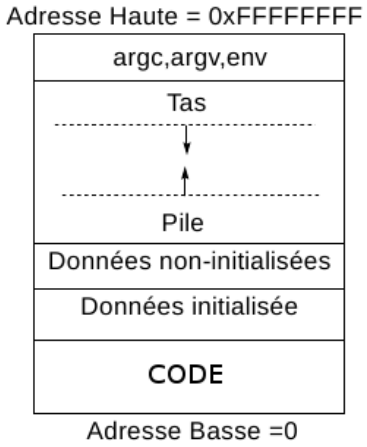
États d'un processus

- Ces états existent dans la plupart des systèmes d'exploitation :



Qu'est-ce qu'un processus ?

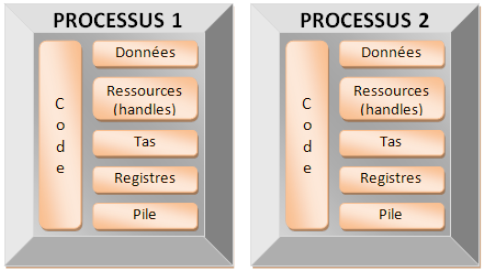
- L'image d'un **processus** comporte du **code machine exécutable**, une **zone mémoire** pour les données allouées par le processus, une **pile** ou *stack* (pour les variables locales des fonctions et la gestion des appels et retour des fonctions) et un **tas** ou *heap* pour les allocations dynamiques (`new`, `malloc`).



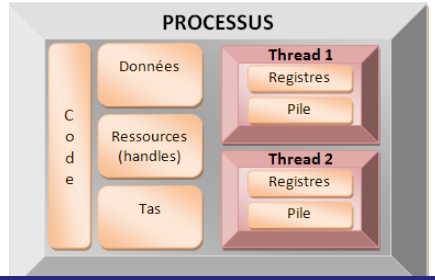
- Ce processus est une entité qui, de sa création à sa mort, est **identifié par une valeur numérique** : le **PID** (*Process IDentifier*).
- Chaque processus a un **utilisateur propriétaire**, qui est utilisé par le système pour **déterminer ses permissions d'accès aux ressources** (fichiers, ports réseaux, ...).

Processus lourd et léger

- *Rappel : L'exécution d'un processus se fait dans son contexte. Quand il y a changement de processus courant, il y a une commutation ou changement de contexte.*
- En raison de ce contexte, la plupart des systèmes offrent la distinction entre :
  - **« processus lourd »**, qui sont complètement isolés les uns des autres car ayant chacun leur contexte, et

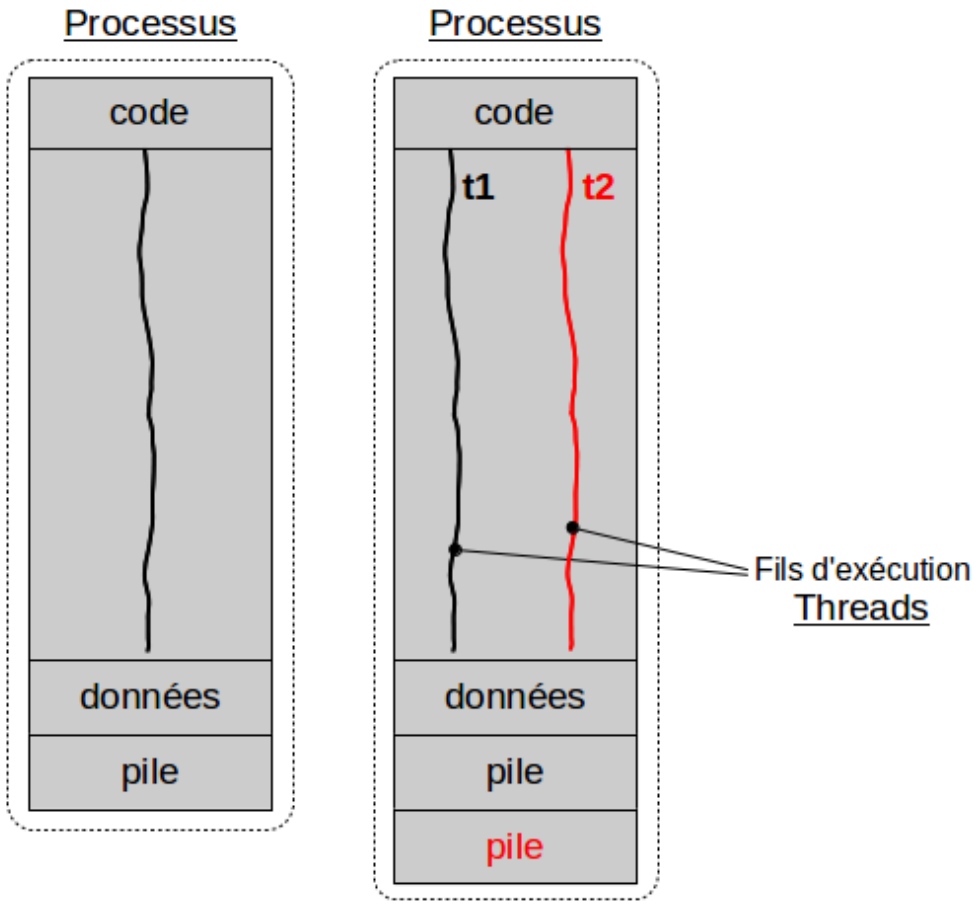


  - **« processus légers » (*threads*)**, qui partagent un contexte commun sauf la pile (les *threads* possèdent leur propre pile d'appel).



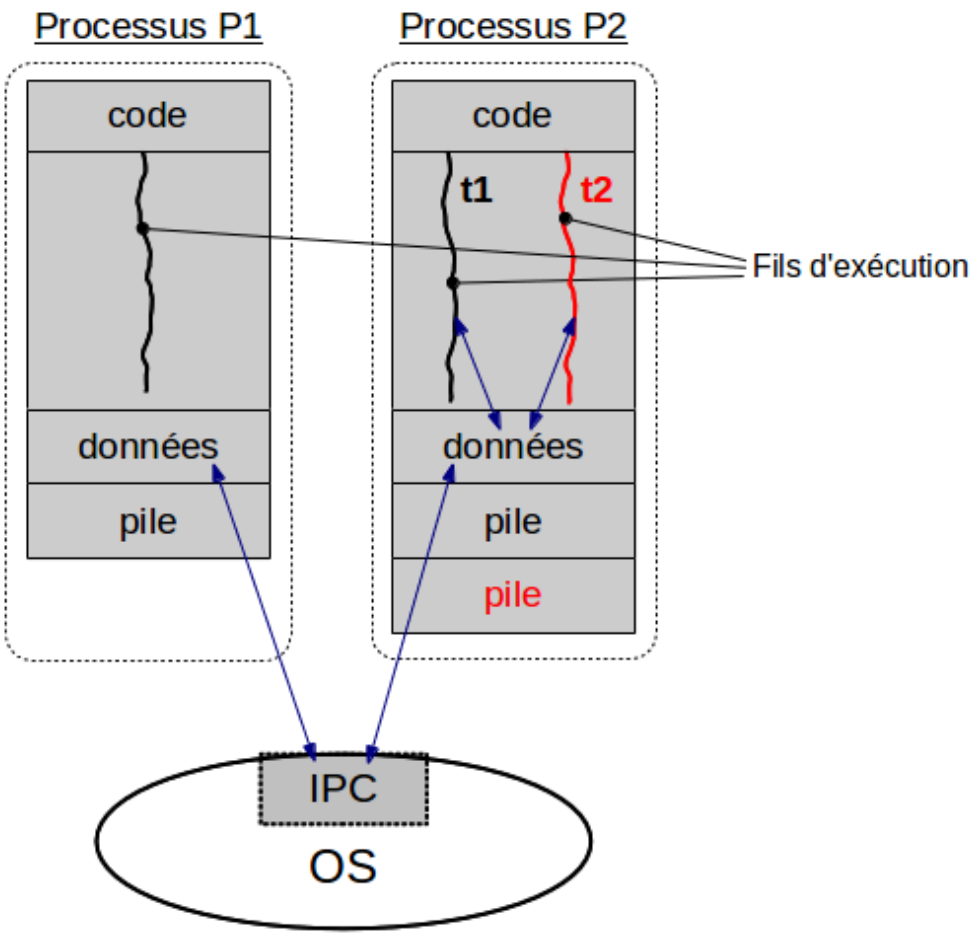
Processus lourd et léger (suite)

- Tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur (notion de fil).



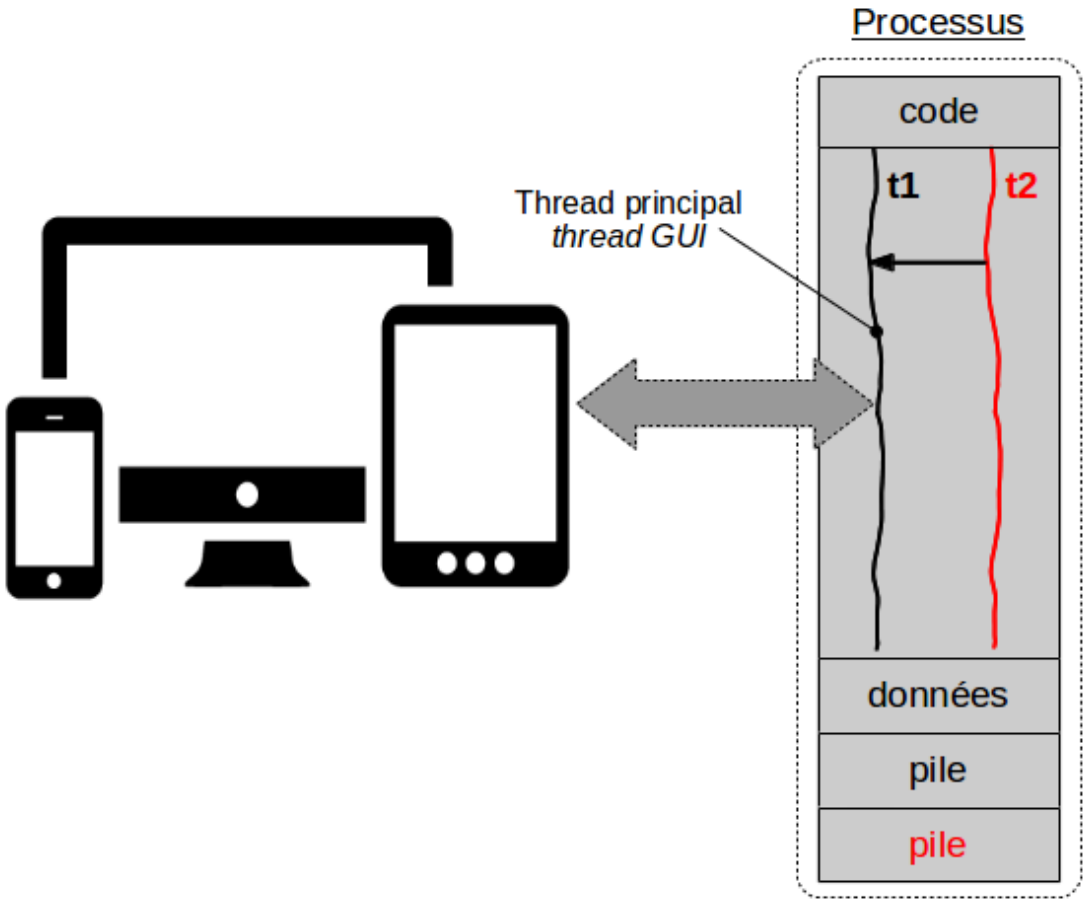
Processus lourd et léger (fin)

- Les processus P1 et P2 ne partagent pas le même espace pour les données. Il faudra donc mettre en place des IPC (*Inter Processus Communication*).
- Les threads T1 et T2 partagent le même espace pour les données. Par contre, il faudra mettre en place une synchronisation en cas d'accès concurrent.



Cas du thread GUI

- Seul le *thread* principal GUI (*Graphical User Interface*) peut accèder aux ressources graphiques (*widegts*). Il faudra alors prévoir une communication *inter-threads* si un *thread* non-gui désire afficher ou interagir avec l'interface graphique.



Processus lourds vs processus légers

- **Multitâche moins coûteux pour les *threads* (processus léger)** : puisqu'il n'y a pas de changement de mémoire virtuelle, la commutation de contexte (*context switch*) entre deux *threads* est moins coûteuse que la commutation de contexte obligatoire pour des processus lourds.
- **Communication entre *threads* plus rapide et plus efficace** : grâce au partage de certaines ressources entre *threads*, les IPC (*Inter Processus Communication*) sont inutiles pour les *threads*.
- **Programmation utilisant des *threads* est toutefois plus difficile** : obligation de mettre en place des mécanismes de synchronisation, risques élevés d'interblocage, de famine, d'endormissement.

Synthèse

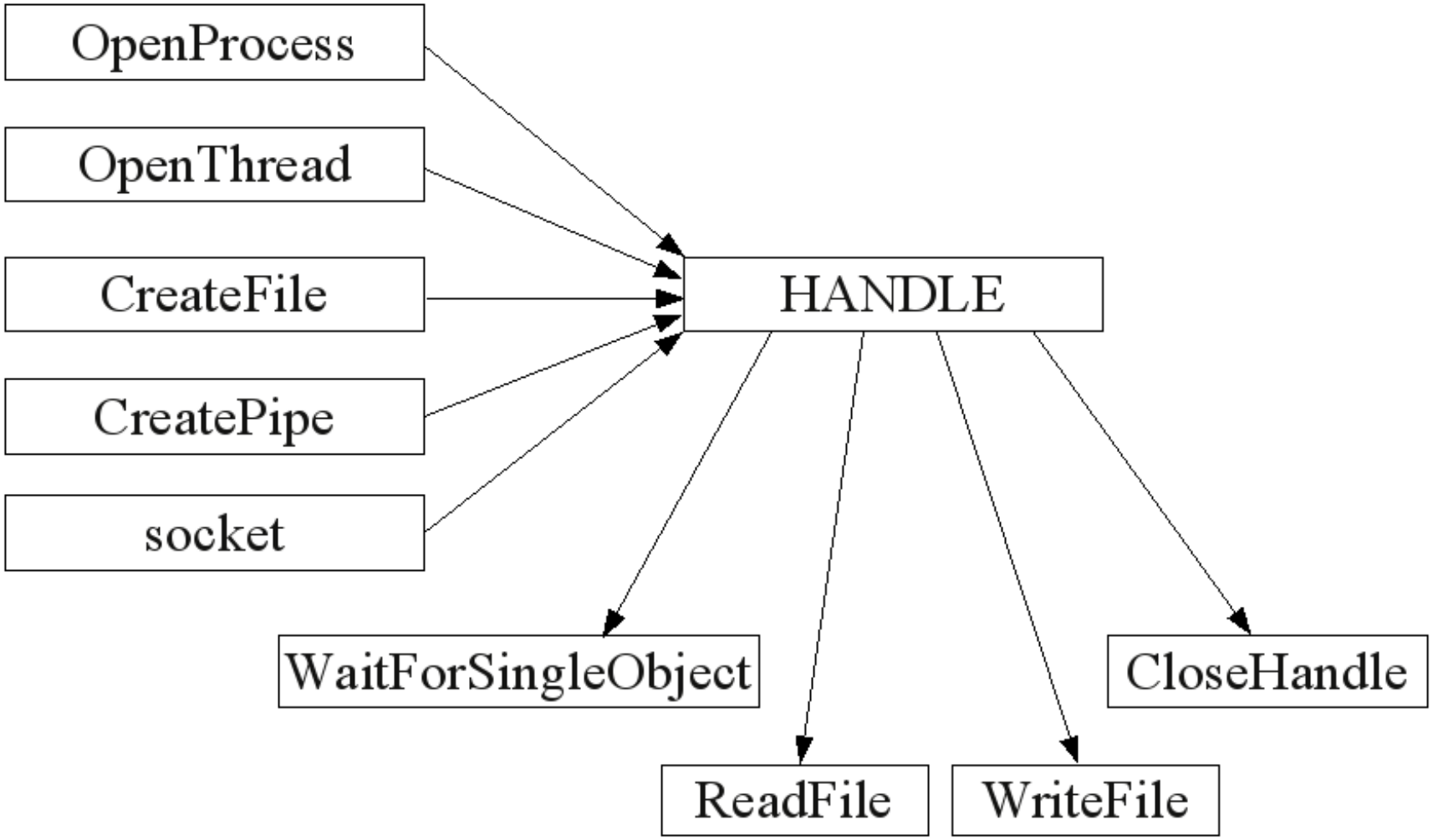
- Processus : programme en cours d'exécution. C'est l'exécution d'une image composée de code machine et de données mémoire.
- Contexte : image d'un processus en mémoire auquel s'ajoute son état (registres, ...).
- Processus lourd : c'est un processus « normal ». La création ou la commutation de contexte d'un processus a un coût pour l'OS. L'exécution d'un processus est réalisée de manière isolée par rapport auxtres processus.
- Processus léger : c'est un *thread*. Un processus lourd peut englober un ou plusieurs *threads* qui partagent alors le même contexte. C'est l'exécution d'une fonction (ou d'une méthode) au sein d'un processus et en parallèle avec les autres *threads* de ce processus. La commutation de *thread* est très rapide car elle ne nécessite pas de commutation de contexte.

Interface de programmation

- *Rappel : le noyau d'un système d'exploitation est vu comme un* ***ensemble de fonctions*** *qui forme l'****API****.*
- Chaque fonction ouvre l'**accès à un service offert par le noyau**.
- Ces fonctions sont regroupées au sein de la **bibliothèque** des **appels systèmes** (*system calls*) pour UNIX/Linux ou **WIN32** pour Windows.
- **POSIX** (*Portable Operating System Interface*) est une **norme relative à l'interface de programmation du système d'exploitation**. De nombreux systèmes d'exploitation sont conformes à cette norme, notamment les membres de la famille **Unix**.

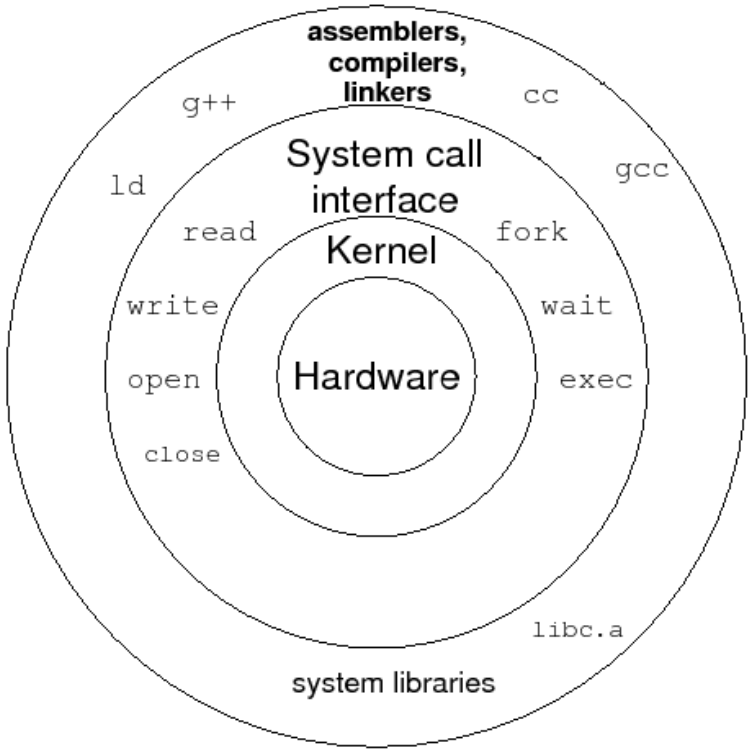
L'API Win32

- L'**API Windows** est orientée « **handle** » et non fichier.
- Un **handle** est un **identifiant d'objet système**.



L'API System Calls

- L'**API System Calls** d'UNIX est orientée « **fichier** » car dans ce système : TOUT est FICHIER !
- Un **descripteur de fichier** est **une clé abstraite** (c'est un entier) pour accéder à un fichier, c'est-à-dire le plus souvent une ressource du système.

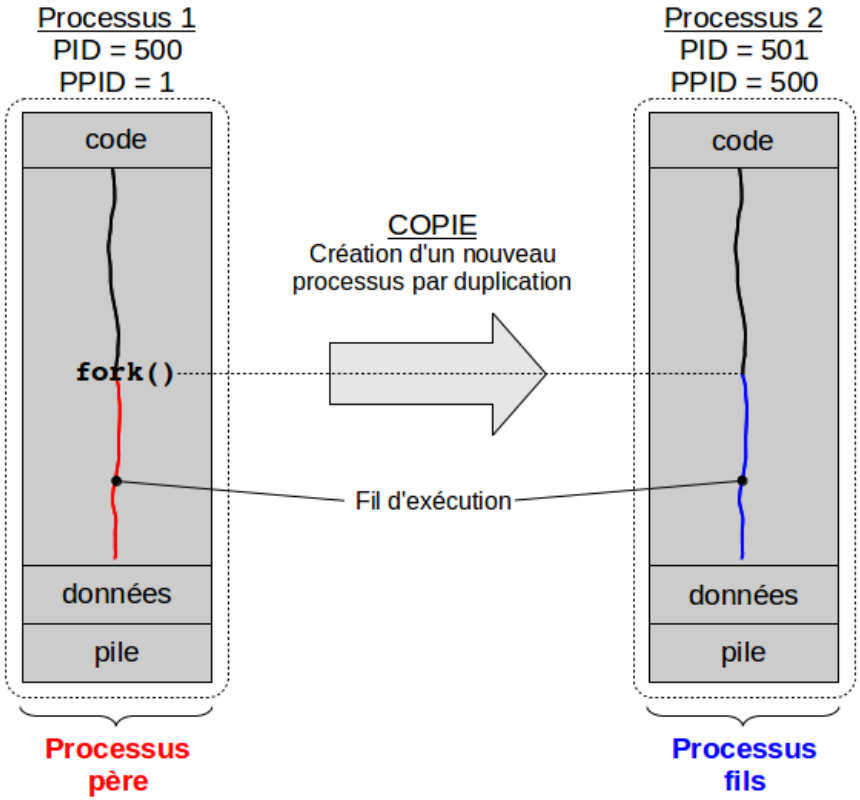


API WIN32 vs System Calls

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Création dynamique de processus

- Lors d'une opération `fork`, le noyau Unix crée un nouveau processus qui est une **copie conforme du processus père**. Le code, les données et la pile sont copiés et tous les fichiers ouverts par le père sont ainsi hérités par le processus fils.

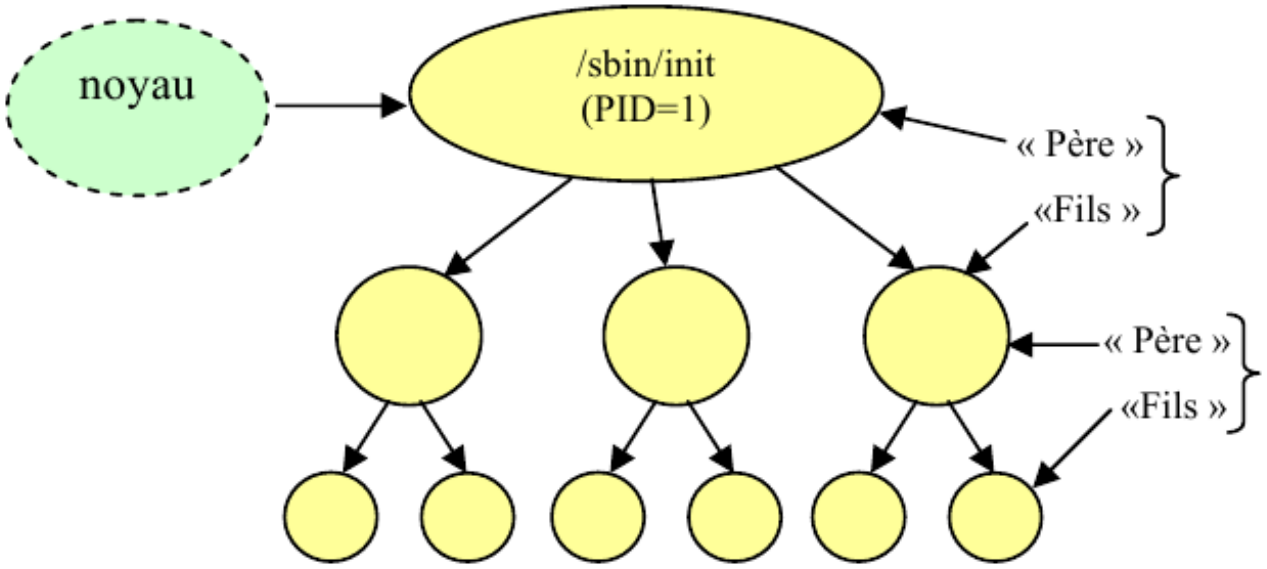


Exécution d'un nouveau processus

- *Rappel : après une opération fork, le noyau Unix a créé un nouveau processus qui est une copie conforme du processus qui a réalisé l'appel.*
- Si l'on désire exécuter du code à l'intérieur de ce nouveau processus, on utilisera un appel de type `exec` : `execl`, `execlp`, `execle`, `execv` ou `execvp`.
- La famille de fonctions `exec` remplace l'**image mémoire du processus en cours par un nouveau processus**.

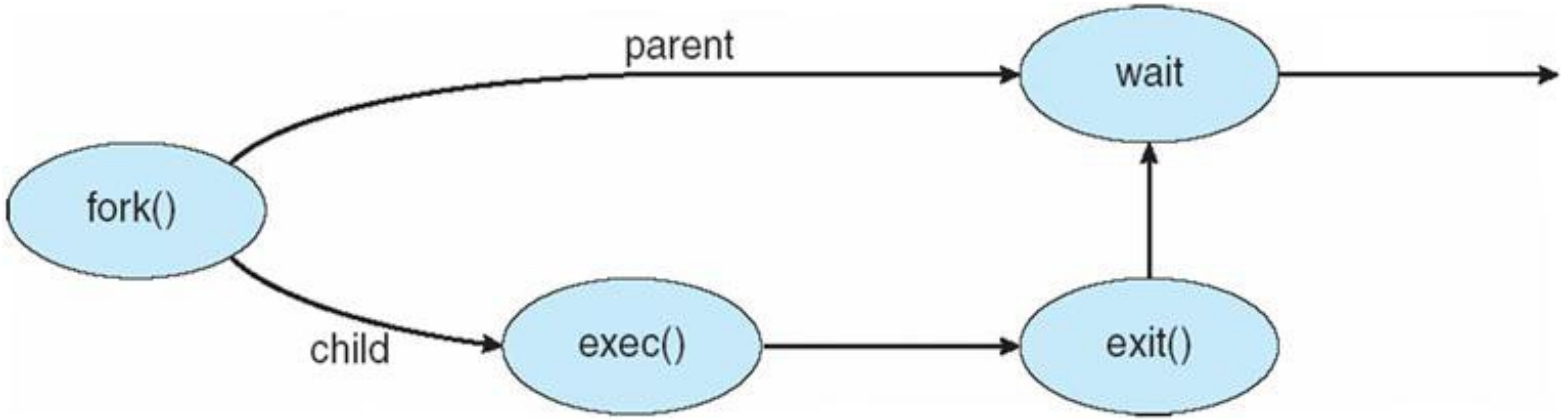
Généalogie des processus

- Chaque processus est identifié par un numéro unique, le **PID** (*Processus IDentification*). Un processus est créé par un autre processus (notion père-fils). Le **PPID** (*Parent PID*) d'un processus correspond au PID du processus qui l'a créé (son père).



Synchronisation des processus

- On ne peut présumer l'ordre d'exécution de ces processus (cf. politique de l'ordonnanceur).
- Il sera donc impossible de déterminer quels processus se termineront avant tels autres (y compris leur père). D'où l'existence, dans certains cas, d'un problème de synchronisation.
- La primitive `wait` permet l'élimination de ce problème en provoquant la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine.



Opérations réalisables sur un processus

- Création : `fork`
- Exécution : `exec`
- Destruction : terminaison normale, auto-destruction avec `exit`, meurtre avec `kill` ou `Ctrl-C`
- Mise en attente/réveil : `sleep`, `wait`, `kill`
- Suspension et reprise : `Ctrl-Z` ou `fg`, `bg`
- Changement de priorité : `nice`

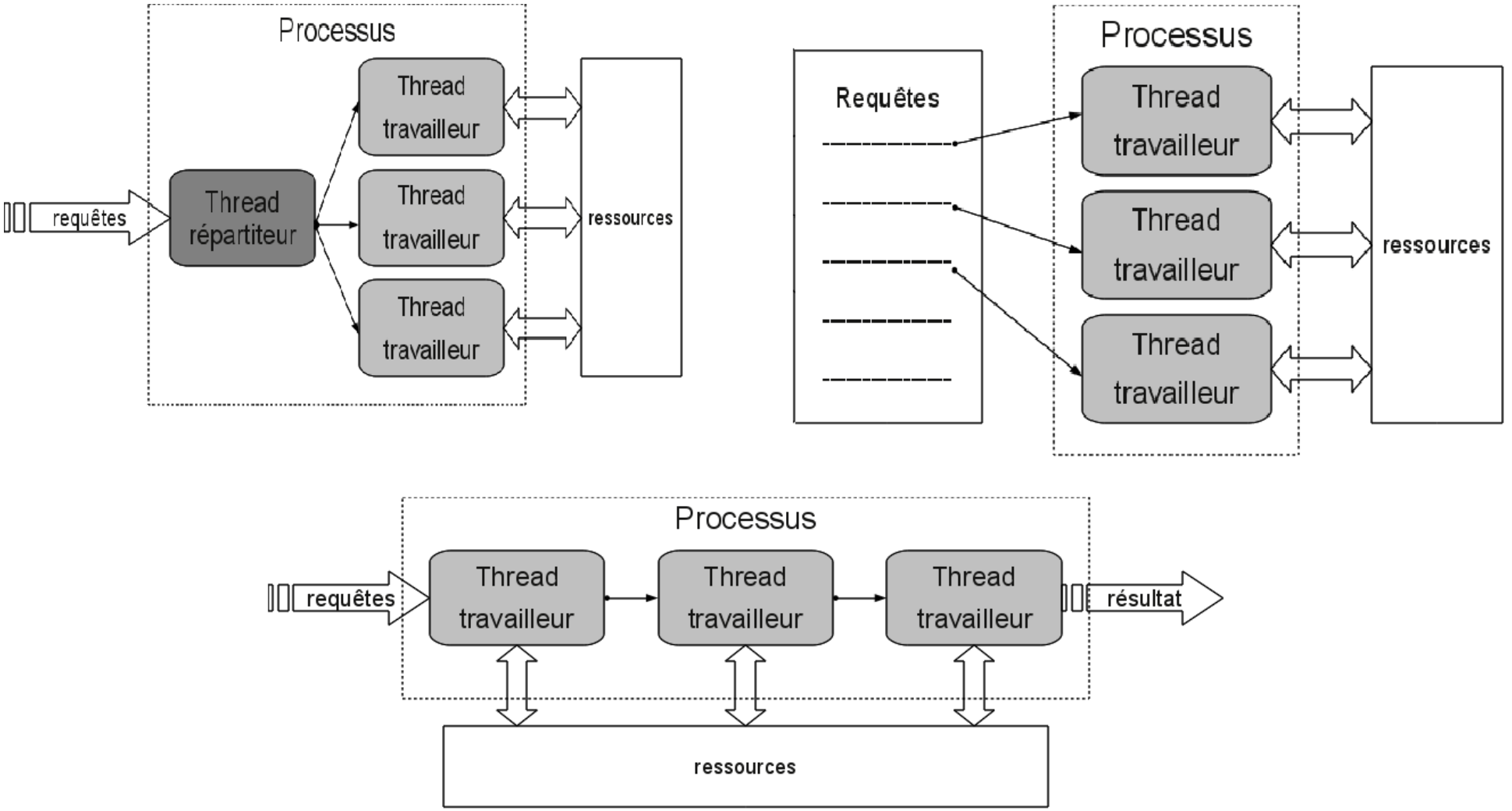
Les threads

- Les systèmes d'exploitation mettent en oeuvre généralement les *threads* :
  - Le standard des processus légers **POSIX** est connu sous le nom de `pthread`. Le standard POSIX est largement mis en oeuvre sur les systèmes **UNIX/Linux**.
  - **Microsoft** fournit aussi une API pour les processus légers : `WIN32 threads` (Microsoft Win32 API threads).
- En **C++**, il sera conseillé d'utiliser un *framework* (**Qt**, **Builder**, commoncpp, boost, ...) qui fournira le support des *threads* sous forme de classes prêtes à l'emploi. La version `C++11` intégrera le support de *threads* en standard.
- **Java** propose l'interface `Runnable` et une classe abstraite `Thread` de base.

Modèle de programmation

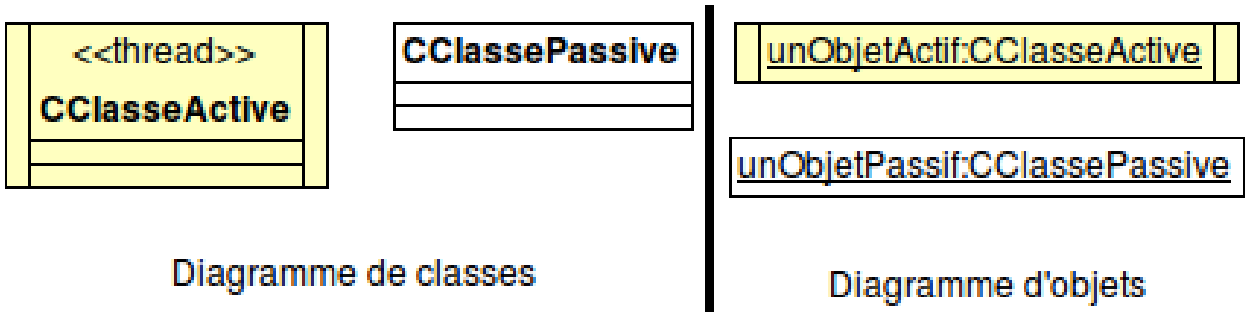
- Un constat : Chaque programme est différent. Cependant, certains modèles communs sont apparus.
- On distingue généralement 3 modèles :
  - **Modèle répartiteur/travailleurs ou maître/esclaves** : Un processus, appelé le répartiteur (ou le maître), reçoit des requêtes pour tout le programme. En fonction de la requête reçue, le répartiteur attribue l'activité à un ou plusieurs processus travailleurs (ou esclaves).
  - **Modèle en groupe** : Chaque processus réalise les traitements concurremment sans être piloté par un répartiteur (les traitements à effectuer sont déjà connus). Chaque processus traite ses propres requêtes (mais il peut être spécialisé pour un certain type de travail).
  - **Modèle en pipeline** : L'exécution d'une requête est réalisée par plusieurs processus exécutant une partie de la requête en série. Les traitements sont effectués par étape du premier processus au dernier. Le premier processus engendre des données qu'il passe au suivant.

Modèles avec des *threads*



Classe active et objet actif

- Une classe active est une classe qui possède une méthode qui s'exécute dans un *thread*. Cela définit alors un **flot de contrôle distinct** (c'est le fil d'exécution). Une **instance d'une classe active** sera nommée **object actif**.
- UML fournit un repère visuel (bord en trait épais ou double trait) qui permet de distinguer les éléments actifs des éléments passifs. Il est conseillé d'ajouter le séréotype « thread ».



Synthèse

- API : c'est une interface de programmation (*Application Programming Interface*) qui est un ensemble de classes, de méthodes ou de fonctions qui offre des services pour d'autres logiciels. Elle est offerte par une bibliothèque logicielle ou un *framework*.
- WIN32 : c'est l'API de Windows qui permet la programmtion système (création de processus ou de thread, communication inter-processus, ...). Elle est orientée *handle*.
- *System calls* : c'est l'API Unix/Linux composée d'appels systèmes pour la création de processus ou de thread, la communication inter-processus, ...). Elle est orientée *fichier*.
- POSIX : c'est une norme d'API que l'on retrouve notamment sous Unix.

Introduction

Dans la programmation concurrente, le terme de **synchronisation** se réfère à deux concepts distincts (mais liés) :

- La **synchronisation de processus ou tâche** : mécanisme qui vise à bloquer l'exécution des différents processus à des points précis de leur programme de manière à ce que tous les processus passent les étapes bloquantes au moment prévu par le programmeur.
- La **synchronisation de données** : mécanisme qui vise à conserver la cohérence entre différentes données dans un environnement multitâche.

Les problèmes liés à la synchronisation rendent toujours la programmation plus difficile.

Définitions

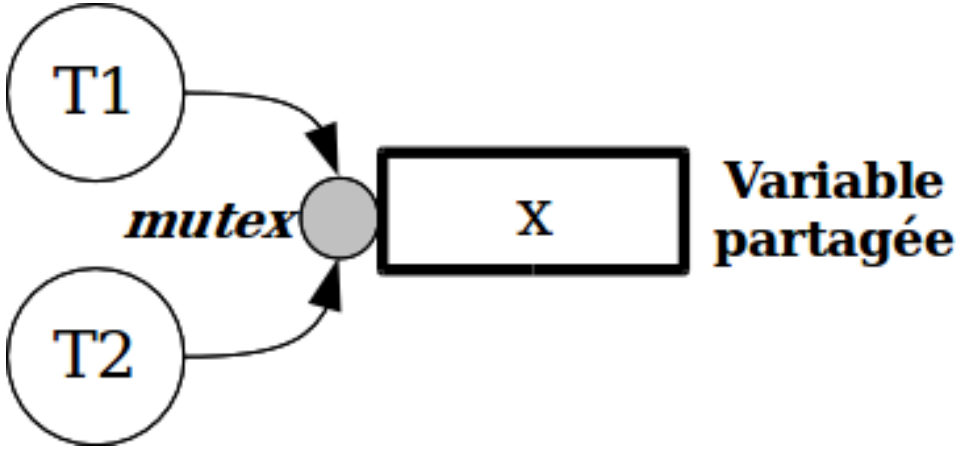
- **Section critique** : C'est une partie de code telle que deux *threads* ne peuvent s'y trouver au même instant.
- **Exclusion mutuelle** : Une ressource est en exclusion mutuelle si seul un *thread* peut utiliser la ressource à un instant donné.
- **Chien de garde (*watchdog*)** : Un chien de garde est une technique logicielle (compteur, *timeout*, signal, ...) utilisée pour s'assurer qu'un programme ne reste pas bloqué à une étape particulière du traitement qu'il effectue. C'est une protection destinée généralement à redémarrer le système, si une action définie n'est pas exécutée dans un délai imparti.

Mécanisme de synchronisation : le mutex

Rappel : La cohérence des données ou des ressources partagées entre les processus légers est maintenue par des mécanismes de synchronisation.

Il existe deux principaux mécanismes de synchronisation de données pour les *threads* : le **mutex** et le sémaphore.

- Un **mutex** (**verrou d'exclusion mutuelle**) possède deux états : verrouillé ou non verrouillé.
- Trois opérations sont associées à un mutex : `lock` pour verrouiller le mutex, `unlock` pour le déverrouiller et `trylock` (équivalent à `lock`, mais qui en cas d'échec ne bloque pas le *thread*).



Mécanisme de synchronisation : le sémaphore

Il existe deux principaux mécanismes de synchronisation de données pour les *threads* : le mutex et le **sémaphore**.

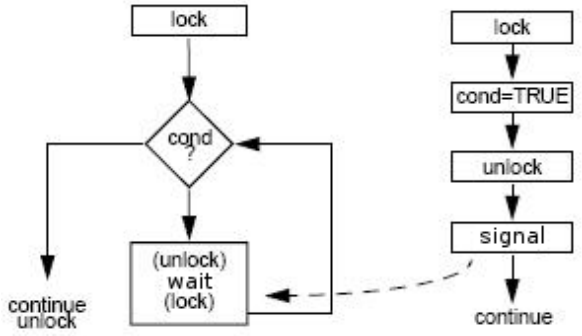
- Un **sémaphore** est un **mécanisme empêchant deux processus ou plus d'accéder simultanément à une ressource partagée**.
- Un **sémaphore général** peut avoir un très grand nombre d'états car il s'agit d'**un compteur** dont la valeur initiale peut être assimilée au nombre de ressources disponibles. **Un sémaphore ne peut jamais devenir négatif**.
- Un **sémaphore binaire**, comme un mutex, n'a que **deux états** : 0=verrouillé (ou occupé) ou 1=déverrouillé (ou libre).
- Un **sémaphore bloquant** est un sémpahore de synchronisation qui est initialisé à 0=verrouillé (ou occupé).
- Trois opérations sont associées à un sémaphore : `Init` pour initialiser la valeur du sémaphore, `P` pour l'acquisition (*Proberen*, tester ou P(uis-je)) et `V` pour la libération (*Verhogen*, incrémenter ou V(as-y)).

Mécanisme de synchronisation : la variable condition

La coordination de l'éxécution entre processus légers est maintenue par des mécanismes de synchronisation.

Il existe deux principaux mécanismes de synchronisation de processus pour les *threads* : la **variable condition** et la barrière.

- Les **variables conditions** permettent de suspendre un fil d'exécution (*thread*) tant que des données partagées n'ont pas atteint un certain état.
- Deux opérations sont disponibles : `wait`, qui bloque le processus léger tant que la condition est fausse et `signal` qui prévient les *threads* bloqués que la condition est vraie.

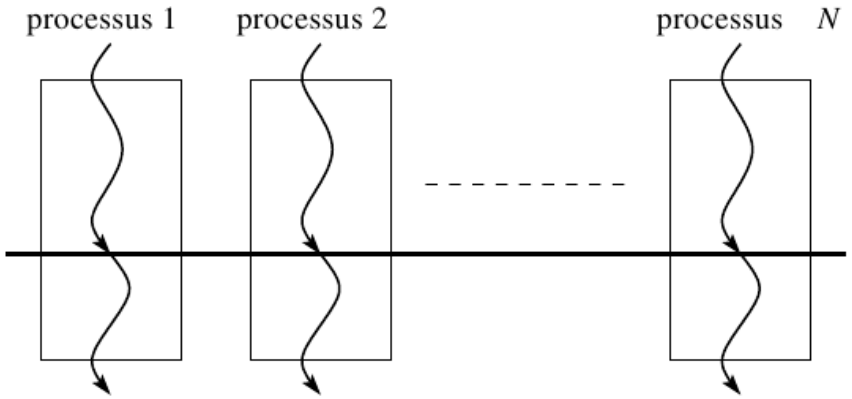


Mécanisme de synchronisation : la barrière

La coordination de l'éxécution entre processus légers est maintenue par des mécanismes de synchronisation.

Il existe deux principaux mécanismes de synchronisation de processus pour les *threads* : la variable condition et la **barrière**.

- Une **barrière de synchronisation** (ou mécanisme de rendez-vous) permet de **garantir qu'un certain nombre de tâches ait passé un point spécifique**. Ainsi, chaque tâche qui arrivera sur cette barrière devra attendre jusqu'à ce que le nombre spécifié de tâches soient arrivées à cette barrière.



Quelques problèmes connus

Les mécanismes de synchronisation peuvent conduire aux problèmes suivants :

- **Interblocage** (*deadlocks*) : Le phénomène d'interblocage est le problème le plus courant. L'interblocage se produit lorsque deux *threads* concurrents s'attendent mutuellement. Les *threads* bloqués dans cet état le sont définitivement.

```
TâcheA :
  Obtenir M1
  Obtenir M2
  Action nécessitant les deux verrous
  Rendre M2
  Rendre M1
```

```
TâcheB :
  Obtenir M2
  Obtenir M1
  Action nécessitant les deux verrous
  Rendre M1
  Rendre M2
```

- **Famine** (*starvation*) : Un processus léger ne pouvant jamais accéder à un verrou se trouve dans une situation de famine. Cette situation se produit, lorsqu'un processus léger, prêt à être exécuté, est toujours devancé par un autre processus léger plus prioritaire.
- **Endormissement** (*dormancy*) : cas d'un processus léger suspendu qui n'est jamais réveillé.

Synthèse

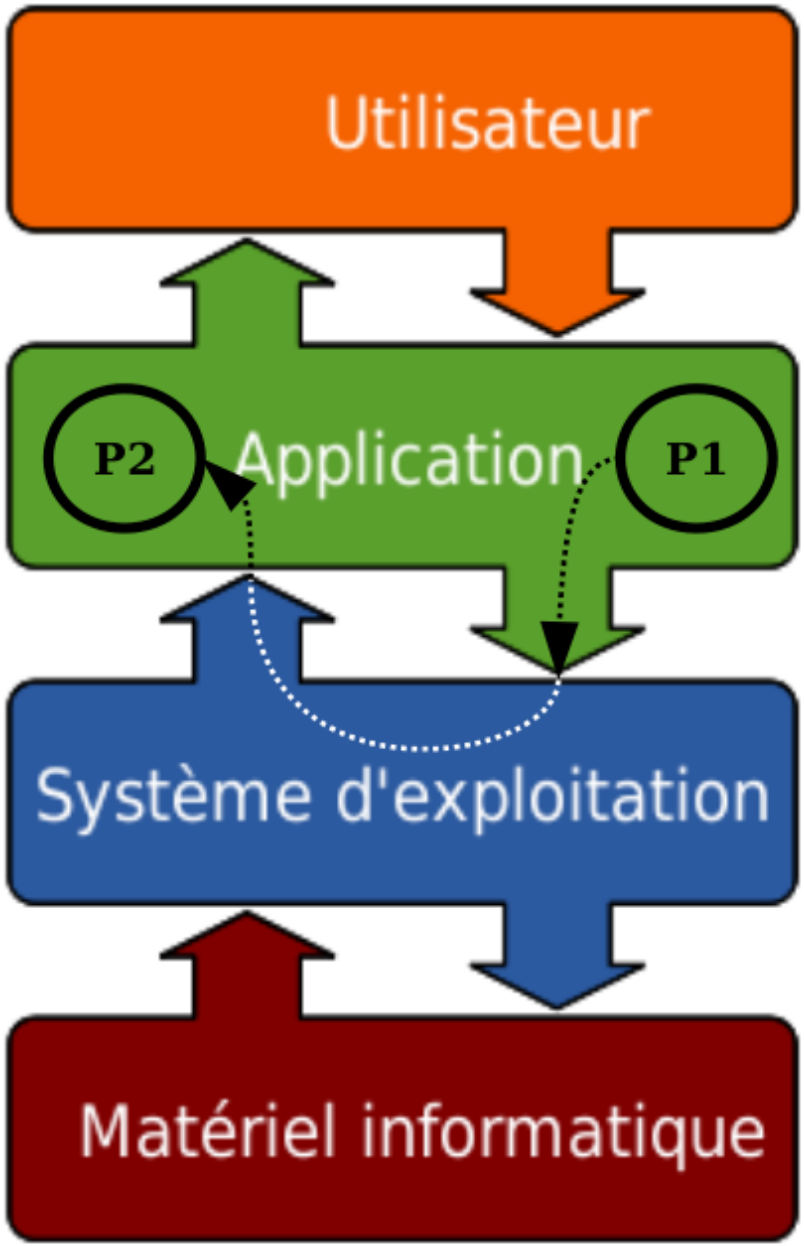
- Section critique : section de code où jamais plus d'une tâche ne peut être active
- Exclusion mutuelle : éviter que des ressources partagées ne soient utilisées en même temps par plusieurs tâches
- Chien de garde (*watchdog*) : tâche de fond assurant la protection contre le blocage de tâches
- Mutex : technique permettant de gérer un accès exclusif à des ressources partagées
- Sémaphore : variable compteur permettant de restreindre l'accès à des ressources partagées

IPC

Les **communications inter-processus** (*Inter-Process Communication* ou **IPC**) regroupent un ensemble de mécanismes permettant à des processus concurrents (ou distants) de communiquer. Ces mécanismes peuvent être classés en trois catégories :

- les outils permettant aux processus de **s'échanger des données**
- les outils permettant de **synchroniser les processus**, notamment pour gérer le principe de section critique
- les outils offrant directement les caractéristiques des deux premiers (échanger des données et synchroniser des processus)

Principe


Utilisateur
P2
Application
P1
Système d'exploitation
Matériel informatique

Échange de données

- Les **fichiers** peuvent être utilisés pour échanger des informations entre deux, ou plusieurs processus. Et par extension de la notion de fichiers, les **bases de données** peuvent aussi être utilisées pour échanger des informations entre deux, ou plusieurs processus.
- La **mémoire** (principale) d'un système peut aussi être utilisée pour des échanges de données. Suivant le type de processus, les outils utilisés ne sont pas les mêmes :
  - Dans le cas des processus lourds, l'espace mémoire du processus n'est pas partagé. On utilise alors des mécanismes de **mémoire partagée**, comme les **segments de mémoire partagée** pour Unix.
  - Dans le cas des processus légers (*threads*) l'espace mémoire est partagé, la mémoire peut donc être utilisée directement.
- Quelle que soit la méthode utilisée pour partager les données, ce type de communication pose le problème des **sections critiques**.

Synchronisation

Les mécanismes de synchronisation sont utilisés pour résoudre les problèmes de **sections critiques** et plus généralement pour bloquer et débloquer des processus suivant certaines conditions :

- Les **verrous** permettent de bloquer tout ou une partie d'un fichier
- Les **sémaphores** (et les **mutex**) sont un mécanisme plus général, ils ne sont pas associés à un type particulier de ressource et permettent de limiter l'accès concurrent à une section critique à un certain nombre de processus
- Les **signaux** (ou les **évènements**) permettent aux processus de communiquer entre eux : réveiller, arrêter ou avertir un processus d'un événement

L'utilisation des mécanismes de synchronisation est difficile et peut entraîner des **problèmes d'interblocage** (tous les processus sont bloqués).

Échange de données et synchronisation

Ces outils regroupent les possibilités des deux autres et sont souvent plus simples d'utilisation.

- L'idée est de communiquer en utilisant le principe des **files** (notion de boite aux lettres), les processus voulant envoyer des informations (**messages**) les placent dans la file ; ceux voulant les recevoir les récupèrent dans cette même file. Les opérations d'écriture et de lecture dans la file sont bloquantes et permettent donc la synchronisation.
- Ce principe est utilisé par :
  - les **files d'attente de message (*message queue*)** sous Unix,
  - les ***sockets* Unix ou Internet**,
  - les **tubes (nommés ou non)**, et
  - la transmission de **messages (*Message Passing*)** (DCOM, CORBA, SOAP, ...)

IPC Système V (Unix) / POSIX

Les IPC (Système V ou POSIX) recouvrent 3 mécanismes de communication entre processus (*Inter Processus Communication*) :

- Les **files de messages** (*message queue*), dans lesquelles un processus peut glisser des données ou en extraire. Les messages étant typés, il est possible de les lire dans un ordre différent de celui d'insertion, bien que par défaut la lecture se fasse suivant le principe de la file d'attente.
- Les **segments de mémoire partagée** (*shared memory*), qui sont accessibles simultanément par deux processus ou plus, avec éventuellement des restrictions telles que la lecture seule.
- Les **sémaphores**, qui permettent de synchroniser l'accès à des ressources partagées.

Les IPC permettent de faire communiquer deux processus d'une manière asynchrone, à l'inverse des tubes.

Files de messages

- Les **files de messages** sont des **listes chaînées** gérées par le noyau dans lesquelles un processus peut déposer des données (**messages**) ou en extraire. Elle correspond au concept de **boîte aux lettres**.
- Un **message** est une structure comportant un nombre entier (le type du message) et une suite d'octets de longueur arbitraire, représentant les données proprement dites.
- Les messages étant typés, il est possible de les lire dans un ordre différent de celui d'insertion. Le processus récepteur peut choisir de se mettre en attente soit sur le premier message disponible, soit sur le premier message d'un type donné.

Files de messages (suite)

Les **avantages** principaux de la file de message (par rapport aux tubes et aux tubes nommés) sont :

- un processus peut émettre des messages même si aucun processus n'est prêt à les recevoir
- les messages déposés sont conservés, même après la mort de l'émetteur, jusqu'à leur consommation ou la destruction de la file.

Le principal **inconvénient** de ce mécanisme est la limite de la taille des messages (8192 octets sous Linux) ainsi que celle de la file (16384 octets sous Linux).

Segment de mémoire partagée

Les processus peuvent avoir besoin de partager de l'information. Le système Unix fournit à cet effet un ensemble de routines permettant de créer et de gérer un segment de mémoire partagée (*shared memory*).

- Un **segment de mémoire partagée** est identifié de manière externe par un **nom** qui permet à tout processus possédant les droits, d'accéder à ce segment. Lors de la création ou de l'accès à un segment mémoire, un **numéro interne** est fourni par le système.
- Parmi les mécanismes de communication entre processus, l'utilisation de segments de mémoire partagée est la technique la plus rapide, car il n'y a pas de copie des données transmises. Ce procédé de communication est donc parfaitement adapté au partage de gros volumes de données entre processus distincts.

Mise en oeuvre d'un segment de mémoire partagée

- Si deux processus écrivent et lisent "en même temps" dans une même zone de mémoire partagée, il ne sera pas possible de savoir ce qui est réellement pris en compte. Le **segment de mémoire partagée** est considéré comme une **section critique**.
- D'où l'obligation d'utiliser des **sémaphores** pour ne donner l'accès à cette zone qu'à un processus à la fois.

API IPC System V

Les IPC **System V** sont une ancienne API (orientée UNIX).

	Files de messages	Mémoire partagée	Sémaphores
Headers	sys/types.h sys/ipc.h sys/msg.h	sys/types.h sys/ipc.h sys/shm.h	sys/types.h sys/ipc.h sys/sem.h
Création / Ouverture	msgget()	shmget()	semget()
Contrôle	msgctl()	shmctl()	semctl()
Opérations	msgsnd() msgrcv()	shmat() shmdt()	semop()
Structures associées	msqid_ds	shmid_ds	semid_ds

Deux opérations sont disponibles à partir du *shell* :

- `ipcs` : consultation des objets de communication.
- `ipcrm` : suppression des objets de communication.

IPC POSIX

Les IPC **POSIX** sont une API plus récente et normalisée.

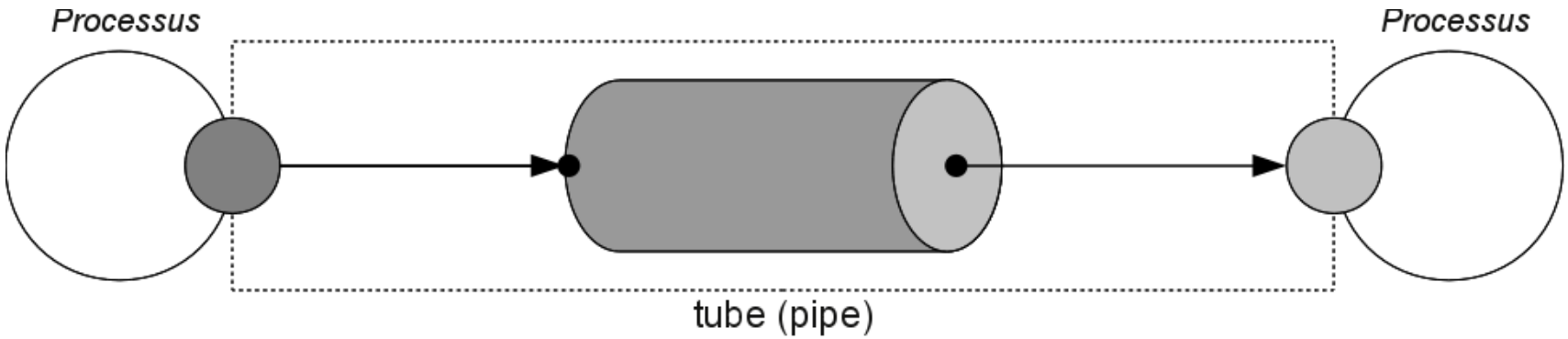
- les appels systèmes pour les files de messages sont : `mq_open()`, `mq_send()`, `mq_receive()`, `mq_close()` et `mq_unlink()`
- les appels systèmes pour les segments de mémoire partagé sont : `shm_open()`, `mmap()`, `munmap()`, `close()` et `shm_unlink()`
- les appels systèmes pour les sémaphores sont : `sem_open()`, `sem_post()`, `sem_wait()`, `sem_close()` et `sem_unlink()`
- Plus de détails : man mq_overview, man sem_overview, man shm_overview

Les IPC POSIX fournissent une interface bien mieux conçue (portable et "*thread safe*") que celles de System V. D'un autre coté, les IPC POSIX sont moins largement disponibles (particulièrement sur d'anciens systèmes) que ceux de System V.

Les tubes

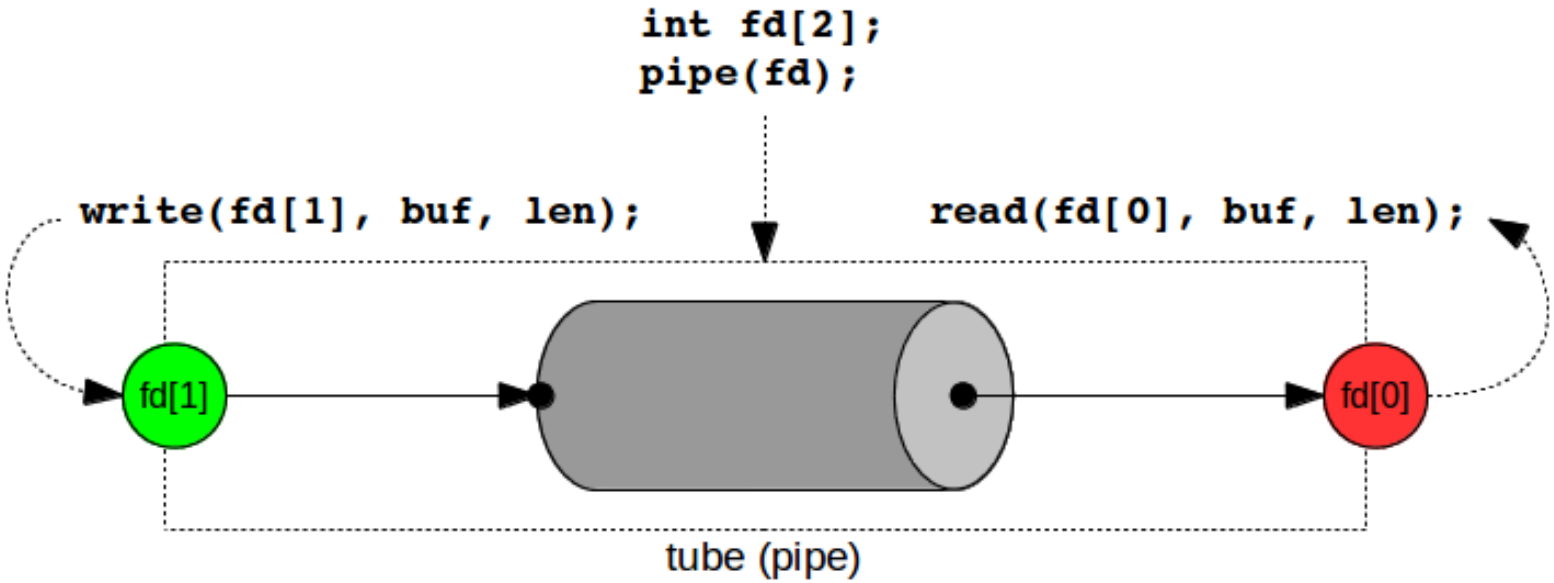
Les **tubes** (*pipe*) sont un mécanisme de communication entre **processus résidants sur une même machine**. On peut distinguer 2 catégories :

- les **tubes anonymes** (volatiles) : ils concernent des processus issus de la même application
- les **tubes nommés** (persistants) : ils concernent des processus totalement indépendants



Principe

Un **tube** peut être vu comme un **tuyau** dans lequel un processus, à une extrémité, produit des informations, tandis qu'à l'autre extrémité un autre processus les consomme.



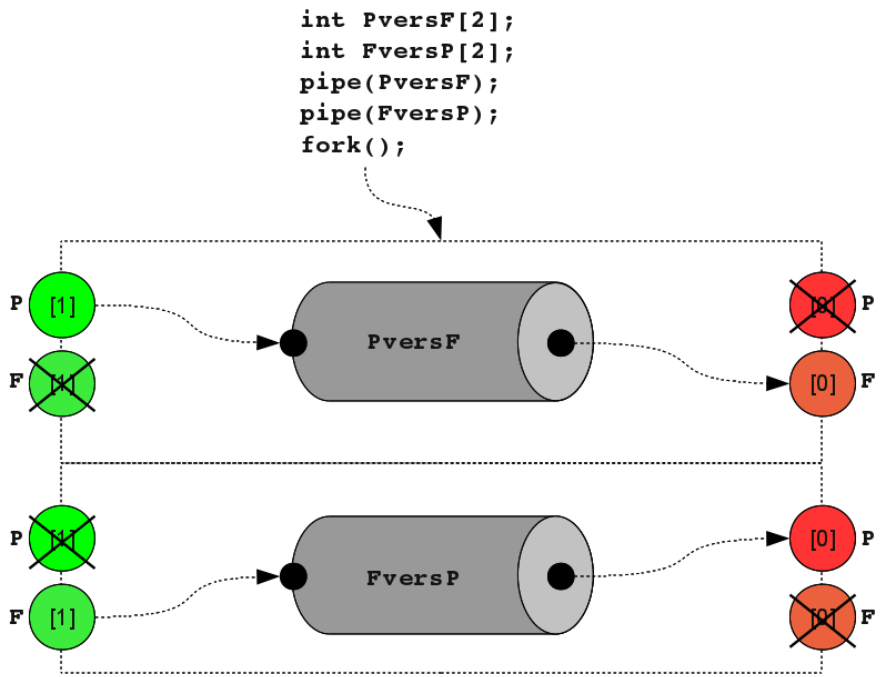
Un tube est créé par l'appel système `pipe(int descripteur[2])` qui remplit le tableau `descripteur` avec les descripteurs de chacune des extrémités du tube (l'entrée en lecture seule et la sortie en écriture seule).

Caractéristiques

- intra-machine
- communication unidirectionnelle
- communication synchrone
- communication en mode flux (*stream*)
- remplissage et vidage en mode FIFO
- limité en taille

Communication bidirectionnelle

Les tubes étant des systèmes de communication unidirectionnels, ils faut créer **deux tubes** et les employer dans le sens opposé si l'on souhaite réaliser une communication bidirectionnelle entre 2 processus.



La création de ces deux tubes est sous la responsabilité du processus père. Ensuite, la primive `fork()` réalisant une copie des données dans le processus fils, il suffit pour le père et le fils de fermer les extrémités de tubes qu'ils n'utilisent pas.

Les tubes nommés

Un **tube nommé** est simplement un noeud dans le système de fichiers. Le concept de tube a été étendu pour disposer d'un **nom** dans ce dernier. Ce moyen de communication, disposant d'une représentation dans le système de fichier, peut être utilisé par des processus indépendants.

- La création d'un tube nommé se fait à l'aide de la fonction `mkfifo()`.
- La suppression d'un tube nommé s'effectue avec `unlink()`.
- Une fois le noeud créé, on peut l'ouvrir avec `open()` avec les restrictions dues au mode d'accès.
- Si le tube est ouvert en lecture, `open()` est bloquante tant qu'un autre processus ne l'a pas ouvert en écriture. Symétriquement, une ouverture en écriture est bloquante jusqu'à ce que le tube soit ouvert en lecture.
- Il est possible d'utiliser la fonction `fdopen()` qui retourne un flux associé au descripteur passé en paramètre. On peut ensuite utiliser les

Les signaux

Un **signal** peut être imaginé comme une sorte d'impulsion qui oblige le processus cible à prendre immédiatement une mesure spécifique.

Le rôle des signaux est de permettre aux processus de communiquer. Ils peuvent par exemple :

- réveiller un processus
- arrêter un processus
- avertir un processus d'un événement

L'utilisation des signaux sous Linux est décrite dans : `man 7 signal`

Envoi et réception d'un signal

- La commande `kill` permet d'envoyer un signal à un processus ou à un groupe de processus
- L'appel système `kill()` peut être utilisé pour envoyer n'importe quel signal à n'importe quel processus ou groupe de processus

À la réception d'un signal, le processus peut :

- l'ignorer
- déclencher l'action prévue par défaut sur le système (souvent la mort du processus)
- déclencher un traitement spécial appelé *handler* (gestionnaire du signal)

Interception d'un signal

- La primitive `signal()` permet d'associer un traitement à la réception d'un signal d'interruption. Une autre primitive, `sigaction()`, permet de faire la même chose et présente l'avantage de définir précisément le comportement désiré et de ne pas poser de problème de compatibilité.
- `signal()` installe le gestionnaire *handler* pour le signal *signum*. *handler* peut être `SIG_IGN` (ignoré), `SIG_DFL` (défaut) ou l'**adresse d'une fonction** définie par le programmeur (un « gestionnaire de signal »).

L'attente d'un signal démontre tout l'intérêt du multi-tâche. En effet, au lieu de consommer des ressources CPU inutilement en testant la présence d'un événement dans une boucle `while`, on place le processus en sommeil (*sleep*) et le processeur est mis alors à la disposition d'autres tâches.

Quelques signaux

- La commande `kill -l` liste l'ensemble des signaux disponibles
- Le fichier *header* `signum.h` déclare la liste des signaux :
  `find /usr -name signum.h` puis
  `grep SIG /usr/include/x86_64-linux-gnu/bits/signum.h`

```
#define SIGINT 2 /* Interrupt = Ctrl-C (ANSI) */...
#define SIGKILL 9 /* Kill, unblockable (POSIX) */...
#define SIGUSR1 10 /* User-defined signal 1 POSIX */...
#define SIGUSR2 12 /* User-defined signal 2 POSIX */...
#define _NSIG 65 /* Biggest signal number + 1 */
```

IPC

L'API Win32 de Microsoft propose différents moyens pour faire **communiquer plusieurs processus ou *threads***.

Les mécanismes fournis sont décrits dans la MSDN dans l'article « *Interprocess Communications* » : http ://msdn.microsoft.com/en-us/library/aa365574(v=VS.85).aspx.

Cela couvre l'utilisation du presse-papier (*clipboard*) jusqu'au *socket*.

L'API Win32 de Microsoft propose différents moyens pour **coordonner l'exécution de plusieurs *threads***. Les mécanismes fournis sont :

- les exclusions mutuelles
- les sections critiques
- les sémaphores

Fichier mappé

La technique officielle préconisée par Microsoft pour **partager de la mémoire** est d'utiliser les différentes fonctions pour accéder à un **fichier mappé (*FileMapping*)**.

- Un des processus ou *threads* doit créer en espace de mémoire partagée avec un fichier mappé en utilisant `CreateFileMapping()`.
- Les autres processus ou *threads* peuvent accéder à la mémoire partagée en l'ouvrant avec la fonction `OpenFileMapping()`.
- Pour avoir un pointeur utilisable pour accéder à une zone de la mémoire, il faut utiliser `MapViewOfFile()`. Après utilisation, il faut appeler `UnmapViewOfFile()` pour détacher le *handle*.
- Pour libérer la mémoire, il suffit d'appeler `CloseHandle()` sur le *handle* obtenu.

L'accès simultanée à la mémoire partagée doit être sécurisé par l'utilisation d'un *mutex* par exemple.

Mutex

Le **mutex** est de type **HANDLE**, comme tout objet ressource dans l'API Win32. Les fonctions utilisables sont :

- Création du *mutex* : `CreateMutex()`
- Opération V(), libération ou déverrouillage du *mutex* : `ReleaseMutex()`
- Opération P(), prise ou verrouillage du *mutex* : `WaitForSingleObject()`

L'objet **section critique** fournit une synchronisation similaire au *mutex* sauf que ces objets doivent être utilisées par les *threads* d'un même processus. Ce mécanisme est plus performant que les *mutex* et il est plus simple à utiliser.

Section critique

Typiquement, il suffit de déclarer une variable de type `CRITICAL_SECTION`, de l'initialiser grâce à la fonction `InitializeCriticalSection()` ou `InitializeCriticalSectionAndSpinCount()`. Les fonctions utilisables sont :

- Le *thread* utilise la fonction `EnterCriticalSection()` ou `TryEnterCriticalSection()` avant d'entrer dans la section critique.
- Le *thread* utilise la fonction `LeaveCriticalSection()` pour rendre le processeur à la fin de la section critique.
- Chaque *thread* du processus peut utiliser la fonction `DeleteCriticalSection()` pour libérer les ressources système qui ont été allouées quand la section critique a été initialisé. Après son appel, aucune fonction de synchronisation ne peut plus être appelée.

Sémaphore

Sous Windows, un **sémaphore** est un **compteur qui peut prendre des valeurs entre 0 et une valeur maximale** définie lors de la création du sémaphore. Le sémaphore est de type **HANDLE**. Les fonctions utilisables sont :

- La fonction `CreateSemaphore()` crée l'objet sémaphore.
- La fonction `OpenSemaphore()` ouvre un objet sémaphore déjà créé par un autre thread ou processus.
- Opération V() : la fonction `ReleaseSemaphore()` libère un jeton.
- Opération P() : la fonction `WaitForSingleObject()` permet de prendre un sémaphore.