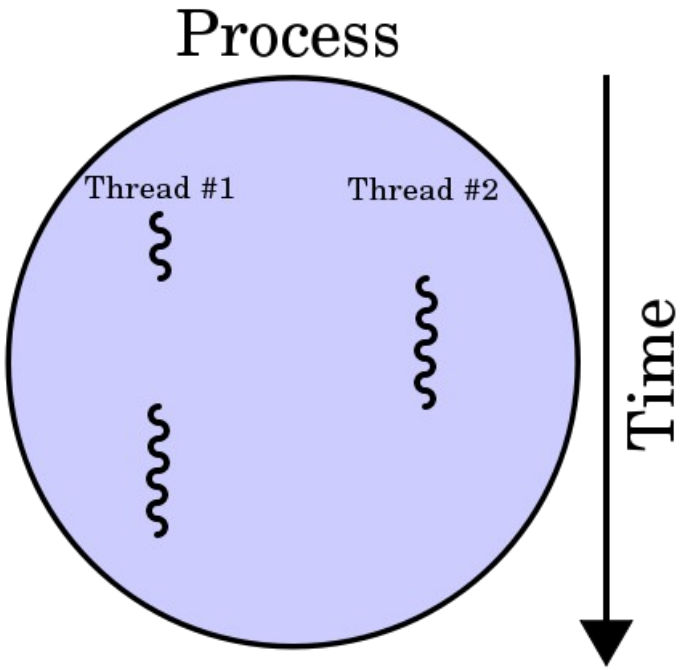
# Programmation Système : les threads

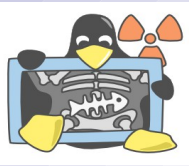




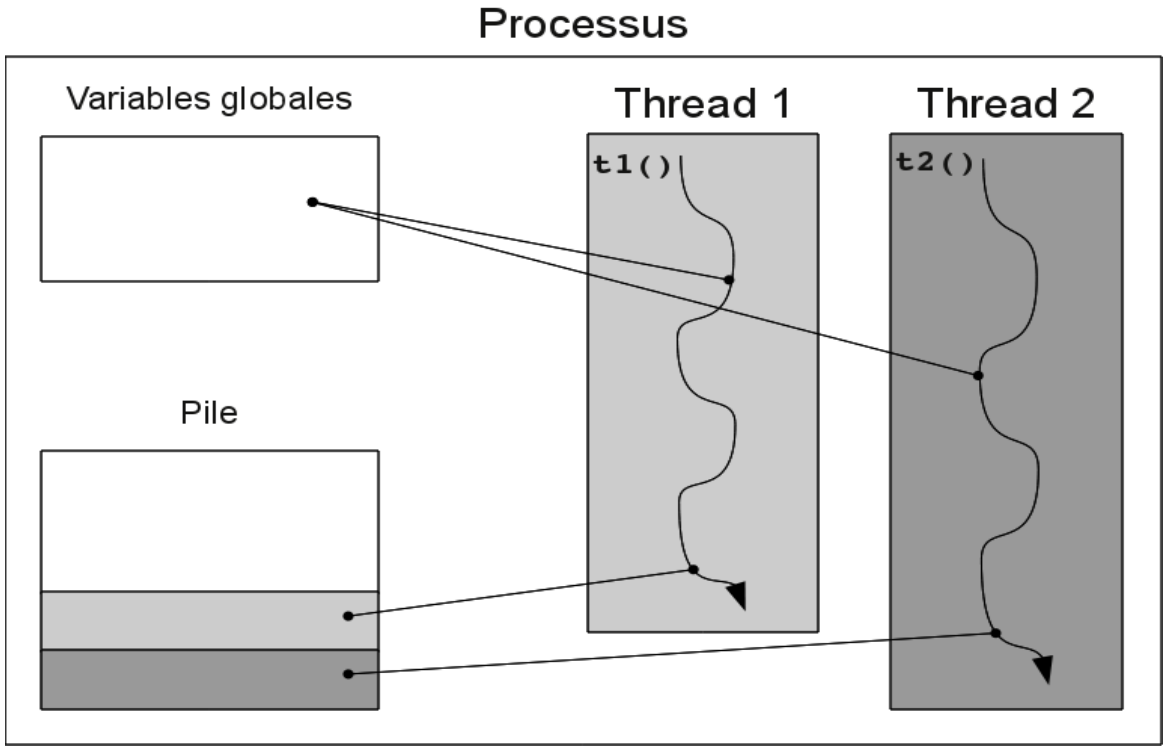
# Exemple introductif

- Problème : pendant le déplacement de l'objet dans la fenêtre, l'IHM doit rester fonctionnelle ...
- Mais le déplacement de l'objet sur la fenêtre prend du temps. Pendant cela, toute action sur l'un des boutons est inopérante.
- Solution : pour rendre l'interface réactive il faut déléguer le travail de déplacement de l'objet à une tâche.

# Définitions



- Thread = fil en anglais, un thread est un fil d'exécution, plusieurs traductions : activité (tâche), fil d'exécution, *lightweight process* (lwp) ou processus léger (par opposition au processus lourd créé par fork)
- Les threads permettent de dérouler plusieurs suites d'instructions, en PARALLELE, à l'intérieur du même processus. Un thread exécute une fonction.



# Avantages - Inconvénients

- **Avantages :**
  - **Multi-tâche moins coûteux** : 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

  - **Communication entre threads plus rapide et plus efficace** : grâce au partage de certaines ressources entre threads, IPC (*Inter Processus Communication*) inutile pour les threads

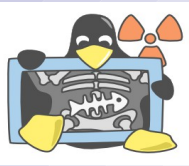
- **Inconvénients :**
  - **Programmation utilisant des threads est toutefois plus difficile** : obligation de mettre en place des **mécanismes de synchronisation** , risques élevés d'interblocage, de famine, d'endormissement

# Support

- Les systèmes d'exploitation mettent en œuvre 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, ACE, …) 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.

***Voir Annexe1***

# Multi-tâche



- Besoin : on désire paralléliser des traitement indépendants : les fonctions f() et g() travaillent sur deux parties d'un tableau. Il existe deux solutions : les processus lourds ou les processus légers.

```
int Tab[100];
void main (void)
{
   ...
   f();
   g();
   ...
}
```

**fork** : deux processus indépendants exécutent l'un f, l'autre g.
Inconvénient : il y a une duplication des zones de mémoire, il faut passer par un segment de mémoire partagée. Dans ce cas la séparation des espaces d'adressage est pénalisante.

```
int Tab[100];

int pid_f, pid_g;
...
pid_f = fork();
if (pid_f == 0) { ... f(); ... }
pid_g = fork();
if (pid_g == 0) { ... g(); ... }
...
```

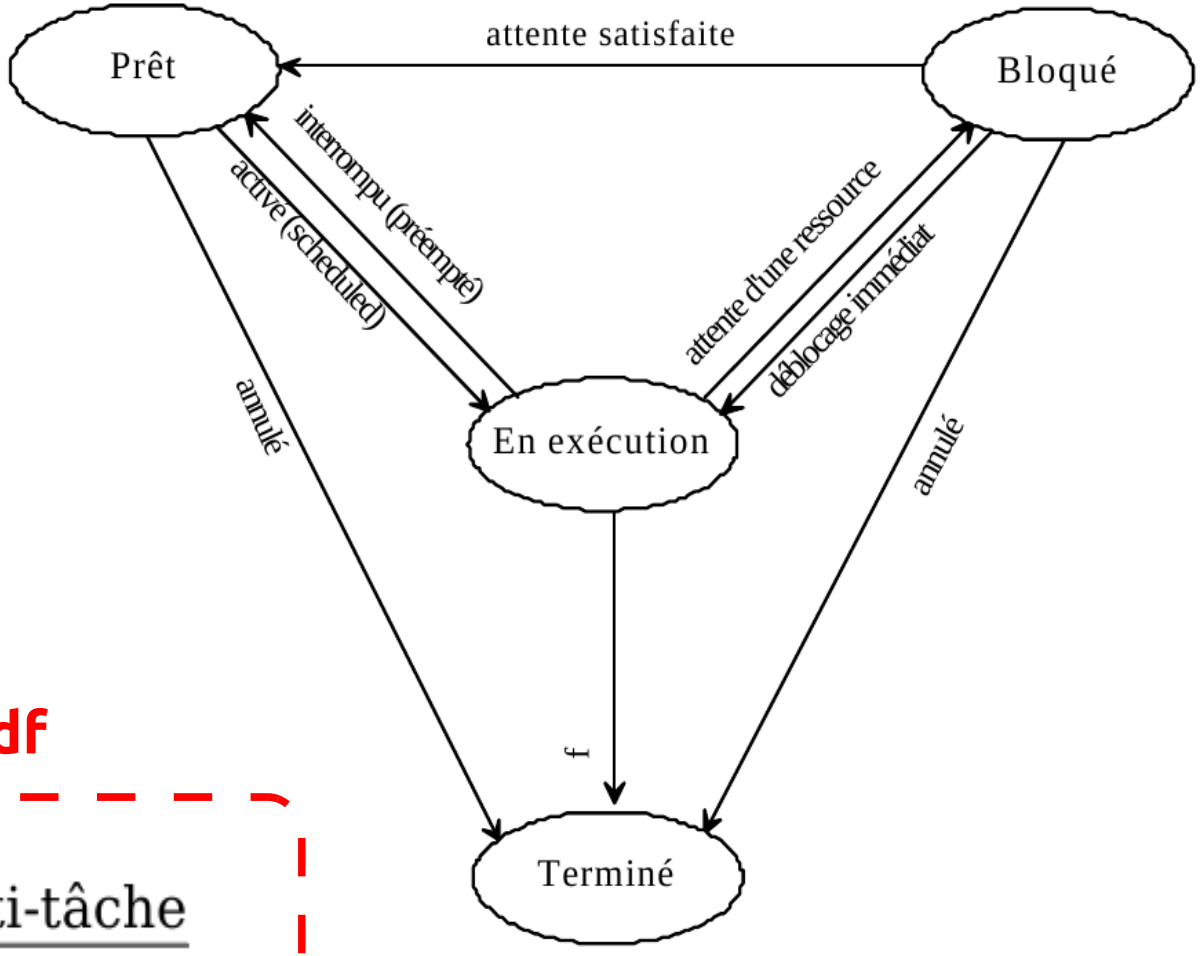
**pthread_create** : deux threads exécutent l'un f, l'autre g.
Avantage : transformation simple du code, parce que, ici, il n'y a pas à gérer de synchronisation entre les deux tâches.

```
int Tab[100];

pthread_t t_f, t_g;
...
pthread_create(&t_f, NULL, f, NULL);
pthread_create(&t_g, NULL, g, NULL);
…
```

# États d'un thread

- Le schéma suivant présente les états d'un processus léger du standard POSIX :

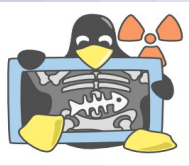




**tp-sys-threads.pdf**


Séquence 1 : multi-tâche

# La synchronisation



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.

# Quelques définitions

- **Section Critique** : C'est une partie de code telle que deux threads ne peuvent s'y trouver au même instant.

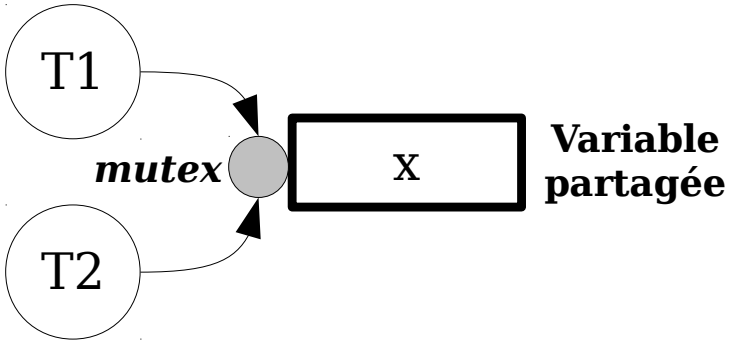
- **Exclusion mutuelle** : Une ressource est en exclusion mutuelle si seul un thread peut utiliser la ressource à un instant donné.

- **Chien de garde (*watchdog*)** : Un chien de garde est une technique logicielle 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. Il s'agit en général d'un compteur qui est régulièrement remis à zéro. Si le compteur dépasse une valeur donnée (*timeout*) alors on procède à un redémarrage (reset) du système. Si une routine entre dans une boucle infinie, le compteur du chien de garde ne sera plus remis à zéro et un reset est ordonné.

# Mécanismes de synchronisation : mutex

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 : **mutex** et variable condition.

- 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).





**tp-sys-threads.pdf**


Séquence 2 : synchronisation de donnée

# Réentrance et thread-safe

- La réentrance est la propriété pour une fonction d'être utilisable simultanément par plusieurs tâches utilisatrices. La réentrance permet d'éviter la duplication en mémoire vive d'un programme utilisé simultanément par plusieurs utilisateurs.
- Dans la plupart des cas, pour transformer une fonction non réentrante en une fonction réentrante, on doit modifier son interface externe pour que toutes les données soient fournies par l'appelant de la fonction (remplacement des éventuelles variables globales utilisées dans la fonction par des variables locales).
- Pour rendre thread-safe une fonction non thread-safe, un changement d'implémentation seul suffit. De manière usuelle, l'ajout d'un point de synchronisation tel qu'une section critique ou un sémaphore est utilisé pour protéger l'accès a une ressource partagée d'un accès concurrent d'une autre tâche/thread.
- *Remarque : Une synchronisation peut être mise en place au niveau des processus légers mais pas au niveau de la bibliothèque car il n'y a pas de modifications possibles des codes sources.*

# Mécanismes de synchronisation : condition

Avec les threads, il existe deux principaux mécanismes de synchronisation : mutex et **variable condition**.

- Les **variables conditions** permettent de suspendre un fil d'exécution tant que des données partagées n'ont pas atteint un certain état. Une variable condition est souvent utilisée avec un mutex. 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.


**tp-sys-threads.pdf**


Séquence 3 : synchronisation de tâche

# 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.

- **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.

# Modèle de programmation

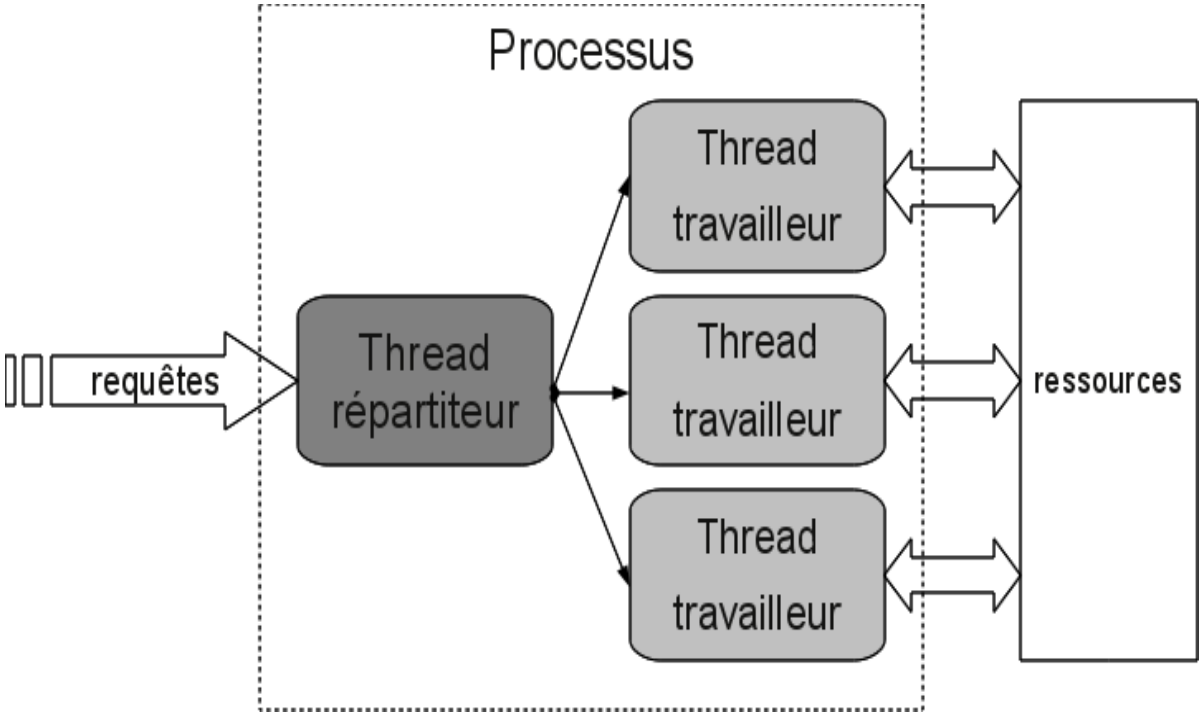
- Chaque programme comportant des processus légers est différent.

- Cependant, certains modèles communs sont apparus. Ces modèles permettent de définir comment une application attribue une activité à chaque processus léger et comment ces processus légers communiquent entre eux.

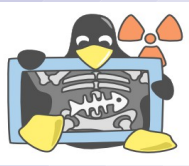
- On distingue généralement 3 modèles :
  - Modèle répartiteur/travailleurs ou maître/esclaves
  - Modèle en groupe
  - Modèle en pipeline

# Modèle répartiteur/travailleurs ou maître/esclaves

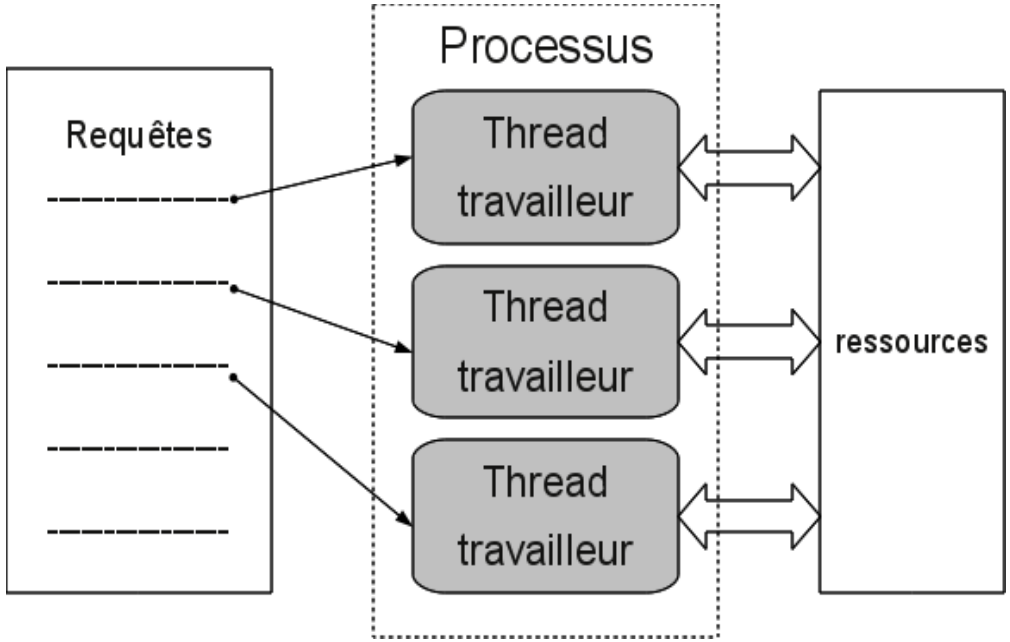
- Un processus léger, appelé le répartiteur (ou le maître), reçoit des requêtes pour tout le programme. En fonction de la requête reçue, le répartiteur attribue l'activité à un ou plusieurs processus légers travailleurs (ou esclaves).
- Ce modèle convient aux serveurs de bases de données, serveurs de fichiers, gestionnaires de fenêtres (window managers) et équivalents ...



# Modèle en groupe

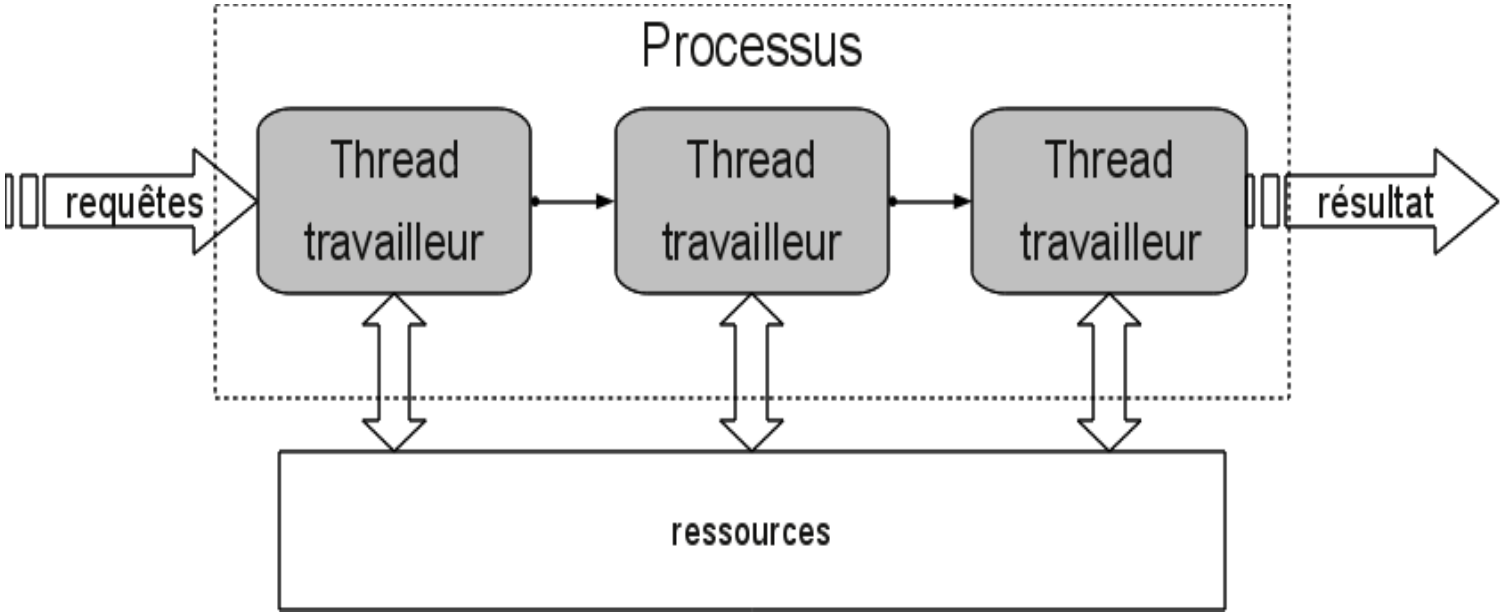


- Chaque processus léger réalise les traitements concurremment sans être piloté par un répartiteur (les traitements à effectuer sont déjà connus). Chaque processus léger traite ses propres requêtes (mais il peut être spécialisé pour un certain type de travail).
- Le modèle en groupe convient aux applications ayant des traitements définis à réaliser. Par exemple un moteur de recherche dans une base de données.



# Modèle en pipeline

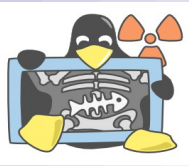
- L'exécution d'une requête est réalisée par plusieurs processus légers exécutant une partie de la requête en série. Les traitements sont effectués par étape du premier processus léger au dernier. Le premier processus engendre des données qu'il passe au suivant.
- Le modèle en pipeline est utilisé pour les traitements d'images, traitements de textes, mais aussi pour les applications pouvant être décomposées en étapes.



# Les threads en C++

- L'utilisation des threads POSIX est tout à fait possible en C++. Toutefois, il sera conseillé d'utiliser un framework C++ (Qt, Builder, commoncpp, ACE, ...).
- Adresse d'une méthode d'une classe :
  - Problème C++ : pouvoir passer l'adresse d'une méthode à la fonction pthread_create() comme on le faisait en C pour une fonction. Pour cela, il faut absolument que la méthode soit déclarée comme **statique**. Cette méthode static sera donc disponible en dehors de toute instance de la classe mais par contre elle ne pourra pas accéder aux attributs de sa classe en dehors des autres membres statiques.
- Squelette d'une classe Thread :
  - La classe Thread sera une **classe abstraite** qui contient une **méthode virtuelle pure** execute() ou run(). Cette classe possèdera aussi une **méthode statique** start() ou Resume(). C'est cette méthode statique qui aura pour rôle d'appeler la méthode execute() ou run(). Pour utiliser cette classe, il faudra donc créer une **classe dérivée** en écrivant le code de la méthode execute() ou run() : c'est ce code qui représentera alors votre thread.

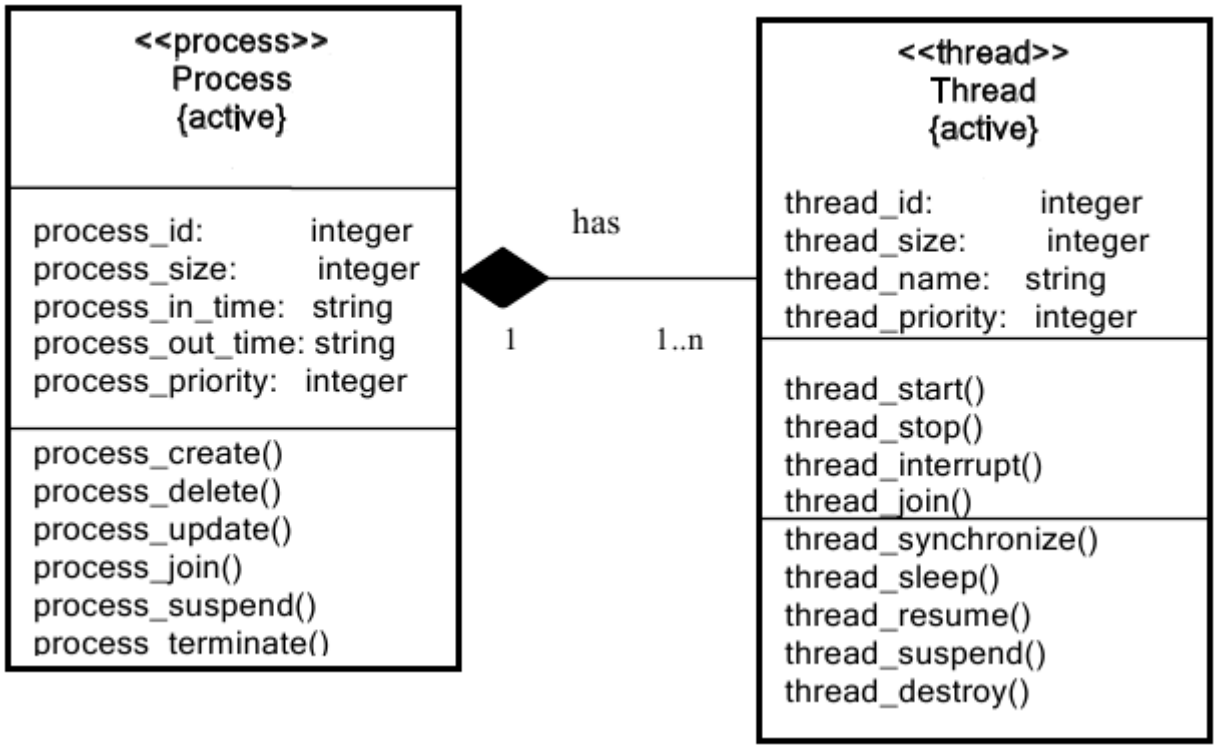
# Les threads en UML



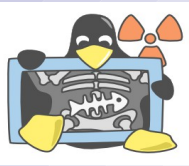
- Rappel : architecture logicielle d'une application = 5 vues (cas d'utilisations, conception, processus, implémentation et déploiement). Une vue est une description simplifiée d'un système observé d'un point de vue particulier.
- La vue des processus précise les threads et les processus qui forme les mécanismes de concurrence et de synchronisation du système. On met l'accent sur les classes actives qui représentent les threads et les processus. Elle montre : La décomposition du système en terme de processus (tâches), les interactions entre les processus (leur communication) et la synchronisation et la communication des activités parallèles (threads).
- **Classe active et objet actif**
  - UML fournit un repère visuel (bord en trait épais) qui permet de distinguer les éléments actifs (processus ou thread) des éléments passifs. Une instance d'une classe active sera nommée object actif. Chaque processus ou thread au sein d'un système définit alors un flot de contrôle distinct.

# Stéréotype

- UML définit deux stéréotypes standards qui s'appliquent aux classes actives :
  - **process** : spécifie un flot « lourd » qui peut s'exécuter en concurrence avec d'autres processus.
  - **thread** : spécifie un flot « léger » qui peut s'exécuter en concurrence avec d'autres threads à l'intérieur d'un même processus.



# Méthodologie



- Le découpage de l'application en threads (ou processus) apparaît lorsqu'on établit :
  - les diagrammes d'activités : activités concurrentes et/ou
  - les diagrammes d'états : états concurrents et/ou
  - les diagrammes de séquences : exécutions concurrentes de plusieurs objets

- Le besoin est toujours la parallélisation :
  - le système réalise plusieurs activités en même temps et/ou
  - un objet prend plusieurs états en même temps et/ou
  - une utilisation du système réalise plusieurs exécutions en même temps

- Le besoin de thread apparaît clairement pendant la phase d'analyse lorsqu'on distingue ce qui se produit de manière séquentielle de ce qui se produit de manière parallèle.

# Conclusion

- Il est conseillé avant de continuer de revoir :
  - Les définitions (threads, mutex, section critique, réentrance, ...)
  - Les exemples fournis pour les environnements Qt et Builder

- Il reste à voir entre autres :
  - L'ordonnancement des processus
  - La communication entre processus


**tp-sys-threads.pdf**

Bonus : les douches