Introduction à la programmation concurrente

(Embarqué, Temps Réel, Robotique)

Cours 1 : Quelques notions sur les activités parallèles

+ Exercices

Alexander Saidi

ECL - LIRIS

Janvier 2018

Plan

- La programmation parallèle par *Processus* et *Threads*
- Programmation parallèle **multi tâches** vs. **mono tache** :
  - Pourquoi utiliser les Processus / Threads ?
  - Un exemple pertinent : calcul de la valeur approchée de Pi (π)
- Dans quels cas utiliser la programmation parallèle ?
  - Des applications où il y a de la concurrence (e.g. schéma *Prod / Conso*)
  - Activités décomposables en *tâches* (quasi) indépendantes

  P. ex. en trait. Image, Réseaux de neurones, calcul de PI, Fib, MCL, ...

  ➛ Outil formel : l'algèbre des tâches et l'indépendance des activités.
- Liens avec l'Embarqué et avec le Temps Réel
- **Processus** vs. **Threads**

Un 1er ex. de tache décomposable : val. de π

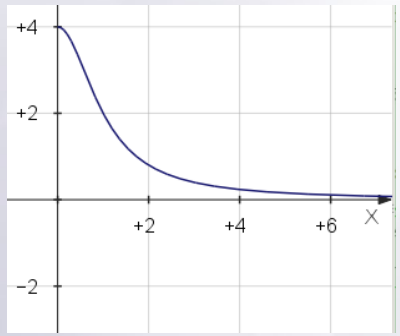
Méthode arc-tangente :

- On peut calculer une valeur approchée de PI par la méthode suivante :

$$\pi \approx \int_0^1 \frac{4}{1+x^2}\,dx \qquad \text{ou la version discrète} \qquad \pi \approx 1/n \sum_1^n \frac{4}{1+x_i^2}$$

où l'intervalle $[0,1]$ est divisé en n partitions (bâton) égales.

N.B. : pour que la somme des bâtons soit plus proche de l'aire sous la courbe, considérons le milieu des bâtons :



$$\sum_1^n \frac{4}{1+x_i^2} \approx \sum_{i=1}^n \frac{4}{1+(\frac{i-0.5}{n})^2} = \sum_{i=0}^{n-1} \frac{4}{1+(\frac{i+0.5}{n})^2}$$

➛ $(\frac{i-0.5}{n})$ ramène i dans $[0,1]$ (i.e. x_i)

☞ Tester et mesurer la différence en temps de calcul : 1 à 8 threads.

Un 1er ex. de tache décomposable : val. de π (suite)

Démarche :

- On décide de "faire faire" ce travail par k ouvriers
  - ➛ On dit *workers* (= Thread) en charge d'une tâche
- Chacun fera "sa part" (dans une somme locale) et on additionnera ces sommes pour obtenir une somme globale = aire.
- La fonction qui fixe la tâche de chaque thread est (doit l'être ici) identique.
- Dans d'autres situations, il peut y avoir des tâches différentes
  - ➛ E.g. construction d'une maison parallélisable pour certaines parties
- Nota Bene sur le découpage en intervalles (ici entrelacé)

☞ Soumission de thread vs. un simple appel de fonction

Un 1er ex. de tache décomposable : val. de π (suite)

```
Définir le nombre N de bâtons : soit 10^6
Définir le nombre k de threads : soit 5


Somme globale = 0.0 (le résultat des calculs)


La tache de chaque Thread No i=1..k :
    somme_locale = 0.0
    verser dans la somme_locale l'aire des  rectangles  du ième intervalle
    Avant de finir ,  verser la somme_locale dans la Somme_globale



Le travaille du chef (main) :
    Créer les k Threads et  assigner à chacun un  No (1..k)  et la tâche définie ci-dessus
    Les lancer (conséquence de la création)
    Attendre qu'ils finissent tous
    Récupérer et afficher la Somme_globale
```

☞ Toujours faire attention aux variables globales modifiées par les threads (voir + loin).

- ➛ Ressource unique + accès concurrent (en W/R) : PROTÉGEZ !
- ➛ Ex. (de mauvais goût mais efficace) : un slot W.C. ouvert à tout vent !

Un 1er ex. de tache décomposable : val. de π (suite)

```
#define NUM_RECTS 1000000
#define NUM_THREADS 5

double somme_globale=0.0f;                    // IMPORTANT de ne pas initialiser avec un simple '0'

void callcul_pi_partiel_arctang(int myNum){
     double somme_locale=0.0f;
     double Un_pas = 1.0 / NUM_RECTS, x;
     for (int i = myNum; i < NUM_RECTS; i += NUM_THREADS){
          x = (i + 0.5f) / NUM_RECTS;
          somme_locale += 4.0f / (1.0f + x*x);
     }
     somme_globale += somme_locale* Un_pas;
 }
 //-------------------------------------
int main() {
     std::thread Ids[NUM_THREADS];

     for(int k=0; k<NUM_THREADS; k++)
          Ids[k]=thread(callcul_pi_partiel_arctang, k);

     for(int k=0; k<NUM_THREADS; k++)     Ids[k].join();

     cout << "Valeur approchee de Pi (méthode basique 1) :  " <<  somme_globale << endl;
     return 0;
 }

// Trace : Valeur approchee de Pi (méthode basique 1) :  3.14159

// Compilation :
// g++-5 -std=c++14 calcul_parallele_pi_version-simple.cpp -lpthread -D_GLIBCXX_USE_NANOSLEEP
```

Un 1er ex. de tache décomposable : val. de π (suite)

Quelques remarques sur l'exemple précédent :

1. Les directives et inclusions (voir aussi la cmd. de compilation) :

```
#include <thread>
#include <iostream>
using namespace std;
```

2. La fonction *main* attend la fin de chaque thread avec **join**.
3. Chaque *thread* (5 ici) calcule sa "part" dans la variable `somme_locale` puis verse ce résultat dans la variable globale `somme_globale`.
4. Il y a un risque (faible mais non nul) que les threads se marchent sur les pieds lors la manipulation de `somme_globale`.
   ➛ Différentes solutions : variable **atomic**, *exclusion mutuelle*, ...
5. Possibilité en C++11/C++14 d'écrire les itérations plus simplement :

```
for(auto& tache : lds)        tache=thread(callcul_pi_partiel_arctang, i++);
for(auto& tache_bis : lds)    tache_bis.join();    // peut pas réutiliser 'tache'
```

6. Les calculs sont (env.) **5 fois plus rapides** qu'avec un seul thread.

Un 1er ex. de tache décomposable : val. de π (suite)

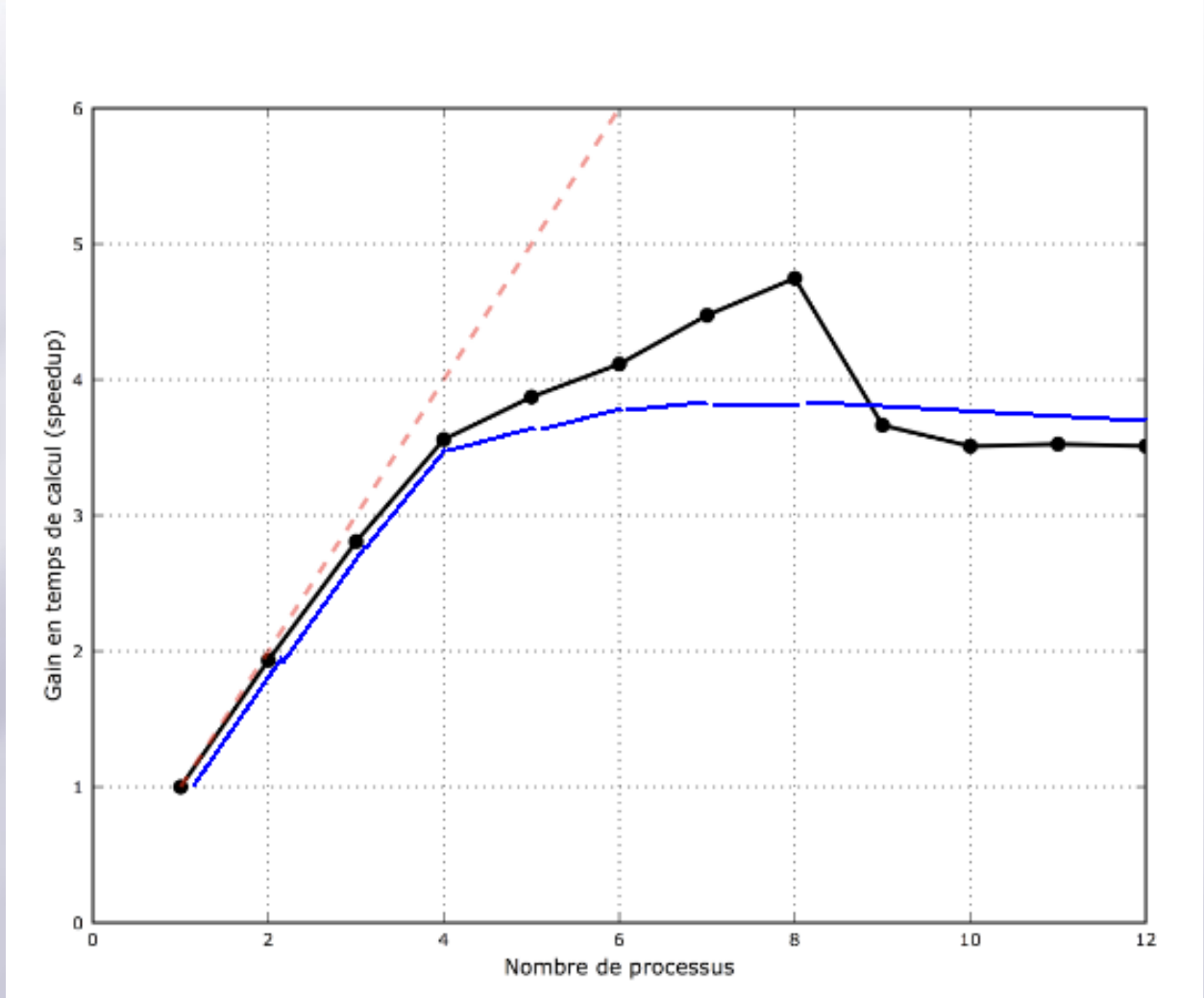
Quid du nombre de threads :

Différents temps de calcul en fonction du nombre de Threads :

```
int main() {
     for (NUM_THREADS=1; NUM_THREADS<9; NUM_THREADS++){
        int i=0,j;
        std::thread Ids[NUM_THREADS];
        auto start = chrono::high_resolution_clock::now();

        for(auto& tache : Ids)  tache=thread(callcul_pi_partiel_arctang, i++);
        for(auto& tache_bis : Ids) tache_bis.join();

        auto end = chrono::high_resolution_clock::now();
        auto diff = end - start;
        auto diff_sec = chrono::duration_cast<chrono::nanoseconds>(diff);
        cout << "Temps  pour " << NUM_THREADS << " threads = " << diff_sec.count() << " ns ";

        cout << " Pi calculée (basique 1) :  " <<  global_sum/NUM_THREADS << endl;
     }
     return 0;
 }
///// TRACE (statistique empirique)
Temps  pour 1 threads = 39618361 ns  Pi calculée (basique 1) :  3.14159
Temps  pour 2 threads = 17449046 ns  Pi calculée (basique 1) :  3.14159
Temps  pour 3 threads = 10578141 ns  Pi calculée (basique 1) :  3.14159
Temps  pour 4 threads =  7160884 ns  Pi calculée (basique 1) :  3.14159
Temps  pour 5 threads =  6445306 ns  Pi calculée (basique 1) :  3.14159
Temps  pour 6 threads =  7722021 ns  Pi calculée (basique 1) :  3.14159
Temps  pour 7 threads =  5568842 ns  Pi calculée (basique 1) :  3.14159
Temps  pour 8 threads =  4909058 ns  Pi calculée (basique 1) :  3.14159
```

Un 1er ex. de tache décomposable : val. de π (suite)

Une **moyenne** des gains : processus (en **noir**) sur un bi-processeur Xeon ou avec des threads (en bleu) sur un mono-processeur I7



Deuxième exemple : accès concurrent

A propos des variables qui se marchent sur les pieds :

Exemple : trois threads incrémentent concurremment une variable globale.

- Qu'a-t-on à la fin dans cette variable ?
- Bien remarquer les valeurs affichées par les threads (dans la trace)

```
#include <thread>
#include <iostream>

int var_globale=0;

void incrementer(void){
    int boucle;
    printf("pthread  d'identite %ld \n",  (long)pthread_self());
    for(boucle = 0; boucle < 8000; boucle++) var_globale= var_globale+1;
    return;
}

int main(){
    std::thread t1( incrementer );
    std::thread t2( incrementer );
    std::thread t3( incrementer );

    t1.join();    t2.join();t3.join();
    std::cout << "var globale = " << var_globale << '\n';
    return 0;
}
```

Deuxième exemple : accès concurrent (suite)

- Trace d'exécution :

```
pthread  d'identite ...
pthread  d'identite ...
pthread  d'identite ...
var globale = 21078
```

```
Arffffff : il me fallait 24000 !
```

- Cause : voir plus loin l'anatomie d'une instruction d'incrémentation.

Une version plus élaborée de calcul de Pi

Avec gestion de valeur renvoyée (remplace **join**)

```
#define NUM_RECTS 1000000
#define NUM_THREADS 4

double callcul_pi_partiel_arctang(int arg){
  int i;
  int myNum = arg;
  double Somme = 0.0, Un_pas = 1.0 / NUM_RECTS, x;
  for (i = myNum; i < NUM_RECTS; i += NUM_THREADS){
    x = (i + 0.5f) / NUM_RECTS;
    Somme += 4.0f / (1.0f + x*x);
  }
  return Somme * Un_pas;
}
//---------------------------------------
int main() {
  int i,j;
  double pi=0.0;

  std::future<double> the_answer[NUM_THREADS];
  for (int i=0; i< NUM_THREADS; i++)
    the_answer[i]=std::async(std::launch::deferred, callcul_pi_partiel_arctang, i);

  // Le résultat disponible quand on fera get
  for (int i=0; i< NUM_THREADS; i++)
    pi += the_answer[i].get();

  cout << "Valeur approchee de Pi (methode 1) :  " <<  pi << endl;
  return 0;
}
```

Calcul de Pi : une autre méthode

- Dans cette autre version, on s'appuie sur la formule de **Gregory** (1638-1675) :

$$Arctan(t) = t - \frac{t^3}{3} + \frac{t^5}{5} - \frac{t^7}{7} + \frac{t^9}{9} - ...$$

➛ Cette série (une déclinaison du dév. de Tylor) converge pour $t \leq 1$

- On s'en accommode en fixant $\alpha = 45^\circ$ et $t = 1$:

$$45^\circ = \frac{\pi}{4} = Arctan(1) = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - ...$$

- Une réécriture de cette dernière série donnera :

$$\frac{\pi}{4} = 1 - \frac{1}{4-1} + \frac{1}{4+1} - \frac{1}{8-1} + \frac{1}{8+1} - \frac{1}{12-1} + \frac{1}{12+1} - \dots$$

$$\frac{\pi}{4} = 1 + \frac{1}{1-4} + \frac{1}{1+4} + \frac{1}{1-8} + \frac{1}{1+8} + \frac{1}{1-12} + \frac{1}{1+12} + \dots$$

$$\frac{\pi}{4} = 1 + \left[\frac{1}{1-(4*1)} + \frac{1}{1+(4*1)}\right] + \left[\frac{1}{1-(4*2)} + \frac{1}{1+(4*2)}\right] + \left[\frac{1}{1-(4*3)} + \frac{1}{1+(4*3)}\right] + \dots$$

$$\frac{\pi}{4} = 1 + \left[\frac{1}{1-4i} + \frac{1}{1+4i}\right]_{i=1} + \left[\frac{1}{1-4i} + \frac{1}{1+4i}\right]_{i=2} + \left[\frac{1}{1-4i} + \frac{1}{1+4i}\right]_{i=3} + \dots$$

$$\frac{\pi}{4} = 1 + \left[\frac{1}{1-4i} + \frac{1}{1+4i}\right]_{i=1}^{N} = 1 + \left[\frac{2}{1-(4i)^2}\right]_{i=1}^{N}$$

Calcul de Pi : une autre méthode (suite)

- Il suffira de faire calculer le terme itératif (en magenta) par k threads et y ajouter 1 pour obtenir $\frac{\pi}{4}$.
- Le code suivant conserve les bases de l'exemple précédent et modifie seulement la fonction de calcul. De plus, chaque thread écrira ses résultats dans une case séparée d'un tableau.

```
#define NUM_RECTS 1000000
#define NUM_THREADS 5

double global_sum[NUM_THREADS];      // Chaque thread fait ses calculs dans son coin
//-------------------------------------
void callcul_pi_partiel_arctang_Gregory(int myNum){
   cout << "Tache " << myNum << " Lancée\n";

    long  x;
    for (int i = myNum+1; i < NUM_RECTS; i += NUM_THREADS){
            // +1 car il faut commencer àpd 1 et non de zéro
        x = 4*i;
        global_sum[myNum]+= 2.0f / (1.0f - x*x);     // Forcer le calcul en flottant
    }
}
//-------------------------------------
```

Calcul de Pi : une autre méthode (suite)

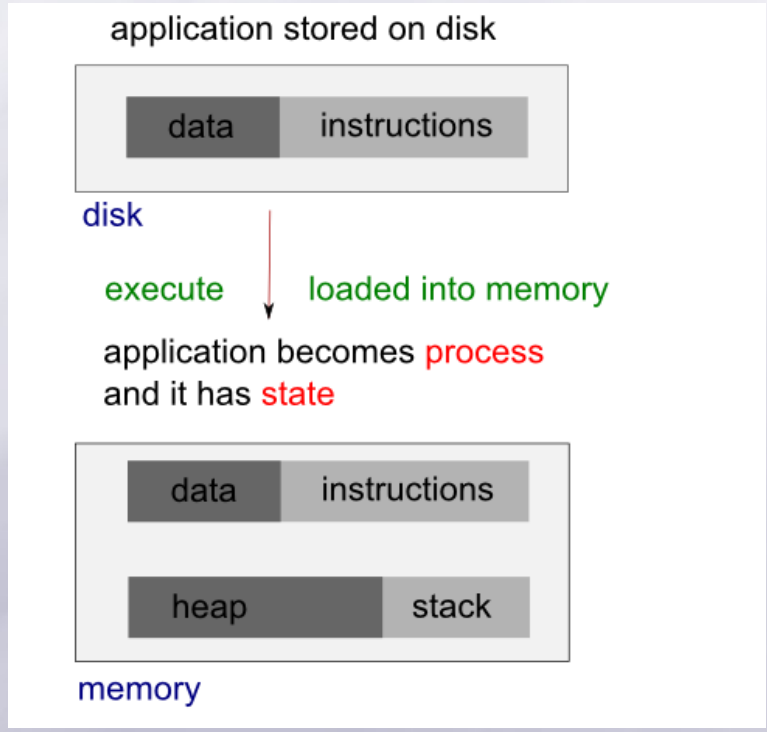
```
int main() {
    int i=0,j;
    double pi=0.0;

    std::thread Ids[NUM_THREADS];

    for(int k=0; k<NUM_THREADS; k++){
        global_sum[k]=0.0f;
        Ids[k]=thread(callcul_pi_partiel_arctang_Gregory, k);
    }
    for(int k=0; k<NUM_THREADS; k++){
        Ids[k].join();
        pi += global_sum[k];
    }

   // Dans pi, on a le terme en magenta. Y ajouter 1 puis multiplier par 4 pou robtenir Pi.
    cout << "Valeur approchee de Pi (methode Gregory) :  " << 4.0*(1.0+pi) << endl;
    return 0;
}

/* TRACE :
...
Valeur approchee de Pi (methode Gregory) :  3.14159
*/
```

Exemple ludique

- Course de chevaux !

Activité Parallèle : processus

Abstraction : d'un programme (sur disque) à un Processus (en mémoire)



• Le noyau (kernel) du système d'exploitation (SE) gère un flot de processus.

Noyau d'un SE : Thread et Processus

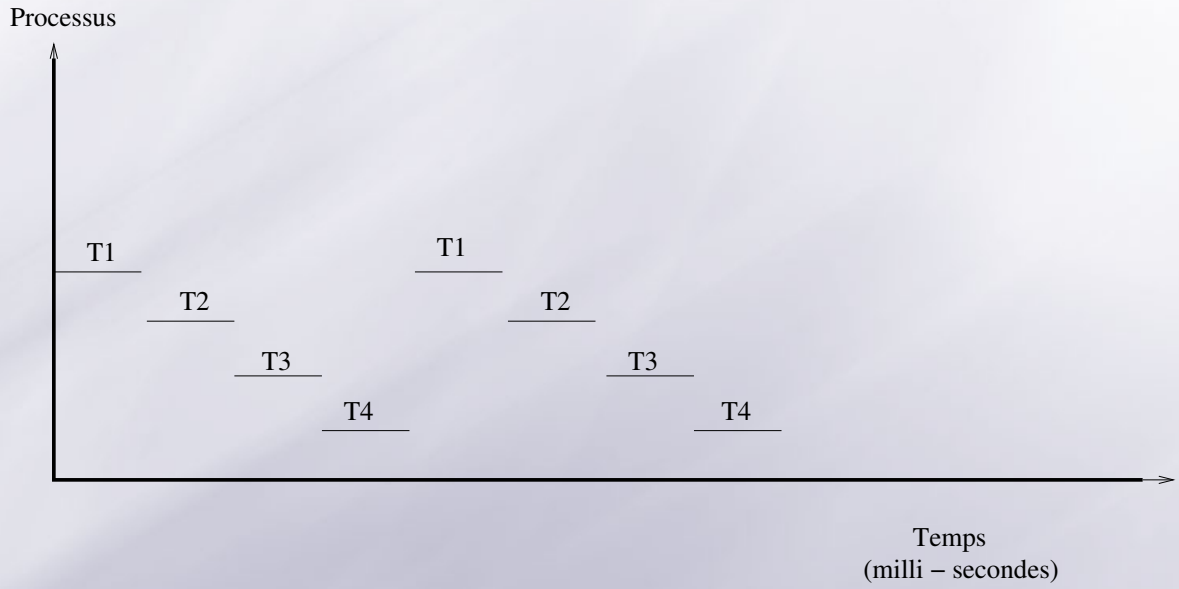
- Une abstraction principale : **processus** (ou *thread* = tâche).
- Un processeur est capable d'exécuter un ou plusieurs processus l'instant t,
  - Un *multiprocesseur* est capable de gérer à tout instant (au moins) autant de processus qu'il a de processeurs.

☞ Un processeur "multi cores" (+ Hyper-Threading) en est une déclinaison.

- Un noyau gère des flots (d'états) d'exécution de processus utilisateurs bloqués lorsqu'ils effectuent des appels systèmes.
  - Un processus bloqué ne consomme pas de temps processeur,
  - Il est réveillé par le noyau lorsque les ressources demandées sont dispos.
- Le noyau n'est pas en lui-même une tâche,
  - mais un ensemble de routines pouvant être appelées par les autres processus (exécution avec un certain niveau de privilèges).

Noyau d'un SE : Thread et Processus (suite)

- Gestion multi-tâches sur un processeur (le pseudo parallélisme)



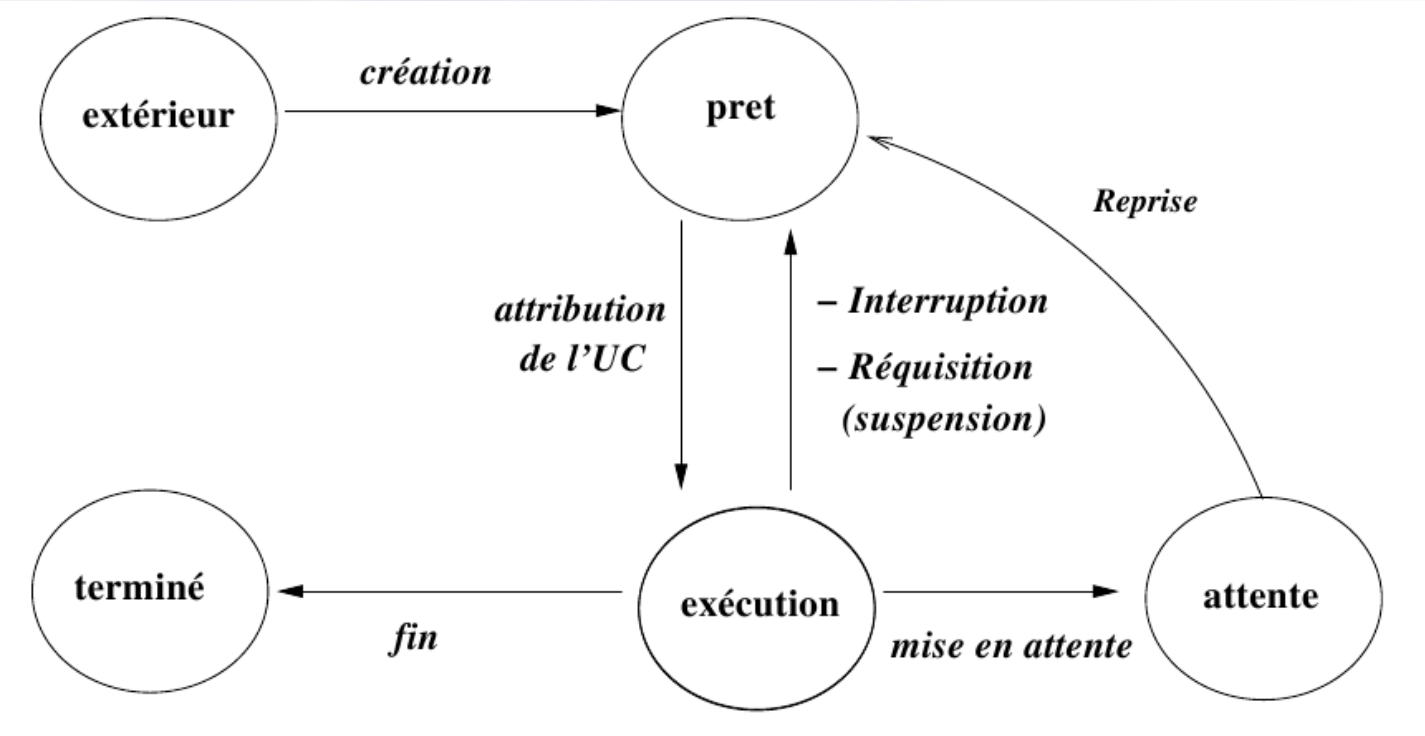
- Il peut y avoir différentes stratégies d'allocation du temps.
- Le pseudo parallélisme sera toujours d'actualité si le nombre de processus / thread dépasse la capacité du hardware.
  - ➛ Même avec les derniers **I9 d'Intel** à 18 cœurs / 36 Threads !

Noyau d'un SE : Thread et Processus (suite)

- Les noyaux multitâches permettent l'exécution de plusieurs processus sur un processeur,
  - en **partageant** le temps du processeur entre les processus.
  - ou avec du multi-threading (sur les processeurs récents)
- Pour une exécution (pseudo) parallèle, un noyau multitâche s'appuie sur les notions de :
  - commutation de contexte ;
  - ordonnancement ;
  - temps partagé.
- Pour les E/S : un traitement spécifique par l'ordonnanceur (attente, ...)
- Voir plus loin les détails sur les processus

Noyau d'un SE : Thread et Processus (suite)

Diagramme (simplifié) d'états des tâches gérées par le noyau :



Intérêts des threads

- Réactivité (exigence d'une "réponse à temps", cf. un ascenseur)
- Partage de ressources
- Économie (des ressources et du temps)
- Utilisation des (multi-) processeurs capables du parallélisme

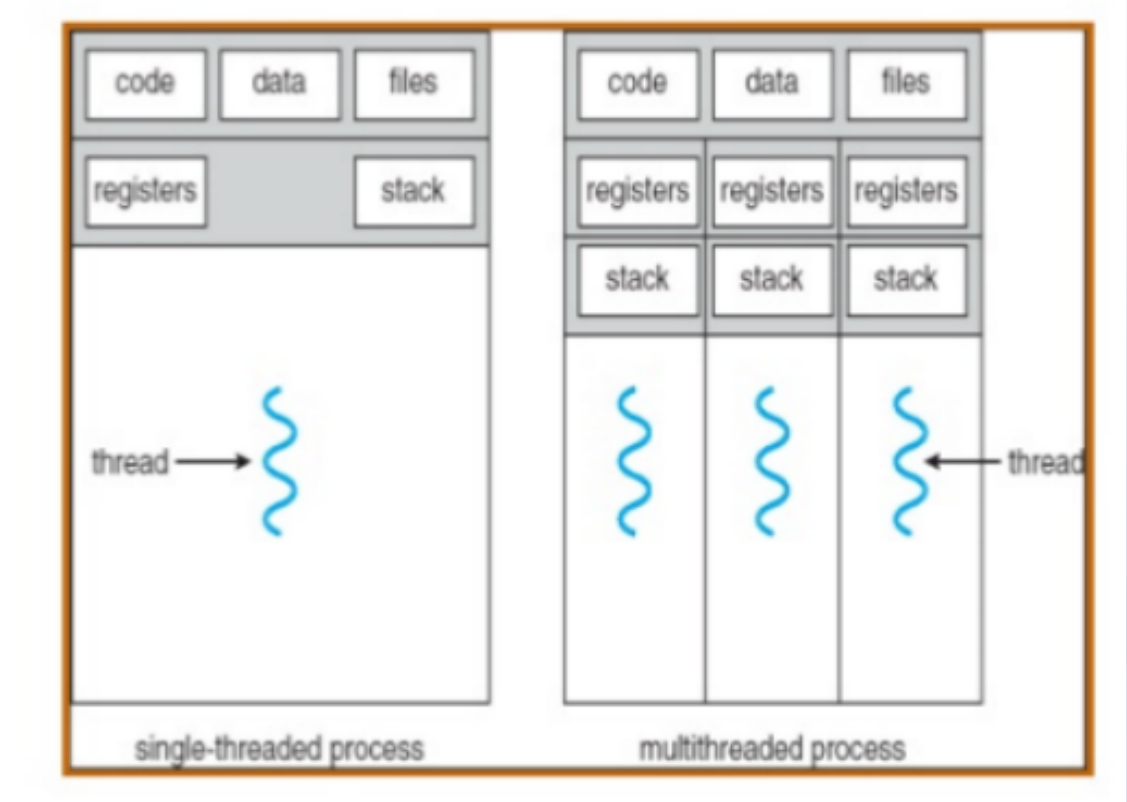
Par rapport aux processus, un thread :

- Prend moins de temps de création
- Prend moins de temps pour terminer
- Moins de temps nécessaire pour basculer (switcher) entre 2 threads,
- 2 threads partageant le même espace mémoire (et mêmes fichiers)

peuvent communiquer sans IPC (sauf nécessité de synchronisation).

Simple vs multiple threads

- Mono-thread vs multi-thread :



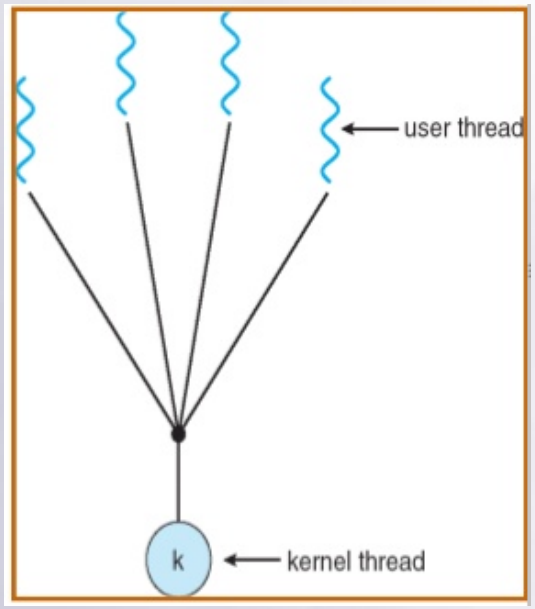
Modèles de Multi-Threading

- 3 schéma généraux :
  - Many-to-One
  - One-to-One
  - Many-to-Many

Cas Many-to-One :

- Des threads "user" pris en charge par un seul thread système ,
- La gestion de ces threads faite par une librairie efficace dans l'espace "user",
- En cas d'appel système bloquant, TOUT se bloque même si d'autres threads auraient pu continuer.
- Exemple : sur certains smartphones / ordinateurs, blocage provisoire d'appareil !

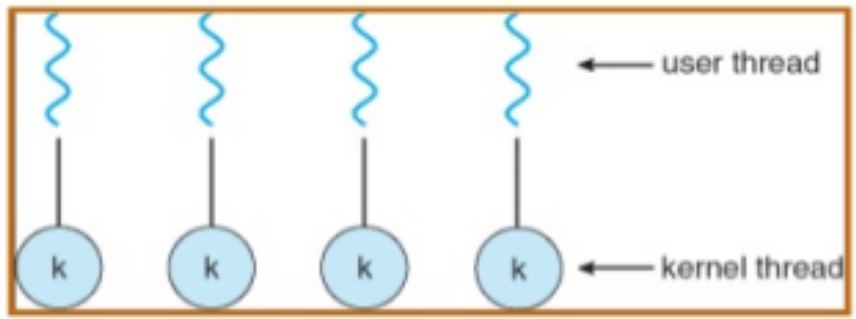
☞ Cas de Python (multi-thread mais synchrone !)



Modèles de Multi-Threading (suite)

Cas One-to-One :

- Chaque thread "user" pris en charge par un thread système,
- Exécution parallèle (multi processeurs / cores / threads)
- En cas d'appel système bloquant, d'autres threads continuent.



- Inconvénient : 2 threads créés à chaque fois
  ➛ un "user" + un "système" (lourd).

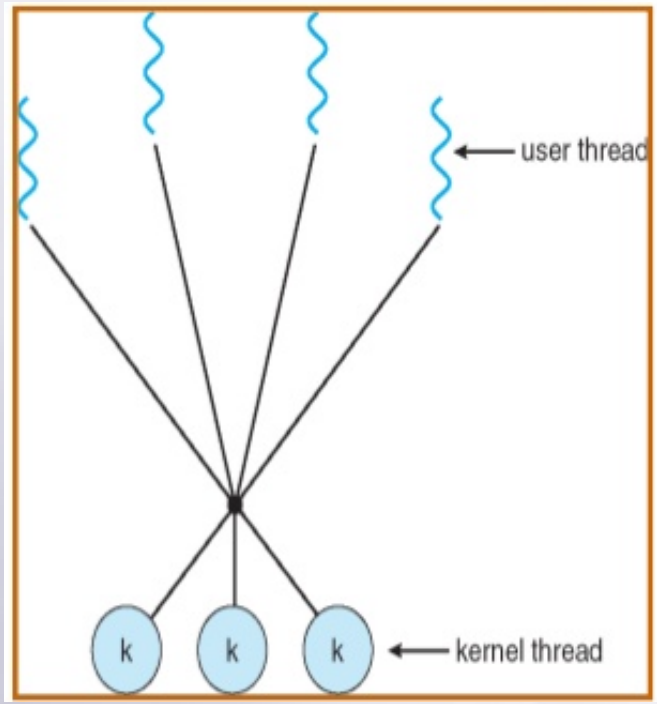
- Exemple : Linux, Windows (à partir de V95).

- Les processeurs (multi-cores) capables de multi-threading peuvent utiliser ce schéma.
  ➛ Un I7 quadri-coeurs + Hyperthreading peut gérer 8 threads "systèmes".

Modèles de Multi-Threading (suite)

Cas Many-to-Many :

- L'ensemble des threads "user" multiplexé sur un ensemble (de taille égale ou inférieur) de threads "système",
- Ce schéma combine le meilleur des 2 schémas précédents.
- En cas d'appel système bloquant, d'autres threads continuent.
- Implantation : un seul *processus* peut prendre en charge plusieurs threads "système" (suivant le nombre de CPUs / Cores),
- Exemple : RIX, HP-UX et Tru64-Unix.



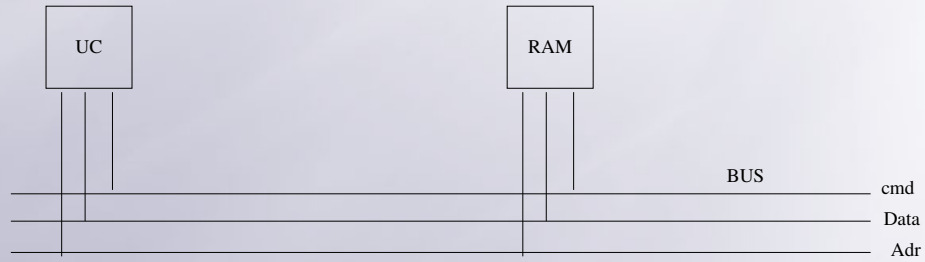
Anatomie d'une instruction

- Granulité d'activité d'un système (via "processus" / "tache", ...)

☞ Un exemple intuitif : qu'est-ce qui se passe lors de l'exécution de $N \leftarrow N + 1$ sur une machine basique (à *Accumulateur*) ?

- L'instruction est décomposée en Assembleur (l'*Accu* se mêle de tout !) :

Load N into *Accu*
Add #1 to *Accu*
Store *Accu* into N



N.B. : Un processeur plus récent utiliserait un registre (p.ex. *R1*) :

Load N, R1
Add R1, #1
Store R1, N

équivalent sur une machine à Accumulateur

Anatomie d'une instruction (suite)

Exemple : cadencement de 'Load N' :

- ➥ UC demande le Bus, dépose 1 cmd "Read RAM", Dépose @N, ...
- ➥ MMU : reçoit demande *Read*, récupère @N, dépose N sur le *Bus Data*
- ➥ L'UC récupère (après k cycles) ...

• Ces mécanismes sont à la disposition du SE (ce n'est pas lui qui s'en charge).

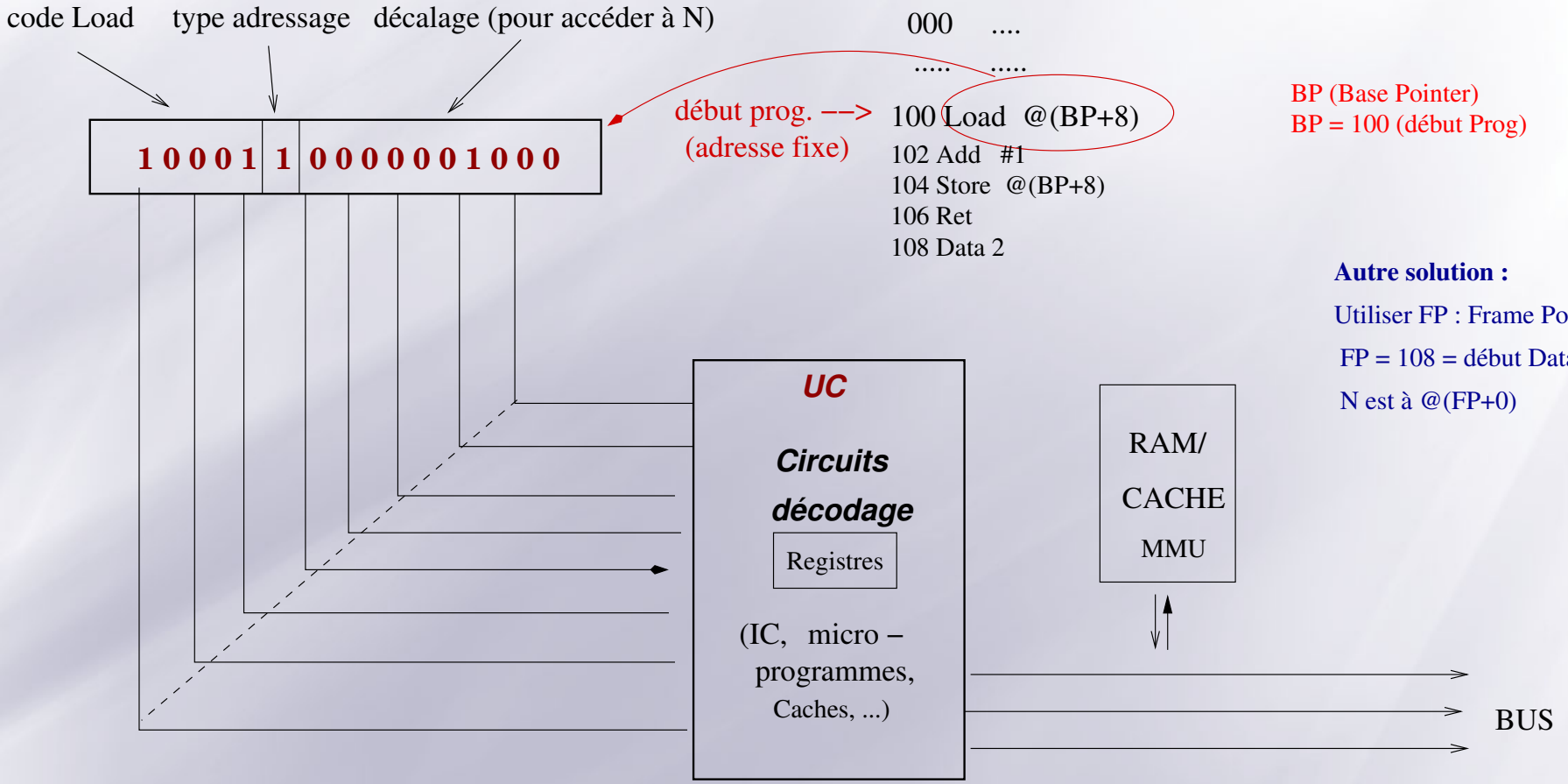
• Comment organiser les tâches plus compliquées ?

◦ Ne pas faire attendre l'UC (sauf dans les mono tâches).

- **Signal** et interruption (*Ready*, *Done*, ..) vs. **cycle** et top d'Horloge.
- Interruptions, Réquisition de Ressources (BUS), Priorité, ...

Anatomie d'une instruction (suite)

Décodage d'une instruction (*Load*) chargée dans le registre d'inst. de l'UC:



Anatomie d'une instruction (suite)

Un autre exemple : soit le programme

```
entier M, N ← 0 ;
M ← N+1 ;
afficher M ;
```

... et sa traduction de ces instructions en assembleur :

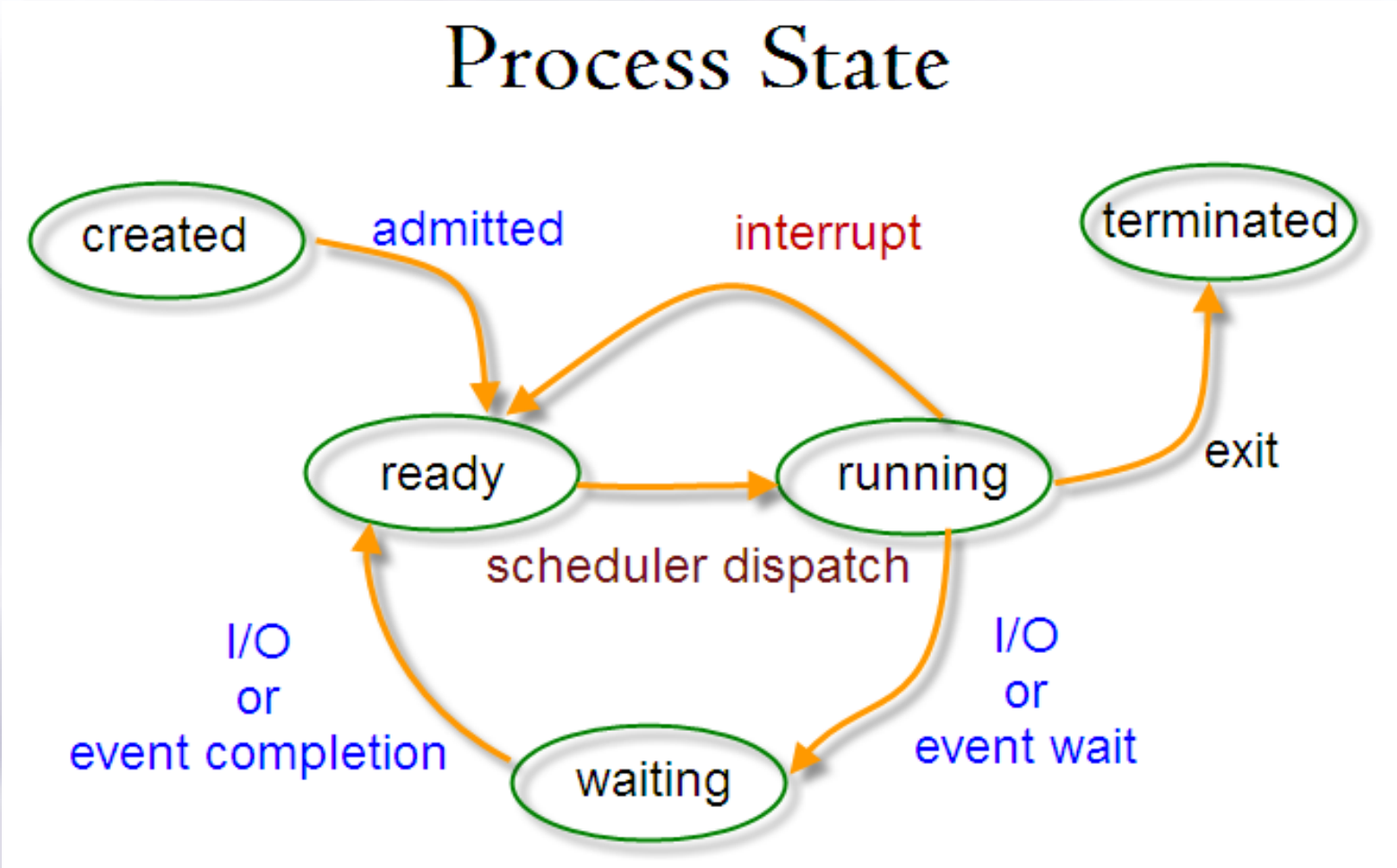
- machine à registres; instructions, adresses et données sur 2 octets :

```
0000    move    #0, @16     % mettre 0 dans N (déplacement +16 pour N)
0002    load    @16, R      % charger N dans le registre R
0004    add     #1, R       % ajouter la constante 1 à R
0006    store   R, @14      % sauvegarde R dans M (déplacement +14 pour M)
0008    push    @14         % mettre M au sommet de la pile
0010    call    affiche     % appeler la fonction affiche
0012    ret                 % retour à l'appelant
0014    data    4           % places pour les variables M et N
```

- A propos des appels de fonctions (pile).
  - Nbr. paramètres ajouté à la pile (ou codé dans l'appelée)
- Si l'adresse de début = X, on ajouter X à toutes les adresses (registre FP).
  - Cette *translation* peut être dynamique (en cas de *re location*)

Addendum : création de Processus avec fork

États d'un processus



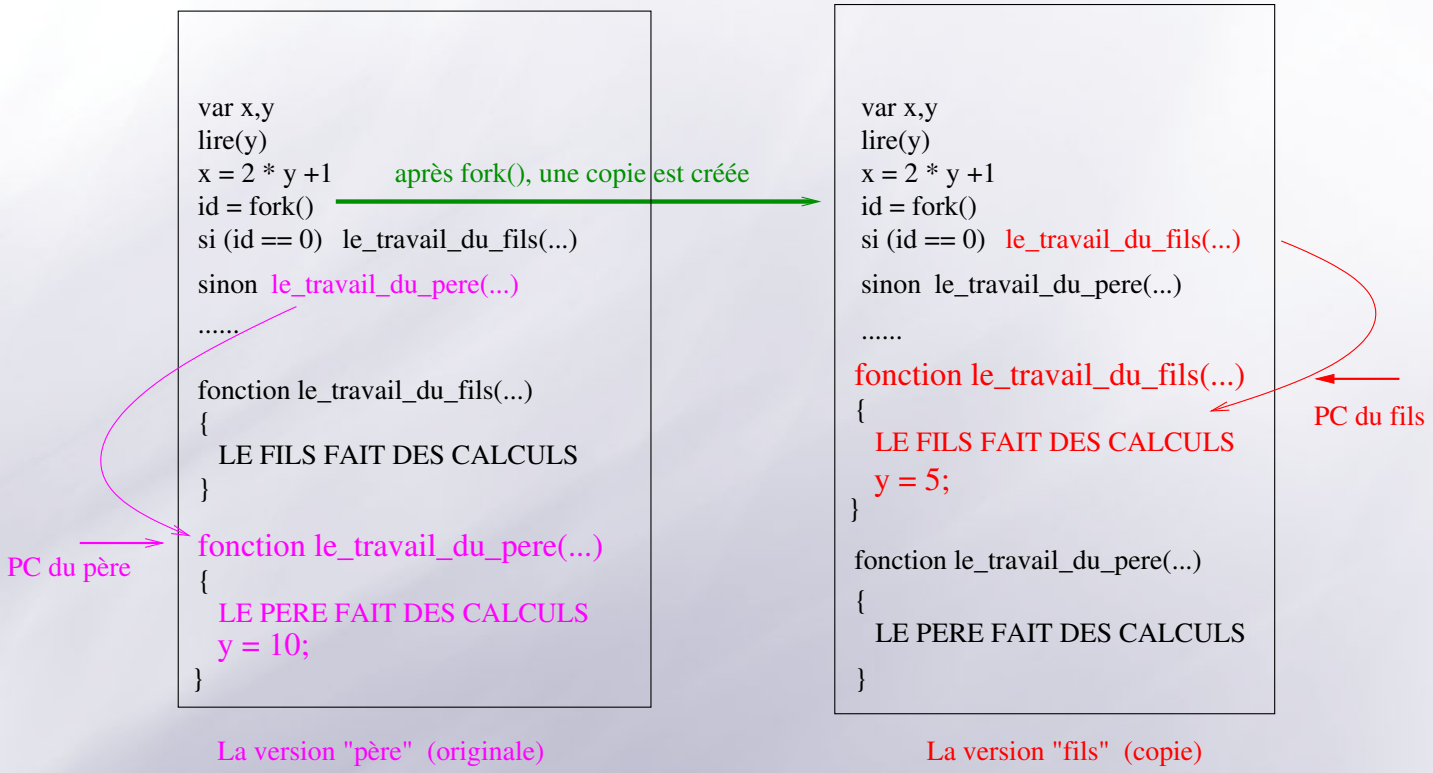
Addendum : création de Processus avec fork (suite)

- Un processus peut être créé par un autre (père - fils)
- Le fils peut à son tour en créer d'autres
  - graphe du père et ses descendants = une structure partiellement ordonnée.
- On appelle parfois "job" l'ensemble du processus père et ses descendants

Exemple

```
idfils = fork ();
if (idfils == 0)                    // en est dans le fils
    <Donner quelque chose à faire au fils>
else                                // on est dans le père
    <continuer la travail du père>
```

Addendum : création de Processus avec fork (suite)



- Aucun intérêt à copier le code,
  - il suffit de donner un ptr d'instruction à chacun (program counter=PC)
  - Mais les données sont copiées : chacun pour soi (chacun son y).

Addendum : création de Processus avec fork (suite)

Que fait-on au retour d'un *fork* (où le père s'est cloné !) ?

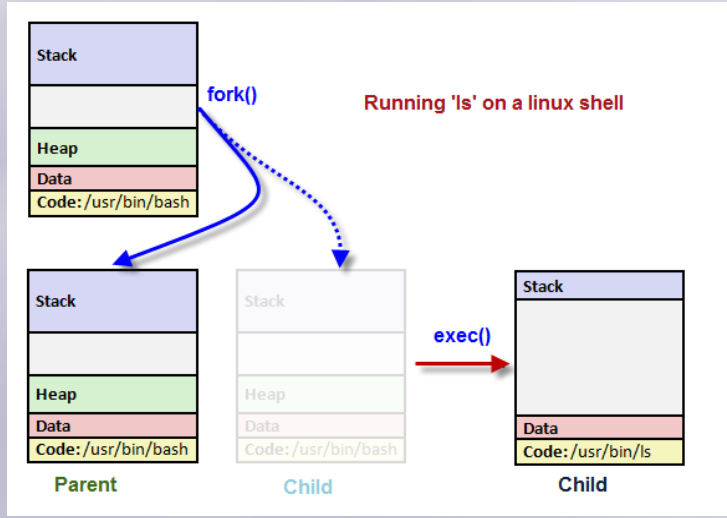
- On peut laisser les 2 codes se dérouler en parallèle faisant (une partie) de la même tâche (suivant des paramètres)
- Le fils peut s'enfermer dans une fonction du programme (e.g. surveillance d'un capteur)
- On peut utiliser la commande "**exec()**" pour exécuter un code différent

- Si **exec()** (où des paramètres d'appel peuvent être passés au fils) :
  - "exec" remplace le code + données du processus qui l'exécute par ceux du code chargé
  - le processus reste le même mais le programme exécuté change
  - "fork" + "exec" : exécution en parallèle (du père et du fils).

Addendum : création de Processus avec fork (suite)

Exemple (d'exec)

```
idfils = fork ();
if (idfils == 0)                    // en est dans le fils
    exec(fichier disque)
else                                // on est dans le père
    <continuer la travail du père>
```



Addendum : création de Processus avec fork (suite)

- Selon les systèmes, on peut visualiser la liste des processus :
  - Unix (MacOs, Linux, etc.) : *ps*
  - Windows : passer par l'interface graphique

  ➥ On peut en arrêter (*kill, signal*)

Addendum : création de Processus avec fork (suite)

- Au **boot**, un processus spécial (*init*) est l'ancêtre des autres
  - Il peut lancer un processus spécialisé "lanceur de programmes"

Pseudo code du lanceur d'application P

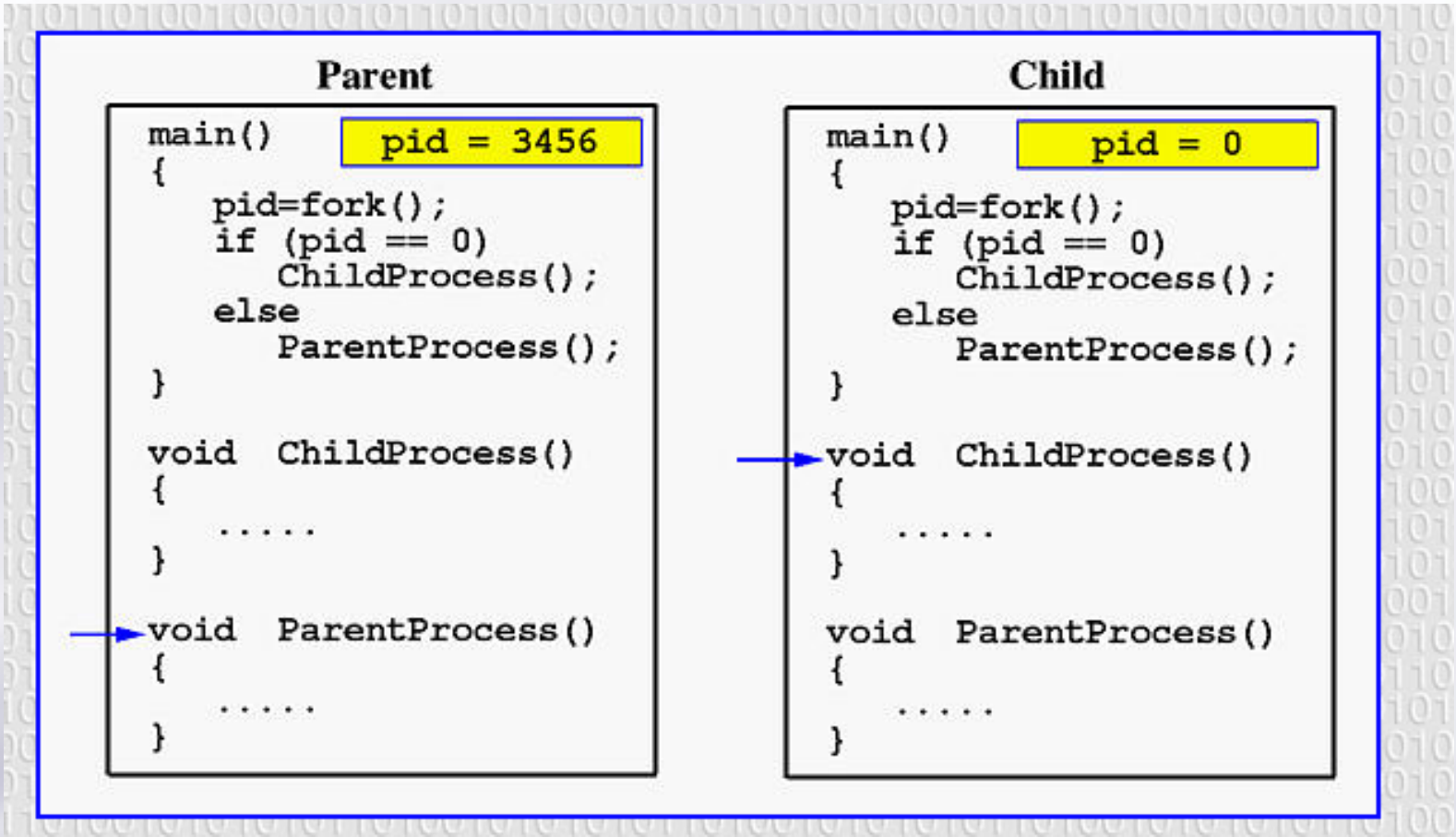
```
Id : process-id;
Do forever
    if   demande de lancement d'un programme P
    then     Id=fork();
             if (Id == 0)          % le fils
             then exec(P);     % attribuer P au demandeur (user)
             % else : on reste dans l'itération
    else attendre          % attente passive
    endIf
endDo
```

  - La demande est traitée puis le lanceur reste dans son itération.

- Bien entendu, une application quelconque peut elle aussi faire des *fork*s.

Addendum : création de Processus avec fork (suite)

Récapitulons : comprendre *Fork* :



Recommandations sur les threads

Création des threads :

- Ne créer jamais un thread avant la fonction **main** (e.g. par déclaration globale/ static, dans un constructeur, ...)
  - ➛ Perturbera la création des objets "static"s
- Le nbr de threads créés doit être indép de la charge du système
  - ➛ Sinon, on ne peut pas mesurer les performances
- Au lieu d'en recréer, privilégier la réutilisation des threads en leur affectant plusieurs rôles
  - ➛ Par exemple pour les E/S ou les *callbacks* (boucle événmentielle)

Recommandations sur les threads (suite)

Threads et son parent (son créateur) :

☞ *join()* permet au parent d'attendre la fin d'un thread.

➛ Comme pour un RDV.

☞ Un mécanisme similaire existe pour récupérer un résultat (renvoyé par *return*)

☞ *detach()* est utilisé par un thread qui ne veut pas participer au RDV avec son parent (via *join()*)

➛ On s'en sert lorsque le threads récalcitrant correspond à une tache de fond.

detach() permet au thread fils de continuer (et terminer) sans que le parent l'attende;

➛ de plus, *detach()* libère les ressources nécessaires à un *join()*.

Recommandations sur les threads (suite)

Terminaison des threads :

- Mort naturelle : laisser un thread retourner normalement de sa fonction assignée.
- En C++11/14 : pas de *exit* (= suicide) ou *cancel* (meurtre) qui peuvent créer les conditions de *deadlock* que l'on ne pourra pas détecter / débloquer.
- Faire en sorte que l'on puisse toujours réveiller un thread pour une terminaison normale.
- Voir aussi *detach()*

☞ *std::terminate()* (nécessite *#include <exception>*)
permet d'abandonner l'exécution d'un thread mais aussi du programme (avortement équivalent à *exit()*, indépendant des threads)

Addendum : Librairie std

Quelques éléments extraits de la documentation sur les threads.

Thread : permet l'exécution de code de manière parallèle sur différents *cores* des processeurs.

- Inclure #include <thread>
- Fonctions de gestion des threads :
  - *yield* : suggests that the implementation reschedule execution of threads
  - *get_id* : returns the thread id of the current thread
  - *sleep_for* : stops the execution of the current thread for a specified time duration
  - *sleep_until* : stops the execution of the current thread until a specified time point

Addendum : Librairie std (suite)

- **Exclusion mutuelle**

Mutual exclusion algorithms prevent multiple threads from simultaneously accessing shared resources. This prevents data races and provides support for synchronization between threads.

- Inclure #include <mutex>
  - *mutex* : provides basic mutual exclusion facility
  - *timed_mutex* : provides mutual exclusion facility which implements locking with a timeout
  - *recursive_mutex* : provides mutual exclusion facility which can be locked recursively by the same thread
  - *recursive_timed_mutex* : provides mutual exclusion facility which can be locked recursively by the same thread and implements locking with a timeout

Addendum : Librairie std (suite)

Et en incluant #include <shared_mutex>

- *shared_mutex* : provides shared mutual exclusion facility
- *shared_timed_mutex* : provides shared mutual exclusion facility

- **Generic mutex management** définies par #include<mutex>
  - *lock_guard* : implements a strictly scope-based mutex ownership wrapper
  - *unique_lock* : implements movable mutex ownership wrapper
  - *shared_lock* : implements movable shared mutex ownership wrapper
  - *defer_lock_t*, *try_to_lock_t*, *adopt_lock_t* :

    tag type used to specify locking strategy
  - *defer_lock*, *try_to_lock*, *adopt_lock* :

    tag constants used to specify locking strategy

Addendum : Librairie std (suite)

- **Generic locking algorithms**
  - *try_lock* : attempts to obtain ownership of mutexes via repeated calls to try_lock
  - *lock* : locks specified mutexes, blocks if any are unavailable
- **Call once**
  - *once_flag* : helper object to ensure that call_once invokes the function only once
  - *call_once* : invokes a function only once even if called from multiple threads

Addendum : Librairie std (suite)

Condition variables

A condition variable is a synchronization primitive that allows multiple threads to communicate with each other. It allows some number of threads to wait (possibly with a timeout) for notification from another thread that they may proceed. A condition variable is always associated with a mutex.

- Defined in header <condition_variable>
  - *condition_variable* : provides a condition variable associated with a std::unique_lock
  - *condition_variable_any* : provides a condition variable associated with any lock type
  - *notify_all_at_thread_exit* : schedules a call to notify_all to be invoked when this thread is completely finished
  - *cv_status* : lists the possible results of timed waits on condition variables

Addendum : Librairie std (suite)

Futures

The standard library provides facilities to obtain values that are returned and to catch exceptions that are thrown by asynchronous tasks (i.e. functions launched in separate threads). These values are communicated in a shared state, in which the asynchronous task may write its return value or store an exception, and which may be examined, waited for, and otherwise manipulated by other threads that hold instances of std::future or std::shared_future that reference that shared state. Defined in header <future>

- *promise* : stores a value for asynchronous retrieval
- *packaged_task* : packages a function to store its return value for asynchronous retrieval
- *future* : waits for a value that is set asynchronously
- *shared_future* : waits for a value (possibly referenced by other futures) that is set asynchronously

Addendum : Librairie std (suite)

- *async* : runs a function asynchronously (potentially in a new thread) and returns a std::future that will hold the result
- *launch* : specifies the launch policy for std::async
- *future_status* : specifies the results of timed waits performed on std::future and std::shared_future

  Et Future errors :
- *future_error* : reports an error related to futures or promises
- *future_category* : identifies the future error category
- *future_errc* : identifies the future error codes

Addendum : Librairie std (suite)

Détails sur la classe std::thread

Un thread permet d'exécuter du code de manière asynchrone et simultanée .

Un std::thread peut également être en état d'attente / suspension (par constructeur vide, ou suite à un *move*, *detach*, *join*).

Un thread peut ne pas être associé à un autre thread (parent) après un *detach*.

Deux threads ne peuvent pas peut représenter la même entité parallèle car un thread n'est pas copiable (pas de copie-constructeur) ni "affectatble" ('=' provoque un move) bien qu'il soit MoveConstructible (construit suite à un move) ou MoveAssignable (par l'opérateur '=') .

Classe membre :

- *id* : represents the id of a thread

  Les opérateurs applicables : le *constructeur*, $==$, $!=$, $<$, $<=$, $>$, $>$.

  Écriture à l'aide de l'opérateur habituel $<<$

- La fonction membre *get_id* donne l'Id d'un thread.

Addendum : Librairie std (suite)

Member functions

- *(constructor)* : constructs new thread object
- *(destructor)* : destructs the thread object, underlying thread must be joined or detached
- *operator=* : moves the thread object (➛ Pas de copie-constructeur, ni affectation)

Observateurs

- *joinable*: checks whether the thread is joinable, i.e. potentially running in parallel context
- *get_id* : returns the id of the thread
- *native_handle* : returns the underlying implementation-defined thread handle

Addendum : Librairie std (suite)

- *hardware_concurrency* : eturns the number of concurrent threads supported by the implementation

Operations

- *join* : waits for a thread to finish its execution
- *detach*: permits the thread to execute independently from the thread handle
- *swap* : swaps two thread objects
- See also *move* (no assignement between threads).

Non-member functions

- *std::swap(std::thread)* : specializes the std::swap algorithm

Exemples

- Exemple de **get_id**

```cpp
#include <iostream>
#include <thread>
#include <chrono>
#include <mutex>

std::mutex g_display_mutex;

void foo(){
  std::thread::id this_id = std::this_thread::get_id();

  g_display_mutex.lock();
  std::cout << "thread " << this_id << " sleeping...\n";
  g_display_mutex.unlock();

  std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main(){
  std::thread t1(foo);
  std::thread t2(foo);

  t1.join();
  t2.join();
}

// Exemple de sortie :
// thread 0x2384b312 sleeping...
// thread 0x228a10fc sleeping...
```

Exemples (suite)

- Exemple de **yield**

  Temporisation en combinant le temps et *yield* :

```
#include <iostream>
#include <chrono>
#include <thread>

// "busy sleep" while suggesting that other threads run
// for a small amount of time
void little_sleep(std::chrono::microseconds us){
  auto start = std::chrono::high_resolution_clock::now();
  auto end = start + us;
  do {
    std::this_thread::yield();
  } while (std::chrono::high_resolution_clock::now() < end);
}

int main(){
  auto start = std::chrono::high_resolution_clock::now();

  little_sleep(std::chrono::microseconds(100));

  auto elapsed = std::chrono::high_resolution_clock::now() - start;
  std::cout << "waited for "
       << std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count()
       << " microseconds\n";
}

// Exemple de sortie :
// waited for 128 microseconds
```

Exemples (suite)

- Exemple d'attente et de temporisation :

  Temporisation directe avec *sleep_for* :

```cpp
#include <iostream>
#include <chrono>
#include <thread>

int main()
{
  using namespace std::literals;
  std::cout << "Hello waiter" << std::endl;
  auto start = std::chrono::high_resolution_clock::now();
  std::this_thread::sleep_for(2s);
  auto end = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double, std::milli> elapsed = end-start;
  std::cout << "Waited " << elapsed.count() << " ms\n";
}

// Trace possible:
// Hello waiter
// Waited 2000.12 ms
```

Exemples (suite)

Exemple future - promise

```
/* Question :   how you can pass parameters and handle exceptions between threads (which also
  explains how a high-level interface, such as async(), is implemented).

 Of course, to pass values to a thread, you can simply
    - pass them as arguments.
    - If you need a result, you can pass return arguments by reference,

 However, another general mechanism is provided to pass result values and exceptions as outcomes of
    a thread: class std::promise.

  A promise object is the counterpart of a future object.
  Both are able to temporarily hold a shared state, representing a (result) value or an exception.

   While the future object allows you to retrieve the data (using get()), the promise object enables
     you to provide the data (by using one of its set_...() functions). The following example
    demonstrates this:
*/

//  g++-4.8 -std=c++11  ce_fichier.cpp -lpthread

#include <thread>
#include <future>
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>
#include <functional>
#include <utility>
```

Exemples (suite)

```
void doSomething (std::promise<std::string>& p) {
  try {
    // read character and throw exceptiopn if 'x'

    std::cout << "read char ('x' for exception): ";
    char c = std::cin.get();
    if (c == 'x') {
      throw std::runtime_error(std::string("char ")+c+" read");
    }

    std::string s = std::string("char ") + c + " processed";
    p.set_value(std::move(s)); // store result
  }
  catch (...) {
    p.set_exception(std::current_exception());  // store exception

//  If you want the shared state to become ready when the thread really ends -- to ensure the
//  cleanup of thread local objects and other stuff before the result gets processed -- you have
//  to call set_value_at_thread_exit() or set_exception_at_thread_exit() instead:
//          p.set_exception_at_thread_exit(std::current_exception());
  }
}

int main(){
  try{
    // start thread using a promise to store the outcome
    std::promise<std::string> p;
    std::thread t(doSomething,std::ref(p));
    t.detach(); // en tache de fond
```

Exemples (suite)

```
    // create a future to process the outcome
    std::future<std::string> f(p.get_future());

    // process the outcome
    std::cout << "result: " << f.get() << std::endl;
  }
  catch (const std::exception& e) {
    std::cerr << "EXCEPTION: " << e.what() << std::endl;
  }

  // La syntaxe de ce catche est OK (on choppe n'importe quoi
  // Mais comment ce catche marche sans 'try')
  catch (...) {
    std::cerr << "EXCEPTION " << std::endl;
  }
}

/* TRACE

read char ('x' for exception): a
result: char a processed

read char ('x' for exception): x
EXCEPTION: char x read
*/
```

Exemples (suite)

- Suite : voir les exemples et cours-2...

TabMat