ARTRAGIS

Zeste de savoir

L'asynchrone et le multithread en .NET

22 mars 2019

Table des matières

1. Introduction

En informatique, il existe un besoin constant dans les programmes : la performance. Plus précisément, le but est d'accomplir le plus rapidement possible une liste de tâches.

Très souvent, les performances sont apportées par un algorithme de meilleure qualité. Mais il n'est pas rare que l'engorgement subsiste. Certains vous encourageront alors à passer à des "langages performants" tels que le C++.

Ce comportement n'est pas forcément adapté à votre besoin. Développer dans ces langages peut, parfois, s'avérer très complexe. En plus changer de technologie, c'est long.

Alors, continuez à utiliser C# ! Vous n'aurez pas besoin d'un doctorat en programmation système pour rendre vos programmes performants. C# possède plein d'outils qui vous permettront d'optimiser tout ça **simplement**.

Avant tout, je vous propose de vous poser trois questions :

— Combien de temps mon programme passe-t-il à attendre ?
— Y a-t-il plusieurs traitements que je puisse faire en parallèle[1] ?
— Mon programme utilise-t-il au mieux les ressources de l'ordinateur (exemple : un processeur multicoeur) ?

1. c'est à dire en même temps les uns que les autres

Vous vous rendrez vite compte qu'en informatique, on peut tout à fait faire un bébé en un mois du moment qu'on a 10 femmes[2], et que cela est rendu possible par les techniques d'asynchronisme.

Le tutoriel se veut accessible mais il faudra que vous ayez les bases en programmation. J'entends par là savoir les structures de contrôle de base (du `if` au `using`) et savoir ce que représente une classe. Il serait intéressant que vous ayez déjà créé une interface graphique.

Ce tutoriel vous sera particulièrement utile si vous faites des applications mobiles. Dès qu'une tâche demande d'attendre, les applications mobiles vous obligent à utiliser l'asynchrone.

2. await/async, un couple inséparable

Le C# se veut dès sa conception un langage *moderne*. Cela signifie qu'il essaie d'apprendre des erreurs et des succès des langages qui ont existé avant lui.

L'un de ces succès a été la mise en place de systèmes de coroutines et de multithreading facilement programmables.

La méthode choisie pour obtenir cette facilité est souvent[3] l'introduction de deux mots clefs async et await. Arrivés en C#5/.NET4.5 ces deux mots clefs sont inséparables.

2.1. Utiliser async et await

Ces mots clefs sont plutôt débrouillards : ils exécutent à eux seuls une incroyable quantité d'instructions.

Avant d'en détailler les subtilités, faisons en sorte de garder un esprit clair et définissons LE terme le plus important de ce cours : ***asynchrone***.

Mettons nous dans une situation de votre vie courante : faisons cuire des cookies.

Un résumé de la recette peut être celle-ci :

```
jusqu'à ce que la file des ingrédient soit vide
   prenez l'ingrédient
   enlevez-le de son emballage/coquille
   déterminez la quantité adéquate
   mettez la quantité dans le récipient
   mélangez
formez les cookies
```

2. oui, j'ai bien dit 10 femmes, c'est pas une erreur de calcul.
3. Et cela n'est pas une mince affaire comme le prouve l'exemple de la version 3.5 du langage python

```
mettez-les au four
attendre quelques dizaines de minutes
sortez les cookies du four
```

Pour bien comprendre notre problématique, mettez-vous dans la peau d'un système d'exploitation, ou d'un processeur. Vous avez exécuté une série d'instructions assez basiques, et d'un coup, on vous demande d'attendre.

Alors vous attendez[4].

Et si quelqu'un vous demande de l'aide, vous dites "non, je dois attendre".

Avouons que cela est peu pratique. Alors vous avez une idée. Vous vous dites "je sais, je vais mettre un minuteur, et je reviendrai dans la cuisine uniquement quand il sonnera".

Voilà, vous venez de faire votre première action asynchrone.

Maintenant, vous êtes libre d'aider ceux qui vous le demandent. La seule condition : il faudra que vous soyez en mesure d'entendre le bip sonore et que vous ne soyez pas trop loin de la cuisine afin de ne pas laisser trop longtemps les cookies cuire. De même il faut que vous gardiez en mémoire que vous attendez ce bip sonore.

Pour une application en C# ça sera le travail de async et await de faire tout ça. Alors allons-y, codons :

```
namespace test_cookie
{
    class Program
    {
        static void Main(string[] args)
        {
            faitDesCookies();
            Console.In.ReadLine();
        }
        static async Task faitDesCookies() // indique que la
            méthode va faire quelque chose d'asynchrone
        {
            Queue<Ingredient> liste = Ingredient.GetRecette();
            while (liste.Count > 0)
            {
                Ingredient ingredient = liste.Dequeue();
                ingredient.Take();
                ingredient.UnPackage();
                ingredient.MesureQuantity();
                Console.Out.WriteLine("ajouté");
                Console.Out.WriteLine("On mélange");
            }
```

4. Et le processeur passe beaucoup de temps à attendre, comme le montre ce billet

```
22
                Console.Out.WriteLine("On forme les cookies et on les met au fo
23          // attend pendant 5 secondes mais ne truste pas les
                ressources
24          await Task.Factory.StartNew(() => {
                Console.Out.WriteLine("On met le minuteur.");
25
                                                    System.Threading.Thread.Sle
26
                                                    Console.Out.WriteLine("DRI
                                                    });
27          Console.Out.WriteLine("Sortir les cookies du four");
28      }
29  }
30 }
```

Contenu masqué n°1

Hey, il y a une erreur dans ton code, ta méthode elle retourne pas de Task !

Eh eh ! je vous avais prévenu : async et await font vraiment beaucoup, beaucoup de choses. Et notamment il y a une sorte de ”return magique” qui est fait.

Plus précisemment, quand vous dites qu'une méthode est async, vous annoncez que ça sera une *tâche* qui va devoir un jour attendre. Du coup le compilateur va regarder le type de retour de la fonction. Et il s'attend à soit `Task` soit `Task<Un type personnel>`.

Une fois cela fait, il va *instancier* cet objet, et y exécuter le code de la méthode. Si vous avez dit `async Task` il comprend ”la tâche à exécuter ne retourne rien”. C'est pourquoi je n'ai pas besoin de faire de `return` dans ma fonction. A l'opposé si j'avais demandé un `async Task<int>` il se serait attendu à ce que je retourne un `int`.

Vous trouverez dans la documentation que `async void` est aussi possible MAIS il est fortement déconseillé. Il n'est là que pour un seul cas : quand vous voulez créer un event listener. Nous reviendrons plus tard sur ce cas.

Plus tard je fais donc appelle à `await`. Ce dernier va dire au programme principal ”bon là je vais devoir attendre un signal, alors je te laisse libre mais reviens me voir rapidement”.

Et ton histoire de Task factomachin là ?

await n'est capable d'attendre que des fonctions asynchrones. On a vu qu'en fait une fonction asynchrone est une Task qui a été créé par async. Mais on peut aussi la créer nous même,pour

cela, il faut utiliser `Task<Votre Type De Retour>.Factory.StartNew(()=>lecode de la fonction; return ce_qu_il_faut)`. C'est à la main la même chose que lorsqu'on avait fait grâce à `async`.

2.2. L'objet Task volume 1

Task est un objet particulièrement intéressant car il ouvre les portes d'un nombre assez conséquent de fonctionnalités et de concepts. Nous l'explorerons tout au long de ce cours, mais je tenais à vous introduire un deuxième concept important dans ce cours : le **parallélisme**.

Reprenons notre recette de cookies, et pallions à un problème majeur : nous n'avons pas nettoyé notre cuisine !

L'idée serait de dire "nettoyons notre cuisine pendant que nous attendons".

Techniquement, il existe deux grandes solutions pour paralléliser des tâches :

— Utiliser deux unités de calculs différentes, autrement dit les coeurs de votre processeur ;
— Utiliser les *coroutines*.

L'idée derrière les coroutines est celle-ci :

— lancer un "cadenceur"[5] en tâche de fond ;
— lancer les coroutines une par une dans le processus du cadenceur ;
— régulièrement le cadenceur vérifie l'état des coroutines et avertit le programme principal de leur fin.

5. En anglais, ça se dit *scheduler*. Toujours bon à garder pour les recherches sur le web.

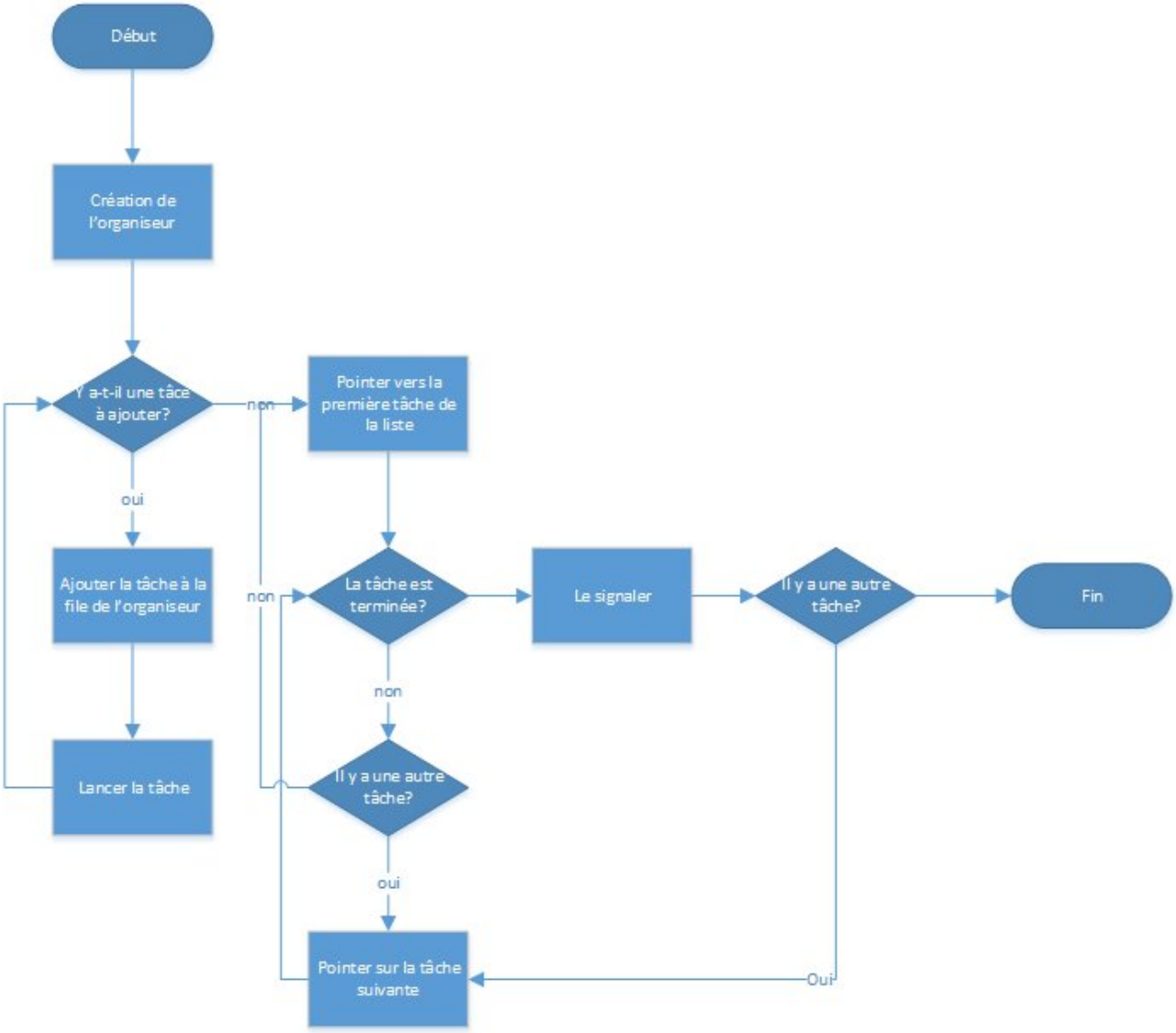

FIGURE 2. – Schéma de fonctionnement d'une coroutine[6].

L'idée maintenant sera de lancer les `Task` asynchrones puis de demander "attend qu'elles soient toutes finies".

Et ça se fait très simplement :

```
List<Task> tasks = new List<Task>();
// On lance les tâches mais on ne les attend pas
tasks.Add(Task.Factory.StartNew(() => {
    Console.Out.WriteLine("On met le minuteur.");

                                                    System.Threading.Thread.Sle

                                                    Console.Out.WriteLine("DRI
                                                    });
```

6. Vous pourrez trouver plus d'information sur ce lien , en anglais.

```
6 tasks.Add(Task.Factory.StartNew(()=>
      {System.Threading.Thread.Sleep(4000);/*trop facile de nettoyer
      :p*/});
7 await Task.WhenAll(tasks);// puis on attend que tout soit fini
```

Une fonction `WaitAll` existe mais ce n'est pas une méthode asynchrone, elle met en pause le programme. Je vous conseille d'utiliser cette méthode dans votre fonction main qui ne peut être asynchrone. A l'opposé, partout ailleurs, vous souhaiterez la majorité du temps utiliser `WhenAll`

Si vos `Task` retournaient une valeur, un entier par exemple, vous pourriez retrouver l'ensemble des valeurs dans le tableau que retourne `WhenAll`.

L'objet `Task` est vraiment primordial, il est à la base de toute un paradigme de programmation asynchrone appellé "Task-based Asynchronous Pattern" qu'on peut traduire par "Programmation Asynchrone basée sur les Tâches". Vous vous doutes bien qu'il existe de ce fait deux autre "Patterns", nous le verrons plus tard. Je vous conseille néanmoins d'utiliser le TAP.

3. TP : Un chargeur de liens web

Le but du TP qui va suivre est de vous permettre de vous mettre en confiance avec les notions que nous venons de voir tout en vous permettant d'*explorer* l'API.

Il se découpe en trois morceaux :

1. Le programme de base que vous devez être capable de faire seul. La correction sera néanmoins donnée.
2. Quelques pistes d'amélioration qui s'appuient sur les notions déjà vues, aucune correction ne sera donnée néanmoins. Je vous conseille de poser vos questions sur le forum avec le tag [async].
3. Une notion pratique un peu plus avancée qui demandera un peu d'exploration et de compréhension.

L'exercice de base

3.0.1. Énoncé

Votre mission, si vous l'acceptez : créer un système qui permet d'extraire tous les liens présents dans plusieurs pages web. Une fois ces liens obtenus, il vous faudra envoyer une requête vers ces liens et sauvegarder la page html qui leur est associé. En fin de programme, vous afficherez le temps mis pour récupérer toutes ces pages.

Vous pouvez utiliser une application console, WinForm, WPF à votre convenance.

Pour envoyer des requêtes, je vous conseille de regarder HttpWebRequest . Pour traiter le HTML vous pouvez utiliser le package HtmlAgility ou si vous aimez Linq : LinqToXML.

Pour mieux observer l'asynchronisme des tâches, je vous conseille de terminer chaque tâche par un `Console.WriteLine("Nom de la tâche " + une_autre_info)`. De même, je vous conseille d'exécuter plusieurs fois le même programme pour observer l'ordre d'exécution des tâches. Enfin, ayiez un oeil sur votre gestionnaire de ressources. Pour y accéder, accédez au gestionaire des tâche ctrl+Maj+echap puis "ouvrir le gestionnaire de ressources".

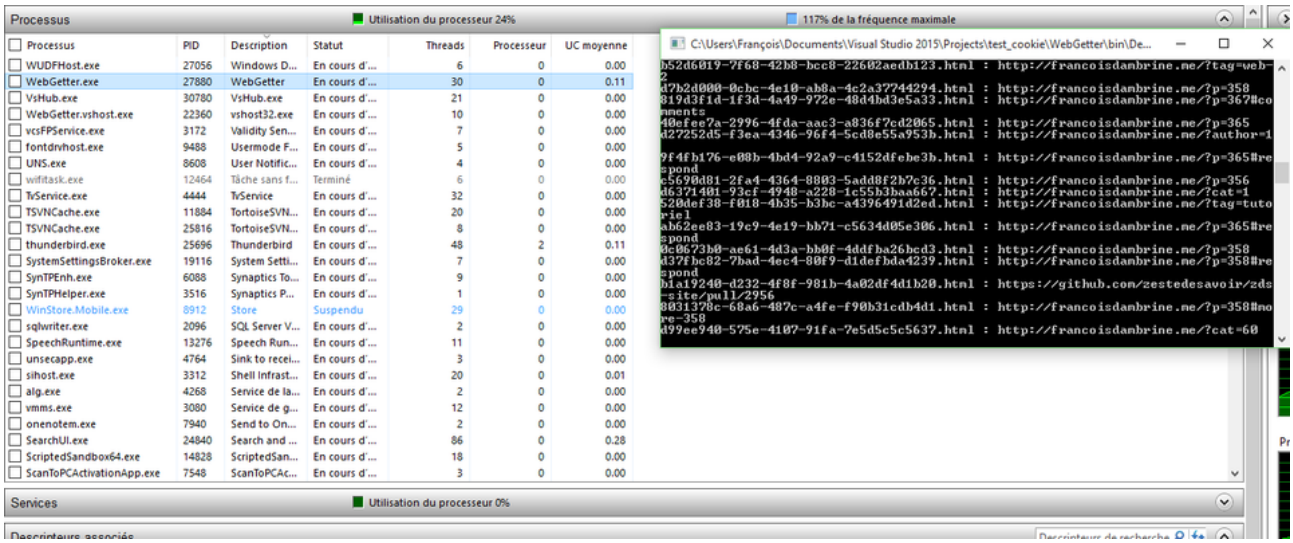

FIGURE 3. – Le gestionnaire de ressources nous indique le nombre de threads. 30 dans notre cas

Notre but n'est pas de faire un aspirateur complet de site web, juste de télécharger les pages actuelles et d'en tirer les liens et d'eux même les télécharger. Vous pourrez bien sûr ajouter vos propres amélioration pour faire un aspirateur MAIS, je tenais à vous rappeler que si vous désirez aspirer un site web, il est fortement recommandé d'en demander l'autorisation au propriétaire afin que vous ne mettiez pas en péril sa disponibilité.

3.0.2. Correction

Contenu masqué n°2

Si vous avez une version de visualstudio qui le supporte, vous pourrez observer sur le profileur que le processeur est vraiment peu utilisé par notre programme mais qu'il l'est plus souvent que si nous devions attendre à chaque fois le téléchargement de chaque page.

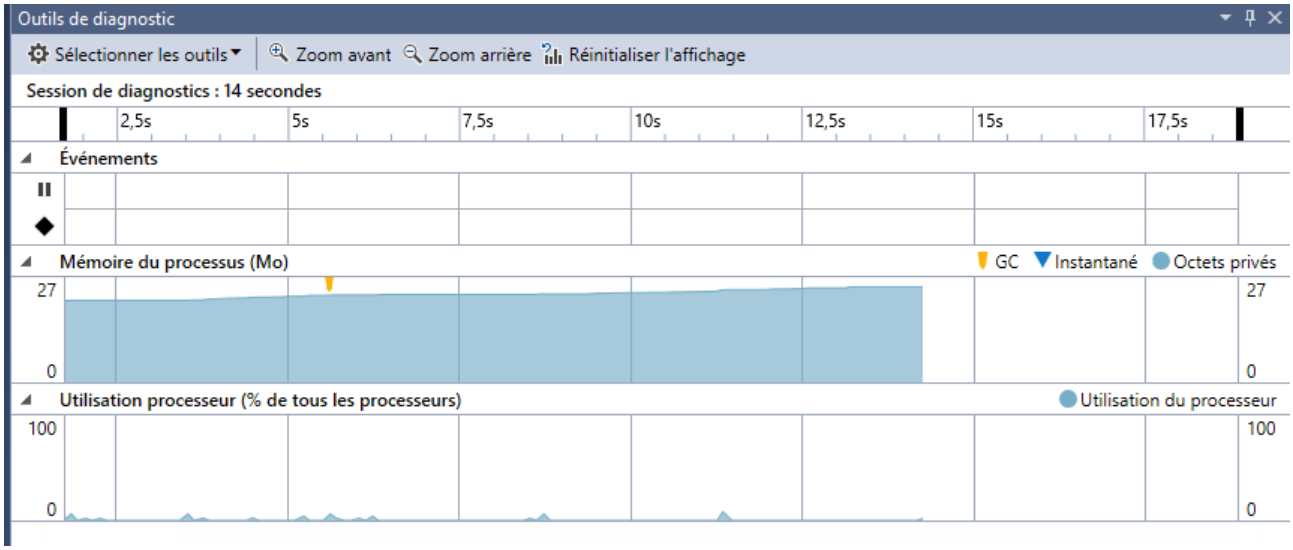

FIGURE 3. – Le profileur nous dit que le processeur attend beaucoup

3.1. Quelques améliorations

Comme je l'ai dit avant de proposer la correction vous pouvez tout à fait songer à aspirer un site web entier.

Pour cela il faudra donc être un peu plus efficace sur la manière de faire :

— il faudra faire en sorte que les liens vers l'extérieur du site ne soient pas pris en compte ;
— il faudra trouver un moyen de ne pas télécharger deux fois le même lien ;
— il faudra aussi télécharger les images et les ressources externes telles que les CSS et les stocker dans le bon sous dossier ;

3.2. Allons plus loin : la progression des tâches

Le TAP ne permet pas à priori d'ajouter le concept de "progression", dans le flot d'exécution des tâches. En effet, comme le TAP ne s'occupe que du concept de "tâche", ce que le programme sait faire avec async et await c'est démarrer la tâche, détecter qu'elle est terminée et nettoyer le contexte.

Pour s'occuper de la progression, il faudra *déléguer* cette fonctionnalité à une autre classe. Le Framework .NET vous propose alors une interface `IProgress<T>` pour vous assurer qu'à chaque fois que vous devez faire cette délégation elle est faite sur le même modèle.

Vous devrez donc choisir le type de données qui sert à notifier la progression. La pratique la plus courante au sein du framework .NET est d'implémenter un `IProgress<long>` qui vous permet de notifier une estimation numérique de l'avancée.

Par exemple, si vous êtes en train de lire un fichier, vous pouvez faire cette classe :

```
class AfficherLesProgresDeLecture: IProgress<long>{
    public DateTime StartDate{get; private set;}
    public AfficherProgresDeLecture(){
        StartDate = DateTime.Now();
    }
    public Report(long value){
        int seconds = (DateTime.Now() - StartDate).Seconds
        Console.out.WriteLine(value.ToString() + " octets lus en "
            + seconds.ToString() + "secondes");
    }
}
```

Ensuite il suffit d'appeler `await stream.ReadAsync(buffer, 0, 1024, new AfficherLesProgresDeLecture());`. Ainsi la méthode ReadAsync appellera la méthode Report à chaque fois qu'elle lit 1024 octets.

Notons que pour vous éviter à réimplémenter une classe complète mais juste la méthode Report, la classe `Progress<T>` existe et vous permet de préciser l'ensemble des méthodes à réaliser à chaque rapport du progrès grâce à l'évènement `OnReport`.

Je vous laisse explorer un peu l'API pour créer une barre de progression dans la ligne de commande ou pour mettre à jour le composant ProgressBar dans une application graphique.

Si vous avez besoin d'aide, n'hésitez pas à poser vos questions sur le forum avec les tag [async].

4. Partage de ressources

Lorsque l'on exécute un programme ligne après ligne, on ne se rend pas compte combien on se facilite la tâche en ce qui concerne les ressources.

Par exemple, revenons à notre recette.

Lorsque nous avions parallélisé nos tâches, nous avions d'un côté la mise en route du four et d'un autre le nettoyage de la cuisine.

Pour simplifier les choses j'avais juste dit qu'à chaque fois on ne faisait qu'attendre. Mais en fait ça va un peu plus loin.

Pour "faire cuire" disons que la tâche se dégage ainsi :

```
prendre le plat
ouvrir le four
mettre le plat dans le four
fermer le four
attendre 15 minutes
```

```
ouvrir le four
prendre le plat
enlever les cookies du plat
poser le plat dans l'évier pour nettoyage
```

tandis que la partie ”nettoyage” se détaille en :

```
nettoyer le plan de travail
pour chaque ustensile ou plat dans l'évier:
    nettoyer l'ustensile
        rincer l'ustensile
        essuyer l'ustensile
        ranger l'ustensile
```

? Du coup le plat à cookie, on le lave ou pas ?

En effet, que faisons nous ? On peut se rendre compte que notre plat pose problème : il est utilisé par les deux tâches. Je pense qu'il est évident pour tous qu'il ne peut à la fois être dans le four et être nettoyé. Alors, que faire ?

En fait il faudra modifier la seconde tâche pour dire :

```
nettoyer le plan de travail
pour chaque ustensile ou plat dans l'évier:
    attendre que l'ustensile soit disponible
    nettoyer l'ustensile
        rincer l'ustensile
        essuyer l'ustensile
        ranger l'ustensile
```

! La solution paraît facile dit comme ça, mais, elle peut mener à pas mal de problèmes, notamment un *dead lock* c'est à dire une coroutine A qui attend qu'une coroutine B libère une ressource. Pendant ce temps la coroutine B attend que A libère la même ressource.

Ce cas arrivera souvent quand vous voudrez partager une liste d'objets entre plusieurs Task pour que certaines consomme la liste alors que d'autres y ajoute des objets.

Je vous conseille fortement d'éviter de partager des ressources. Préférez exécuter d'abord toutes les tâches qui obtiennent des objets puis celles qui les consomme.

Par contre si vous n'y arrivez pas, lire la suite vous permettra de vous en sortir.

Pour attendre qu'une ressource soit ”disponible”, il existe globalement deux possibilités mises en place par le système d'exploitation :

— La sémaphore (qui peut être un domaine très complexe à comprendre)
— Le MutEx, abréviation de *Mutuellement Exclusif*.

L'un comme l'autre sont à envisager dans le cadre du *multi thread* et du *multiprocess*. Mais ils demandent des notions que ce cours n'a pas pour but d'aborder. Du coup, nous allons utiliser ce que .NET appelle `SemaphoreSlim` , qui est un objet qui permet de gérer les ressources des coroutines/Task.

Cet objet se trouve dans `System.Threading` et il se base sur deux compteurs :

— le premier est le compteur de ressources libre. A priori le nombre de ressource libre initial doit valoir 1 ou 0.
— le second est le compteur de ressource disponibles au maximum. A priori il sera toujours de 1.

L'utilisation se fait ainsi :

```
1         static void Main(string[] args)
2         {
3             Task t = RunAsync();
4             t.Wait();
5             Console.In.ReadLine();
6         }
7         public static async Task RunAsync()
8         {
9             using (SemaphoreSlim verrouRessource = new
                  SemaphoreSlim(0, 1))// on crée un sémaphore qui ne
                  peut libérer qu'une ressource mais qui la marque
                  comme occupée pour l'instant
10            {
11                Queue<int> ressourcePartage = new Queue<int>();
12                                  //on prépare les tâches
13                List<Task> taches = new List<Task>();
14                Console.Out.WriteLine("Start write");
15                taches.Add(EnvoieDansLaFile(ressourcePartage,
                      verrouRessource));
16                Console.Out.WriteLine("Start read");
17                taches.Add(LireLaFile(ressourcePartage,
                      verrouRessource));
18                verrouRessource.Release();//on peut commencer
19                await Task.WhenAll(taches.ToArray());
20            }
21        }
22        public static async Task EnvoieDansLaFile(Queue<int> file,
              SemaphoreSlim verrou)
23        {
24            for (int i = 0; i < 15; i++)
25            {
26                await verrou.WaitAsync();//on attendq que la
                      ressource soit libre
```

```
27                  file.Enqueue(i);//on utilise la ressource
28                  verrou.Release();//on dit qu'on a finit d'utiliser
                        la ressource pour l'instant
29                  System.Console.Out.WriteLine("On a ajouté " +
                        i.ToString());
30                  Thread.Sleep(100);//on attend histoire que
                        l'exemple soit utile
31              }
32          }
33
34          public static async Task LireLaFile(Queue<int> file,
                SemaphoreSlim verrou)
35          {
36              bool first = true;
37              await verrou.WaitAsync();//on va utiliser la ressource
38              while (file.Count > 0 || first)
39              {
40                  // pour éviter le cas où cette tâche serait
                        exécutée avant la première fois où on met un
                        entier
41                  if (file.Count > 0 && first)
42                  {
43                      first = false;
44                  }
45                  System.Console.Out.WriteLine(file.Dequeue());
46                  verrou.Release();//on enlève le verrou
47
                        System.Console.Out.WriteLine("on attend le prochain tour")
48                  Thread.Sleep(150);
49                  await verrou.WaitAsync();//on va utiliser la
                        ressource au prochain tour de boucle
50              }
51          }
```

Le résultat montre bien le phénomène :

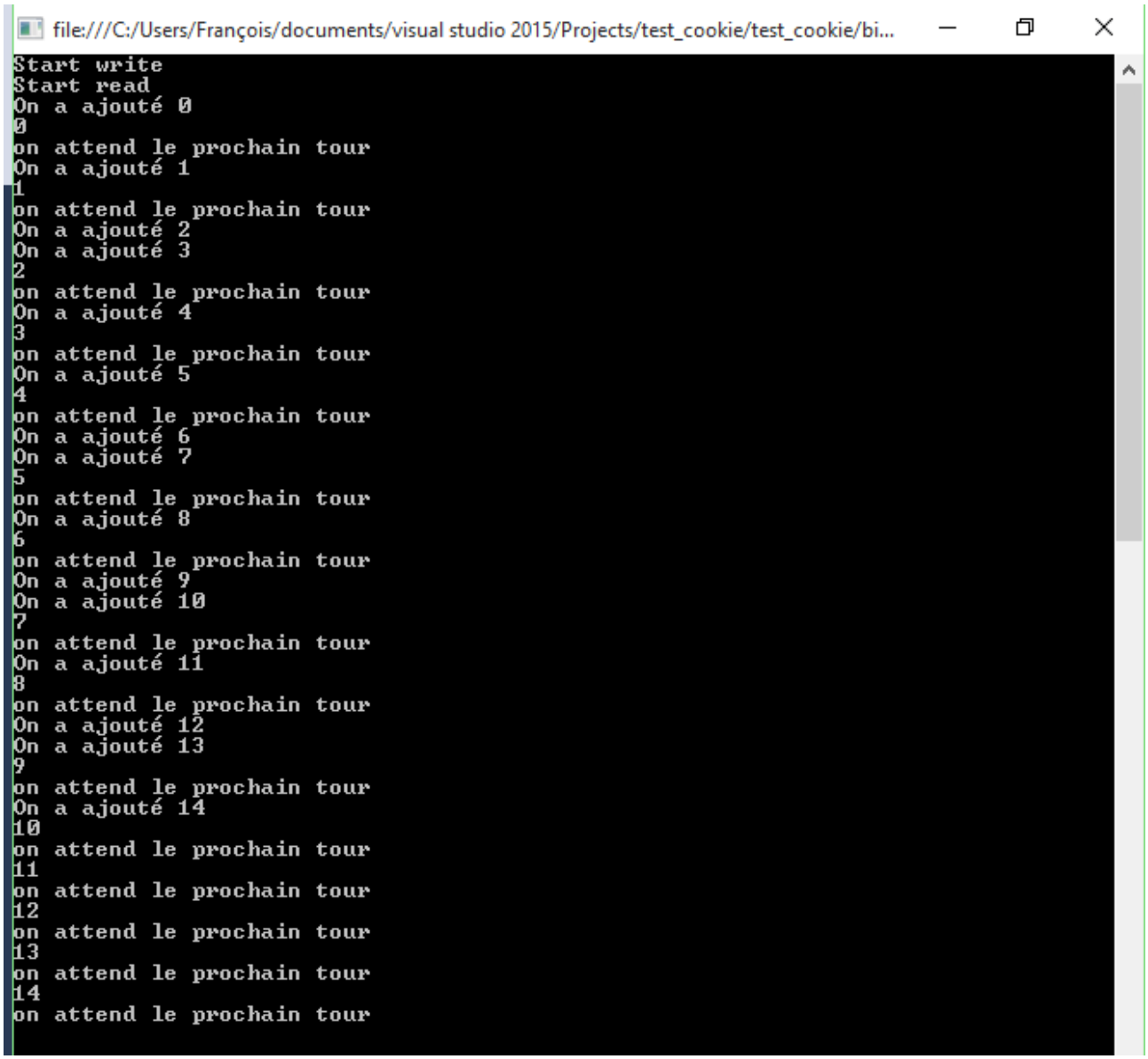

FIGURE 4. – Exécution en parallèle avec ressource partagée

Vous noterez que j'ai initié mon `SemaphoreSlim` avec aucune ressource disponible. Cela m'a permis de déclencher les tâches de manière à ce qu'elles exécutent toute la partie *préparatoire* puis bloque sur la ressource. Ensuite j'ai dit que la ressource était libre, ce qui a eu pour conséquence de lancer les tâches.

4.1. Le cas particulier de la fenêtre graphique

Si jusque là j'ai présenté les "ressources" comme quelque chose dans laquelle on peut lire ou écrire (ce qui peut être une liste, une imprimante, un objet connecté...), il existe un cas particulier qu'il me faut absolument vous présenter : l'interface graphique.

Si ce cas est si particulier, c'est qu'il est particulièrement verrouillé pour éviter de donner au logiciel un état instable.

De plus, votre interface graphique doit toujours être fluide. Et c'est là que les problèmes arrivent. En effet, si votre interface doit réagir à une action de l'utilisateur, disons un click, l'évènement est exécuté de manière synchrone au moteur de rendu de la fenêtre. C'est à dire que votre code sera exécuter par le même code qui affiche la fenêtre.

Cela pose donc un problème lorsque vous devez faire des calculs importants ou que vous devez *attendre*. Pendant le temps de l'exécution de votre évènement, l'interface sera figée.

Heureusement, l'objet Task et à async/await vous éviteront ce soucis : le thread principal continuera de s'exécuter de manière fluide.

Cas numéro 1 : la modification de l'interface se fait avant ou après la tâche, pas pendant

Pour gérer ce cas, vous n'aurez pas vraiment de nouveau concept à apprendre, vous serez peut être simplement heureux d'apprendre qu'un délégué peut tout à fait être async. Par exemple, si vous avez une tâche asynchrone à réaliser au click :

```
public async Task VotreTache(){
   await QuelquechoseDeLong();
}

private async void DoClick(object sender, EventArg args){ //votre
    délégué pour le click
    this.Button1.Text = "Before the task"
    await VotreTache();
    this.Button1.Text = "After the task"
}
```

Rien de très compliqué, on notera simplement que le listener d'événement est le seul cas où l'on tolère que votre fonction asynchrone retourne `void`.

Cas numéro 2 : la tâche elle-même modifie l'interface au cours de son exécution

Ce cas arrivera par exemple si votre tâche récupère des données de plusieurs sources différentes. Les affiche, puis les traite pour par exemple changer la couleur d'un indicateur ou l'état d'une barre de progression.

Il me faut alors dire que temporairement, le temps de changer l'interface, la tâche (ou une sous tâche) doit s'exécuter dans le thread de la GUI. Pour cela, nous allons utiliser la notion de *contexte d'exécution*.

Si les `Task` peuvent être démarrées dans un thread différent, ce n'est pas une nécessité. Vous pouvez leur demander, de s'exécuter dans le même thread que la tâche parente ou dans un contexte qui ne demande pas l'établissement de nouveaux thread.

A l'opposée, les parties de la tâche que vous désirez paralléliser peuvent être lancées dans une `threadpool`.

```
public async Task VotreTache(){

   await
       QuelquechoseDeLongQuiNeTouchePasALaGui().ConfigureAwait(continueOnCaptu
   await
       QuelquechoseDeLongQuiToucheALaGui().ConfigureAwait(continueOnCapturedCo
}

private async void DoClick(object sender, EventArg args){ //votre
    délégué pour le click
    this.Button1.Text = "Before the task"
    await
        VotreTache().ConfigureAwait(continueOnCapturedContext:true);//on
        continue sur la GUI
    this.Button1.Text = "After the task"
}
```

4.2. Cas numéro 3 : l'objet Task Chapitre 2

Vous connaissez peut être l'installateur de VisualStudio qui possède deux barres de progression : une pour le téléchargement et une pour l'installation.

Le principe de cet installateur est de vous faire gagner du temps en installant les fichiers déjà téléchargés et profiter de ce temps d'installation pour télécharger les autres.

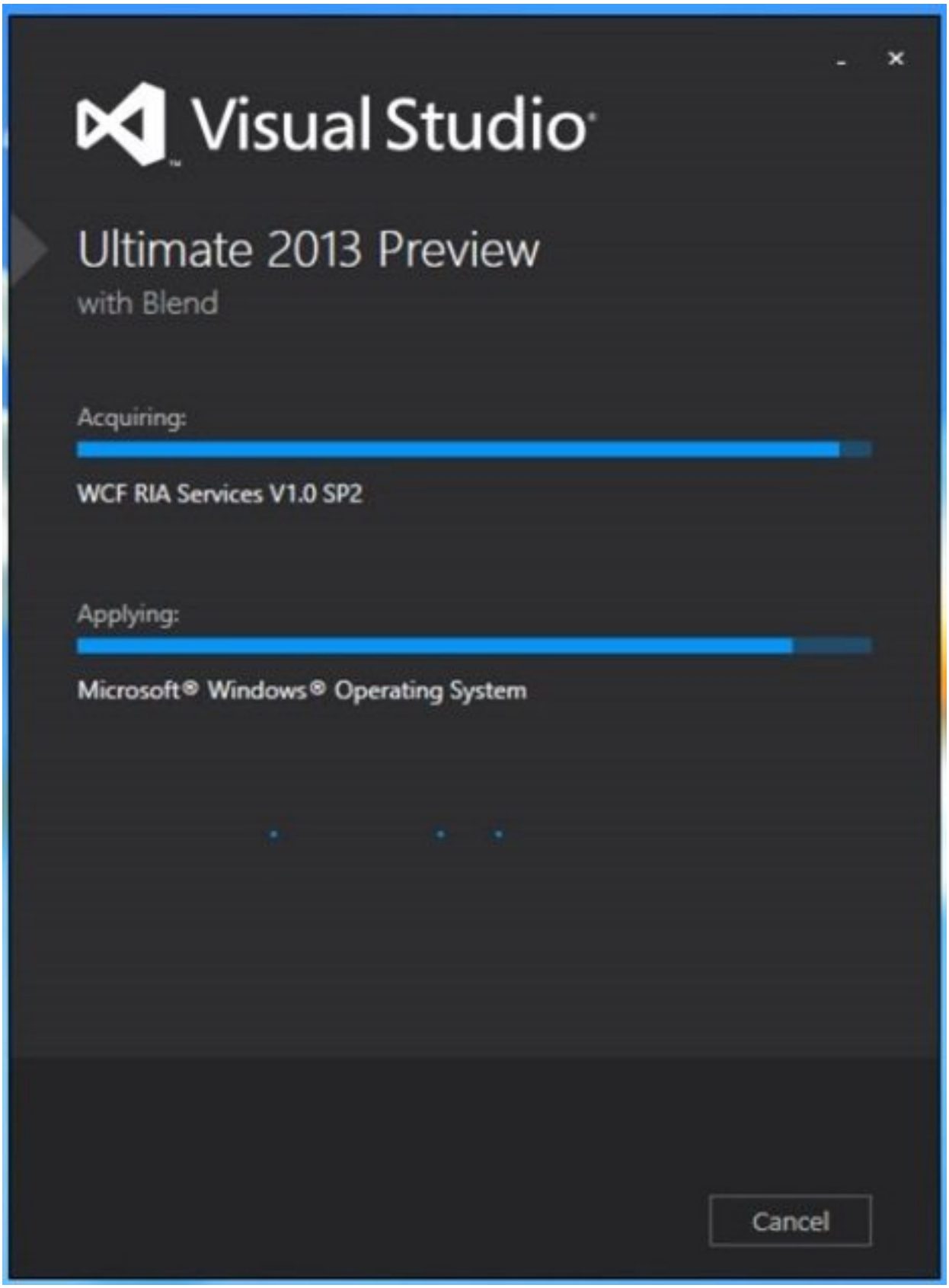

FIGURE 4. – L'installateur de VS2013

Si on devait représenter le code abstrait de cette fonctionnalité ça serait :

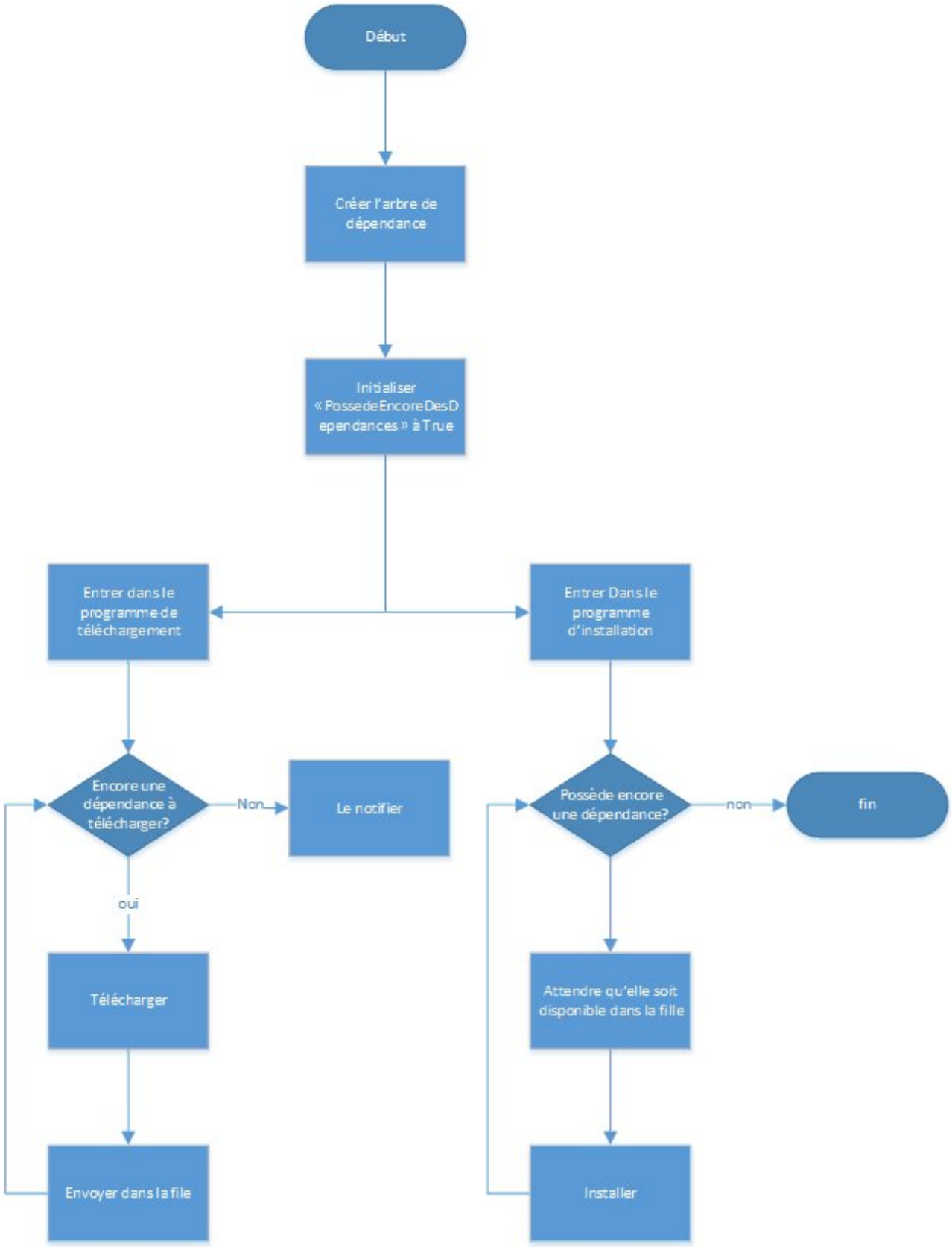

FIGURE 4. – Procédure d'installation de VS

On peut donc imaginer un système avec des événements. Mais comme finalement chaque tâche est très séquencée, on peut simplement dire :

— télécharge
— puis notifie la barre d'avancement
— puis attend que l'installation soit disponible
— puis installe
— puis notifie la barre d'avancement

et lancer ces tâches de manière parallèle.

L'objet Task possède une méthode qui exprime cela : ContinueWith qui a le bon goût de permettre à chaque `ContinueWith` de s'exécuter dans le contexte désiré (donc l'UI par exemple).

Pour simplifier les choses j'ai supposé que chaque objet "dépendance" représentait un ensemble de paquets à installer indépendant des autres (comme ça on n'a pas à gérer les conflits).

Le code final pourrait ressembler à ça :

```
1  public async Task DownloadAndInstall(List<Dependance> dptList)
2  {
3     List<Task<Dependance>> tasks = new List<Task<Dependance>>();
4     foreach(Dependance dpt in dptList)
5      {
6          Task<Dependance> downloadTask = Download(dpt).ContinueWith(
7                                      (downloadTask, dpt)=>{
                                           progressBarDownload.update(1);
8                                                  return dpt;},
9
                                           TaskScheduler.FromCurrentSynchronizationCon
10                                 ).ContinueWith((t, dpt)=>{
                                       InstallDependance(parentTask, dpt);
                                       return dtp;}
11                                 ).ContinueWith(
12                                     (installTask, dpt)=>{
                                           progressBarInstall.update(1);
                                           return dpt;},
13
                                           TaskScheduler.FromCurrentSynchronizationCon
14                                 );
15         tasks.Add(downloadTask);
16      }
17      await Task.WhenAll(tasks);
18 }
```

Vous noterez que la dépendance est passée de tâche en tâche et qu'à chaque fois la tâche parente aussi.

5. Les autres modèles de programmation

5.1. EventBased Asynchronous Pattern

Vous vous souvenez sûrement de la première utilité que j'ai donnée à la programmation asynchrone dans le premier chapitre : éviter de bloquer le programme lorsqu'il attend quelque chose.

Nous avons observé la manière la plus conseillée de faire lorsqu'on utilise C# : le task-based asynchron pattern. Mais une des premières méthodes qui a été historiquement développée dans le langage utilisait une philosophie différente directement basée sur la programmation événementielle.

En C# implémenter un évènement est très simple il suffit de déclarer dans notre classe `public event TypeEvenement OnEvenement;` puis d'y ajouter les actions qui doivent se passer lorsque l'événement est déclenché.

De ce fait tout un pan de la programmation asynchrone peut être pilotée à partir d'événements. C'est ce qu'on appelle l'EventBased Asynchron Pattern.

Le principe est de se baser sur une convention qui appellera les méthodes et les événements associés au fur et à a mesure. Les règles sont les suivantes :

1. Créer une méthode FaireQuelqueChose. Elle sera charger de faire la chose de manière synchrone ou bloquante
2. Créer la méthode FaireQuelqueChose**Async**
   — elle ne retourne rien
   — elle a les mêmes paramètres que FaireQuelqueChose
3. Créez un événement qui s'appelle FaireQuelqueChoseCompleted et qui a la signature que vous désirez par exemple `public event FaireQuelqueChoseCompletedHandler FaireQuelqueChoseCompleted;`
4. Le plus simple étant de définir FaireQuelqueChoseCompletedHandler comme ceci : `public delegate void FaireQuelqueChoseCompletedHandler (object sender, FaireQuelqueChoseCompletedEventArgs e);` (sans oublier de définir FaireQuelque-ChoseCompletedEventArgs)

Bref vous l'aurez compris, ça peut vite devenir long et répétitif à implémenter, tous les conseils se trouvent ici . Sachez juste que les BackgroundWorker du framework .NET fonctionnent selon ce principe.

##Asyncronous Programming Model

En complément du EAP, le framework .NET propose d'ajouter une autre manière de faire de l'Asynchrone à partir d'une interface nommée IAsyncResult.

Le principe est celui-ci :

— séparer la tâche en trois parties :
   — le lancement (BeginFaireQuelqueChose)
   — faire la chose
   — terminer et traiter le résultat (EndFaireQuelqueChose)

Ce patron de conception est détaillé ici et est présent dans les objets tels que HttpWebRequest comme vous avez pu le constater.

6. Conclusion

Nous voici donc à la fin de ce tutoriel sur l'utilisation de la programmation asynchrone avec C#. Souvenez vous de ces trois règles d'or :

— Lorsque vous commencez à faire de l'asynchrone, faites le de bout en bout ;
— Toujours retourner un `Task` ou un `Task<quelquechose>` sauf si vous créez un écouteur d'événement ;
— Configurez le contexte pour que par défaut il s'exécute sur plusieurs thread sauf dans le cas d'une tâche que touche à l'UI.

Contenu masqué

Contenu masqué n°1

Juste comme ça, voici la classe ingrédient, pas très bien codée :

```
class Ingredient
{
    public string Name { get; set; }
    public string Action { get; set; }
    public int Quantity { get; set; }
    public void Take()
    {
        Console.Out.WriteLine("J'ai pris " + Name);
    }

    public void UnPackage() {
        Console.Out.WriteLine(Action);
    }

    public void MesureQuantity()
    {
        Console.Out.WriteLine(Quantity);
    }

    public static Queue<Ingredient> GetRecette()
    {
        Queue<Ingredient> liste = new Queue<Ingredient>();
        liste.Enqueue(new Ingredient
        {
            Name = "farine",
```

Contenu masqué

```
                Quantity = 500,
                Action = "Rien de spécial"

            });
            liste.Enqueue(new Ingredient
            {
                Name = "chocolat pâtissier",
                Quantity = 500,
                Action = "Faire des morceaux"

            });
            liste.Enqueue(new Ingredient
            {
                Name = "beurre",
                Quantity = 250,
                Action = "Faire fondre"

            });
            liste.Enqueue(new Ingredient
            {
                Name = "Cassonade",
                Quantity = 200,
                Action = "Enlever les gros morceaux"

            });
            liste.Enqueue(new Ingredient
            {
                Name = "oeufs",
                Quantity = 2,
                Action = "Casser"

            });
            liste.Enqueue(new Ingredient
            {
                Name = "sucre vanillé",
                Quantity = 2,
                Action = "Rien de spécial"

            });
            liste.Enqueue(new Ingredient
            {
                Name = "sel",
                Quantity = 1,
                Action = "Prendre une pincée"

            });
            liste.Enqueue(new Ingredient
            {
                Name = "levure",
                Quantity = 1,
```

```
76                  Action = "Vider le sacher"
77
78              });
79              return liste;
80          }
81      }
```

Retourner au texte.

Contenu masqué n°2

```
1  using HtmlAgilityPack;
2  using System;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Linq;
6  using System.Net;
7  using System.Text;
8  using System.Threading.Tasks;
9  using System.Xml.Linq;
10
11 namespace WebGetter
12 {
13     class Program
14     {
15         static void Main(string[] args)
16         {
17             List<string> urls = new List<string>();
18             string[] _urls = {"http://francoisdambrine.me"};
19             urls.AddRange(_urls);
20             GetLinksThenDownloadPages(urls);
21             Console.ReadLine();
22         }
23         static async void
               GetLinksThenDownloadPages(IEnumerable<string> urls)
24         {
25             List<string> downloadedUrls = await GetLinks(urls);
26             await DownloadFilesFromLinks(downloadedUrls);
27         }
28         static async Task<List<string>>
               GetLinks(IEnumerable<string> startUrls)
29         {
30             List<Task<List<string>>> tasks = new
                   List<Task<List<string>>>();
31             foreach(string url in startUrls)
32             {
```

Contenu masqué

```
33                  tasks.Add(DownloadLinksFromUrlAsync(url));
34              }
35              return (from list in await
                    Task<List<string>>.WhenAll(tasks)
36                      from link in list
37                      select link).ToList();
38          }
39
40          static async Task<List<string>>
                DownloadLinksFromUrlAsync(string url)
41          {
42              HttpWebRequest request = WebRequest.Create(url) as
                    HttpWebRequest;
43              HttpWebResponse response = await
                    request.GetResponseAsync() as HttpWebResponse;
44              using (var streamReader = new
                    StreamReader(response.GetResponseStream()))
45              {
46                  string htmlCode = streamReader.ReadToEnd();
47                  HtmlDocument parsed = new HtmlDocument();
48                  parsed.LoadHtml(htmlCode);
49                  IEnumerable<string> aElement = from element in
                        parsed.DocumentNode.Descendants()
50                                                 where
                                                       element.Name.ToString().ToLo
                                                       == "a"
51                                                 where
                                                       element.Attributes["href"]
                                                       != null
52                                                 select
                                                       element.Attributes["href"].V
53                  return aElement.ToList();
54              }
55
56          }
57
58          static async Task WriteFileFromUrl(string url)
59          {
60              try {
61                  HttpWebRequest request = WebRequest.Create(url) as
                        HttpWebRequest;
62                  HttpWebResponse response = await
                        request.GetResponseAsync() as HttpWebResponse;
63                  Guid g = Guid.NewGuid();
64                  using (StreamReader streamReader = new
                        StreamReader(response.GetResponseStream()))
65                  {
66                      using (StreamWriter writer = new
                            StreamWriter(g.ToString() + ".html"))
67                      {
```

Contenu masqué

```
68                          await writer.WriteAsync(await
                                streamReader.ReadToEndAsync());
69                          Console.WriteLine(g.ToString() + ".html : "
                                + url);
70                      }
71                  }
72              }
73              catch(UriFormatException exception)
74              {
75                  Console.Error.WriteLine(exception.Message);
76              }
77              catch(WebException exception)
78              {
79                  Console.Error.WriteLine(url +
                        " was correct but not downloaded due to " +
                        exception.Message);
80              }
81          }
82
83          static async Task
                DownloadFilesFromLinks(IEnumerable<string> list)
84          {
85              List<Task> tasks = new List<Task>();
86              foreach(string url in list)
87              {
88                  tasks.Add(WriteFileFromUrl(url));
89              }
90              await Task.WhenAll(tasks);
91          }
92      }
93  }
```

Listing 1 – correction

Retourner au texte.

Liste des abréviations

GUI Graphical UserInterface. 15