PARENT-CHILD TASK RELATIONSHIPS IN THE .NET FRAMEWORK 4

Ling Wo, Cristina Manu
Parallel Computing Platform
Microsoft Corporation

Abstract

This document provides an in-depth explanation on parent-child task relationships offered by the Task Parallel Library as in the .NET Framework 4. This includes the behavioral changes implied by being a parent or child task in terms of task completion, task waiting, as well as task cancellation. In addition, it also points out a few common oversights and provides general guidelines on when to take advantage of this feature.

Last Updated

11/30/2009

TABLE OF CONTENTS

INTRODUCTION

One of the goals of the Task Parallel Library (TPL), part of the .NET Framework 4, is to provide support for structured parallelism, a programming model that clearly defines the scope of parallel code. This is enforced through several constructs:

1. High-level abstractions such as Parallel.For, Paralle.Invoke, and Parallel.ForEach methods.
2. Fine-grained Tasks that employ parent-child relationships.

While the Parallel methods are structured by definition, Tasks need to opt-in to the structured approach. Structured tasks have many advantages, but their use can also lead to subtle coding errors. This document explains, in detail, the nature of Task parent-child relationships, the benefits of using them, and common usage errors to be avoided.

PARENT-CHILD RELATIONSHIPS BETWEEN TPL TASKS

In order to establish parent-child relationships between Tasks, two conditions need to be met:

1. One task (the task destined to be the child) needs to be created during the execution of another task (the task destined to be the parent).
2. The task destined to be the child must be created using the AttachedToParent option.

```
// Sample: both child_1 and child_2 tasks below will be attached to the parent task
Task parent = Task.Factory.StartNew (()=>
{
        Task child_1 = Task.Factory.StartNew(()=>
        {
                Console.WriteLine("The child_1 says hello!");

        }, TaskCreationOptions.AttachedToParent);

        Task child_2 = child_1.ContinueWith((t)=>
        {
                Console.WriteLine("The child_2 says hello!");

        }, TaskContinuationOptions.AttachedToParent);

});
```

By default, all tasks are created "detached", meaning they are not attached to any parent task. The AttachedToParent option is there in order to support structured parallelism.

THE LIFE OF A PARENT TASK

For a Task to become a parent Task, it must execute code during its lifetime, and that code must create at least one task with the AttachedToParent option. The existence of such child tasks has an effect on the lifecycle of the parent Task, exposed through the Task's Status property. The Status property returns a TaskStatus enumeration value, which will be one of the following values:

- *Starting states:*

    - Created
    - WaitingForActivation
- *Intermediate states:*
    - WaitingToRun
    - Running
    - WaitingForChildrenToComplete
- *Final states*
    - RanToCompletion
    - Canceled
    - Faulted

Out of the eight states listed above, there is only one TaskStatus value unique to parent tasks: WaitingForChildrenToComplete. A task will be in this state when:

- It has finished its own execution, and
- It has at least one child task that has not yet completed (i.e. reached a final state).

```
// Sample: Demonstrate parent task's WaitingForChildrenToComplete status
Task parent = Task.Factory.StartNew(() =>
{
        Task child = Task.Factory.StartNew(() =>
        {
                //intentionally delay the task so that it finishes after its parent
                Thread.Sleep(3000);

        }, TaskCreationOptions.AttachedToParent);

        Console.WriteLine("Parent execution body is done.");
});

//monitor the parent State, one of which will be the WaitingForChildrenToComplete
while (!parent.IsCompleted)
{
        Console.WriteLine("*** Current parent state: {0} ***", parent.Status);
        Thread.Sleep(300);
}

Console.WriteLine("Done - Parent State: {0}", parent.Status);
```

This means that a parent task will not reach a final state until itself and all its children finish execution. As with any task, only after a parent task has transitioned into a final state will its IsCompleted, IsCanceled, IsFaulted, and Exception properties be updated to reflect its completion. Also, only then will any code waiting for the task to complete succeed. In the example below, parent.Exception will remain null even though it and one of its children have thrown exceptions.

```
// Sample: Demonstrate parent task's Exception will not be updated
//         when it is in WaitingForChildrenToComplete state
Task parent = Task.Factory.StartNew(() =>
{
        Task child_1 = Task.Factory.StartNew(() =>
        {
                Console.WriteLine("Child1 Exception: Exception 1.");
```

```
                throw new Exception("Child1 Exception: Exception 1.");

        },TaskCreationOptions.AttachedToParent);

        Task child_2 = Task.Factory.StartNew(() =>
        {
                //intentionally delay the throwing from the second child task
                for (int i = 0; i < 3; i++)
                {
                        Console.WriteLine("Child 2 is still working ...");
                        Thread.Sleep(1000);
                }
                Console.WriteLine("Child2 Exception: Exception 2.");
                throw new Exception("Child2 Exception: Exception 2.");

        }, TaskCreationOptions.AttachedToParent);

        Console.WriteLine("Parent Exception: Exception 3.");
        throw new Exception("Parent Exception: Exception 3.");
});

//monitor the parent Exception property
while (null == parent.Exception)
{
        Console.WriteLine("*** Current parent state: {0} ***", parent.Status);
        Thread.Sleep(500);
}

//parent final status should be Faulted
Console.WriteLine(
   "Done - parent Exception is not null. Parent Status: {0}", parent.Status);
```

In .NET 4, a parent keeps a count of its remaining children, and a child maintains a reference back to its parent. When a child completes, it notifies its parent (decrementing the count). This causes the parent to transition to a final state upon the completion of its last child.

When a new child task is created (using AttachedToParent), it first looks to see if it has a parent. This is done by examining the current thread of execution to see if another task is currently running; that running task would be the new parent. Because of this check, creating a task from a new thread, as below, will not result in a parent-child relationship.

```
// Sample: fork task in a separate thread from the parent broke parent-child relationship
Task task1 = Task.Factory.StartNew(() =>
{
        Console.WriteLine("Outer Task: Thread: {0}, Id: {1} ",
                                Thread.CurrentThread.ManagedThreadId,
                                Task.CurrentId);

        Thread t = new Thread (()=>
        {
                //this task will not have any parent
                //even if it is created as AttachedToParent
                Task task2 = Task.Factory.StartNew(() =>
                {
                        //delay task2's execution
```

```
                        Thread.Sleep(2000);
                        Console.WriteLine("Inner Task: Thread: {0}, Id: {1} ",
                                          Thread.CurrentThread.ManagedThreadId,
                                          Task.CurrentId);

                    }, TaskCreationOptions.AttachedToParent);
            });
            t.Start();
});

//this Wait below will finish before the print out from inside task2,
//since task2 is not a child of task1, thus no need to be waited by task1
task1.Wait();
```

When a parent task finishes executing and has children that have not yet completed, it transitions into the WaitingForChildrenToComplete state. Once all children have completed, the parent will exit the WaitingForChildrenToComplete state and transition into its final state. The "Explicit Waiting vs. Implicit Waiting" section, below, will discuss how the final state of a parent can be influenced by child tasks that have faulted.

EXPLICIT WAITING VS. IMPLICIT WAITING

As explained above, a parent task won't reach a final state (one of TaskStatus.RanToCompletion, TaskStatus.Canceled or TaskStatus.Faulted) until it and all of its attached child tasks (those created with the AttachedToParent option) finish executing. This behavior of a parent task waiting for its child tasks before signaling its own completion is referred to as "implicit waiting", which carries many subtle differences from "explicit waiting", used to denote actions taken in the code to wait for a Task to complete, such as calling Task.Wait, Task.WaitAll, or Task<TResult>.Result. In this section, we will talk about the differences between implicit and explicit waiting in detail.

EXCEPTION HANDLING

OBSERVING FAULTED TASK

If a Task faults (ends in the TaskStatus.Faulted final state), its exception needs to be observed. Otherwise, that exception will, by default, cause the application to crash (behavior typical for unhandled exceptions in .NET since version 2.0). In .NET 4, this behavior is accomplished by throwing the unhandled exception on the finalizer thread upon finalizing an unobserved task object. A faulted task's exception is "observed" when any of the following happens:

1. A wait operation is performed on the task that causes it to propagate its exception. Waiting with a timeout or a cancellation token may not propagate an exception, since the task may not complete by the time the Wait times out or the cancellation occurs. Similarly, the Task.WaitAny method does not propagate unhandled task exceptions, and thus does not cause a faulted task to be observed.

2. Its Exception property is accessed after the task reaches a final state. Accessing the Exception before the task completes does not observe this task.

3. The task is a child task, such that its exception is automatically propagated by default to its parent. This observes the exception for the child, but it also causes the parent to become faulted; the parent will separately need to be observed.

```
//Sample: Demonstrate faulty child task's exception automatically propagated to its parent
Task parent = Task.Factory.StartNew(() =>
{
        Task child = Task.Factory.StartNew(() =>
        {
                throw new Exception("Faulting");

        }, TaskCreationOptions.AttachedToParent);
});

// parent needs its own observation due to the exception
// propagated from the attached child task
try
{
        parent.Wait();
}
catch (AggregateException ae)
{
        Console.WriteLine("Parent caught {0}", ae);
}
```

EXCEPTION PROPAGATION

When a task ends in the Faulted state, its exceptions are accessible through its Exception property. This property returns an AggregateException containing any unhandled exceptions from the task itself, as well as the AggregateException (as returned from the child.Exception property) for each of its faulted children. When a wait operation is performed on a faulted task that causes it to propagate (i.e. throw) its exceptions, such an AggregateException is also thrown, again containing both this task's exceptions and any child exceptions. When using Task.WaitAll to wait on multiple tasks, if any of the tasks faulted, an AggregateException will be thrown containing the exceptions that would have been contained in the AggregateException thrown from waiting on each Task individually.

```
// Sample: Task.WaitAll propagating an aggregated exception
Task[] tasks = new Task[11]; // one parent task plus 10 children tasks
ManualResetEventSlim mres = new ManualResetEventSlim (false); // signal all tasks created
tasks[0] = Task.Factory.StartNew(() =>
{
        for (int i = 1; i < 11; i++)  // child task indexed from i=1
        {
                tasks[i] = Task.Factory.StartNew(() =>
                {
                        throw new Exception("Child Faulting");

                }, TaskCreationOptions.AttachedToParent);
        }
        mres.Set();   // by now all tasks should have been created

        throw new Exception("Parent Faulting");
});

mres.Wait();

try
{
        Task.WaitAll(tasks);
```

```
}
catch (AggregateException ae)
{
        // expect 21 inner exceptions, because it aggregates
        // - tasks[0]: The inner exceptions contained in the tasks[0].Wait(), i.e.
        //      1. Its own exception: new Exception("Parent Faulting")
        //      2. The AggregateException (as returned from the Exception property) for
        //       each attached child task, i.e.
        //                AggregateException(new Exception("Child Faulting"))
        // - tasks[1-10]: The inner exceptions contained in the tasks[i].Wait(), i.e.
        //      1. Just the child's own exception: new Exception("Child Faulting")
        Console.WriteLine("Inner Exceptions Count = {0}", ae.InnerExceptions.Count);

}
```

PREVENTING PROPAGATION OF EXCEPTIONS TO PARENTS

When a child task completes, its exceptions are automatically propagated to its parent, if it has one. When a child task's exception is aggregated to its parent, both the parent and the child task will transition to the Faulted state. This observes the exception for the child task, but not for the parent. The parent task will now carry all unhandled exceptions from its children, and like any other faulted task, the parent task must be observed.

This automatic exception aggregation is avoided if the parent, rather than any other task, explicitly propagates and handles its children's exceptions in its body.

```
// Sample: Suppress the implicit child exception being aggregated to the parent
Task parent = Task.Factory.StartNew(() =>
{
        Task child = Task.Factory.StartNew(() =>
        {
                throw new Exception("Faulting");

        }, TaskCreationOptions.AttachedToParent);

        // because the child is Faulted, the wait below will throw;
        // failure to take care of this new AggregateException will leave the parent
        // with another unhandled exception and end the parent with Faulted final state
        try { child.Wait(); }
        catch(AggregateException)
        {
                // Child's exception will now not propagate to the parent
        }
});

// the parent task will complete successfully with no exception
parent.Wait();
```

In cases like the above, where the child task is waited on inside its parent, the child's exception is not propagated. Therefore, only the child task will be Faulted, while its parent will end in the RanToCompletion state (assuming no other reason for the parent to fault, such as another child faulting or the parent itself throwing an unhandled exception).

```
// Sample: The child is waited via task.Wait but not inside of the parent
//         the exception still will be aggregated to the parent's exception
Task parent = Task.Factory.StartNew(() =>
{
        Task child = Task.Factory.StartNew(() =>
        {
```

```
                throw new Exception ("Faulting");
        },
        TaskCreationOptions.AttachedToParent);

        // chain a continuation to observe the child task
        child.ContinueWith(_ =>
        {
                // because the child is Faulted, the wait below will throw
                try { child.Wait(); }
                catch(AggregateException) { }
        });
});

// since the observation on faulty child happens in a different task (in the continuation)
// than the parent task, it does "not" count, so the exception is still aggregated to
// its parent task which needs to be observed
try
{
        parent.Wait();
}
catch (AggregateException ae)
{
        Console.WriteLine("Parent caught {0}", ae);
}
```

CANCELLATION

Cancellation of structured tasks follows the same pattern as that of unstructured tasks. For more information about cancellation in TPL please refer to the "Reference" section.

A couple of short examples are illustrated below.

It is worth mentioning several key points:

- In the following code examples, the parent Task and all of its children will finish in the "Canceled" state. The aggregate exception thrown while waiting on the parent will contain only one InnerException of type "TaskCancelledException" which indicates that the parent itself was canceled. In other words, the implicit exception aggregation logic, discussed above, does not apply to a canceled child, only faulty child's exception could be propagated to its parent.

```
//Sample: both parent and child are canceled and acknowledge cancellation
CancellationTokenSource cts = new CancellationTokenSource();
CountdownEvent cde = new CountdownEvent(2); // countdown both the parent and the child

TaskFactory factoryWithToken = new TaskFactory(cts.Token);

//this thread will do the cancellation
Thread cancelThread =  new Thread(() =>
{
        cde.Wait();//only cancel after both parent and child tasks started execution
        cts.Cancel();
});
cancelThread.Start();
```

```
//start one parent and its child, they both use the same cancellation token;
//and in case of cancellation, both will acknowledge it
Task parent = factoryWithToken.StartNew(() =>
{
        cde.Signal();

        factoryWithToken.StartNew(() =>
        {
                cde.Signal();

                // wait for the token to be cancelled
                cts.Token.WaitHandle.WaitOne();

                //acknowledge cancellation via throwing an OCE with same token
                throw new OperationCanceledException("child was cancelled", cts.Token);

        }, TaskCreationOptions.AttachedToParent);

        // wait for the token to be cancelled
        cts.Token.WaitHandle.WaitOne();

        //acknowledge cancellation via throwing an OCE with same token
        throw new OperationCanceledException("parent was cancelled", cts.Token);

});

try
{
        parent.Wait();
}
catch (AggregateException ex)
{
        // expect only 1 TaskCancelledException, which is for the parent
        Console.WriteLine("Inner Exceptions Count = {0}", ex.InnerExceptions.Count);
}
finally
{
        cancelThread.Join();
        Console.WriteLine("Parent Status : {0}", parent.Status);
}
```

- If only the children finishes in the “Canceled” state and the parent does not acknowledge cancellation, the parent state will be “RanToCompletion”, as shown in the second example below.

```
//Sample: only child task acknowledges cancellation
CancellationTokenSource cts = new CancellationTokenSource();
CountdownEvent cde = new CountdownEvent(2); // countdown the parent and the child

TaskFactory factoryWithToken = new TaskFactory(cts.Token);

//this thread will do the cancellation
Thread cancelThread = new Thread(() =>
{
        cde.Wait();//only cancel after both parent and child tasks started execution
        cts.Cancel();
});
```

```
cancelThread.Start();

//start one parent and its child, they both use the same cancellation token;
//but in case of cancellation, only child will acknowledge it
Task parent = factoryWithToken.StartNew(() =>
{
        cde.Signal();

        factoryWithToken.StartNew(() =>
        {
                cde.Signal();
                // wait for the token to be cancelled
                cts.Token.WaitHandle.WaitOne();

                //acknowledge  cancellation via throwing an OCE with same token
                throw new OperationCanceledException("child was cancelled", cts.Token);

        }, TaskCreationOptions.AttachedToParent);
});

try
{
        // no need to take care of any exception out of this parent.Wait()
        // since parent will be in RanToCompletion, rathan than Faulted final state
        parent.Wait();
}
finally
{
        cancelThread.Join();
        Console.WriteLine("Parent Status : {0}", parent.Status);
}
```

COMMON OVERSIGHTS

After we've talked about the special features provided through the parent -child relationship, it is worth a special section to point out a few of the most common mistakes and/or confusions with using child tasks.

CONSISTENT BEHAVIOR OF ATTACHEDTOPARENT

The AttachedToParent option has the same effect on all tasks, regardless of how they are created. This includes:

- Task constructor (.ctor)
- TaskFactory.StartNew
- ContinueWith
- TaskFactory.ContinueWhenAll/Any
- TaskFactory.FromAsync
- TaskCompletionSource<TResult> Task

As long as a Task is given the AttachedToParent option at creation time and is created in the context of another task, it will be attached to its parent, the task under which this attached child starts its life. One slightly special case (still conforming to the above statements) is that the TaskExtensions.Unwrap method determines whether to create the new Task with AttachedToParent based on whether the task being unwrapped is itself

AttachedToParent: if the wrapper was created as AttachedToParent, the unwrapped task will also be attached and they both will have the same parent task.

```
//samples of different attached tasks
Task parent = Task.Factory.StartNew(() =>
{
        // child_1 created via StartNew and is attached
        Task child_1 = Task.Factory.StartNew(
                () => { }, TaskCreationOptions.AttachedToParent);

        // child_2 (Task<T>) created via constructor and is attached
        Task child_2 = new Task<int>(() => 123, TaskCreationOptions.AttachedToParent);
        child_2.Start();

        // detached task
        Task detached = Task.Factory.StartNew(() => { });

        // child_3 created via ContinueWith and is attached
        Task child_3 = detached.ContinueWith(
                (t) => { }, TaskContinuationOptions.AttachedToParent);

        // child_4 created via ContinueWhenAll and is attached
        Task child_4 = Task.Factory.ContinueWhenAll(
                new Task[] {detached}, (tasks) => { },
                TaskContinuationOptions.AttachedToParent);

        // child_5 created via FromAsync and is attached
        Task child_5 = Task.Factory.FromAsync(
                detached, (ia) => { }, TaskCreationOptions.AttachedToParent);

        // child_6 implied by TaskCompletionSource (tcs.Task) and is attached
        var tcs = new TaskCompletionSource<int>(TaskCreationOptions.AttachedToParent);
        tcs.SetResult(123);

        // child_7 created via StartNew and is attached
        Task<Task> child_7 = Task<Task>.Factory.StartNew(
                () => Task.Factory.StartNew(() => { }),
                TaskCreationOptions.AttachedToParent);

        // child_8 created via Unwrap
        // and carries the same attached/detached option as its wrapper child_7
        Task child_8 = child_7.Unwrap();

});

// since all inner tasks from child_1 to child_8 are attached, wait on the
// parent task below will not complete until all the attached children
// reaching their final state
parent.Wait();
```

As per the comments above, child_1 through child_8 are all attached to the parent task, even though they are created via different approaches. They will all exhibit behaviors associated with any child task –parents will implicitly wait on all child tasks before claiming itself to be completed, and if a child task ends in the Faulted state, its exceptions will be propagated to its parent unless parent explicitly propagated and dealt with the child's exceptions in the parent's body.

Care should be taken when using attached Tasks. For example, note the following:
- Possible deadlock as in the case if we remove the child_2.Start() in the above example
- Possible crash as in the case if we replace tcs.SetResult(123) with tcs.SetException(some_exception)

PARENT-CHILD RELATIONSHIP ESTABLISHED AT CREATION TIME

Another common oversight is that a task does not need to be started in order to become a child, since the parent-child relationship is established at child creation time rather than child start time. In the case below, the child task will be attached to the task of parent1 not of the parent2.

```
// Sample: Identify the right parent-child relationship
Task child = null;
ManualResetEventSlim mres = new ManualResetEventSlim(false); // to signal child created
Task parent1 = Task.Factory.StartNew(() =>
{
        //child task will be a child of parent1
        child = new Task(() =>
        {
                Console.WriteLine("The child says hello!");

        }, TaskCreationOptions.AttachedToParent);

        mres.Set();
        child.Wait();
});

Task parent2 = Task.Factory.StartNew(() =>
{
        mres.Wait();//wait for the child to be created before starting it
        child.Start();
});

//just some wait for tasks to be launched
Thread.Sleep(2000);
```

It is critical to understand the correct parent-child relationship between tasks, due to the entailed behavior changes on the parent task in terms of task completion, task exception proportion, as well as task cancellation per our discussions above.

OBSERVING FAULTED TASK EXCEPTIONS VS. PROPAGATING CHILD TASK EXCEPTIONS

It has already been explained that a faulted task needs to be observed; otherwise, its unhandled exception will be re-thrown during finalization causing your application to crash. We've also pointed out that waiting on a faulted child task explicitly in its parent's delegate will suppress propagation of the child's exception to the parent. These two concepts are not exactly equivalent to each other.

```
// Sample: A fault child task is observed but not has its exception propagated.
Task parent = Task.Factory.StartNew(() =>
{
```

```
        Task child = Task.Factory.StartNew(() =>
        {
                throw new Exception ("Faulting");

        }, TaskCreationOptions.AttachedToParent);

        // Waiting on a task using IAsyncResult.WaitHandle will not throw,
        // as compared with waiting on a task using Task.Wait
        ((IAsyncResult)child).AsyncWaitHandle.WaitOne();

        // Child task is "observed", since we access its Exception after it completes
        Console.WriteLine("Faulty Child's Exception – {0}", child.Exception);
});

// Even if the child task is "observed" inside the parent, its exception has not been
// propagated to its parent. Thus the parent will still get an aggregated exception from
// its faulty child and end with Faulted (i.e. needs to be observed)
try
{
        parent.Wait();
}
catch (AggregateException ae)
{
        Console.WriteLine("Parent caught {0}", ae);
}
```

DEBUGGING OF ATTACHEDTOPARENT TASK

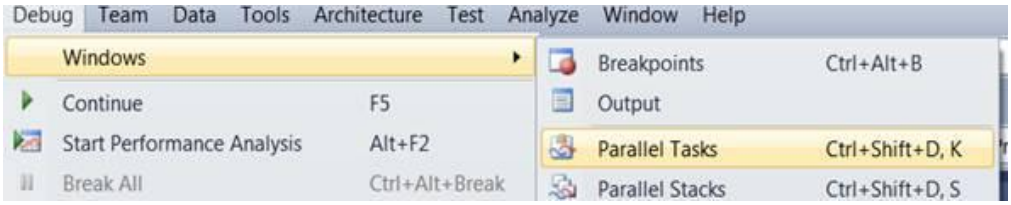
Parent – child relationships can be illustrated in debugging sessions. There are two ways of identifying the parent of a child while debugging a scenario involving attached tasks.

1. Using the new "Parallel Tasks" Debug window.
2. Expanding the Raw View for a child so that we can see the reference to its parent (if any).

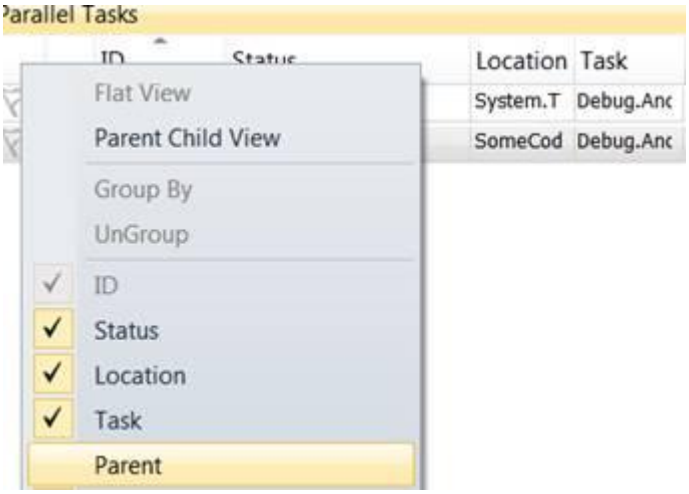
```
//samples: Demonstrate debugging parent / child tasks
Task parent = Task.Factory.StartNew(() =>
{
        Task child = Task.Factory.StartNew(() =>
        {
                //set a breakpoint at the shaded line below
                Console.WriteLine("Child Id : {0} says hello!", Task.CurrentId);

        }, TaskCreationOptions.AttachedToParent);

        //do some work
        Thread.Sleep(2000);
});
parent.Wait();
```

- Parallel Stacks

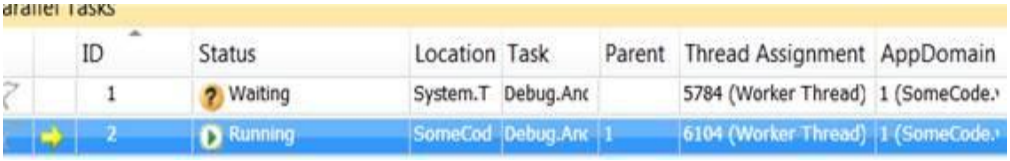
    a. Open ParallelStacks window:



b. Configure the window to show parent-child relationship; check the "Parent" check box as below:



c. See the parent Id of the current child



- Child's parent Reference.

a. In a "Watch" window expand the RawView for the child. Look for m_parent reference.

GENERAL GUIDELINES

Finally, here comes the most anticipated question – under what circumstance should I use AttachedToParent and under what other cases should I not? The answer varies based on your own application; however, below are a few guidelines to help determine appropriate usage.

You should consider using AttachedToParent when:

- You rely on the structured parallelism in that a parent task should not complete until its attached children complete. For example, in the case of fork-join design pattern, you could fork all the tasks as attached children, and choose to wait on the parent task only rather than performing a join on each forked task. As

long as this approach is applied consistently, it can lead to fewer blocked threads during the processing of the application, which can in turn yield better performance.

- You prefer a single location to look for unhandled exceptions when quite a few subtasks might all end with certain exceptions, especially when those exceptions are anticipated. This is to take advantage of the implicit exception aggregation from child to parent, so that the parent task holds all exceptions of its children, and you don't have to look around on each individual task.

You should consider not using AttachedToParent when:

- Your asynchronous work being done is primarily fire-and-forget.

CONCLUSION

The AttachedToParent option is an advanced TPL feature that needs to be opted into with caution. You should not utilize this option unless you are fully aware of its benefits and consequences. In addition, the rule of thumbs listed above are all based on the assumption that you do have the need to control each subtask individually; otherwise, TPL does provide other high-level APIs such as Parallel.Invoke which can also serve your purpose of ensuring structure parallelism and centralized exception aggregation.

REFERENCES

1. "Task Wait and Inlining"

   http://blogs.msdn.com/pfxteam/archive/2009/10/15/9907713.aspx

2. "TaskStatus"

   http://blogs.msdn.com/pfxteam/archive/2009/08/30/9889070.aspx

3. "Ways to create task"

   http://blogs.msdn.com/pfxteam/archive/2009/06/03/9691796.aspx

4. "Walkthrough Debugging a Parallel Application"

   http://msdn.microsoft.com/en-us/library/dd554943(VS.100).aspx

5. "Parallel Debugging"

   http://msdn.microsoft.com/en-us/magazine/ee410778.aspx