

ジョブスケジューラー(タスクシステム)

ノーマル(最初から)

ジョブスケジューラーって何？から
簡単な実装までの話

ハード(応用)

より実践的な実装へ
前提条件などの
実行順序を制御する

ベリハード(より発展的)

更に進化する
別スレッドも使った
ジョブシステムへ

ジョブスケジューラーってなに？

ジョブスケジューラー(タスクシステム)とは？

複数の処理を一括で起動や終了、その監視や状態の通知など

使用者が追加した複数のジョブを管理するためのシステムです。

使うとどんなメリットがあるの？

- ・ 事前に実行タイミングや前後関係を指定し、ジョブとして登録しておくことで、
システムがジョブを自動で管理し、実行などを行ってくれるようになる。
- ・ 複数スレッドを使うジョブスケジューラーにすることで更に最強になる。

※想像しづらいと思うのでちょっとずつ解説します。

まずは、ジョブを使わないコードを見てみる 1

```
11 int main()  
12 {  
13     while (true)  
14     {  
15         // 更新を必要しないコンポーネントまで更新処理が呼ばれてしまう。  
16         for (auto component : components)  
17         {  
18             component->Update();  
19         }  
20     }  
21 }
```

#仮のコード例

左のように、ポリモーフィズムを使って更新処理を書くと、
更新を必要としないコンポーネントやシステムまで
更新処理が呼びだされてしまうため、
プロジェクトが大きくなると関数呼び出しによる
オーバーヘッドが無視出来ないほどになってしまいます。

まずは、ジョブを使わないコードを見てみる 2

#仮のコード例

```
48
49 int main()
50 {
51     while (true)
52     {
53         UpdateInput();
54         UpdateAudio();
55         UpdateGame();
56         UpdateScene();
57         UpdateGameObject();
58         UpdateComponent();
59         UpdatePhysics();
60         UpdateRender();
61     }
62 }
```

また左のように普通に処理を書くと、
処理を記述している部分がドンドン増えていくため、
処理の実行順序や実行条件を考慮したり、
呼び出し元を考慮したりしていくと、
コードの保守が非常に難しくなるのが分かります。

#プラスα

もしイベントシステムについて理解している人だったら多分、
「Updateというイベントにしてしまったらいいのでは？」と
思うかもしれません。その通りです。間違いありません。
ですが、前後関係や実行条件を考慮出来る点が違います。

↑ 場所や人にもよりますが、前後関係や実行条件が実装されて、
初めてジョブスケジューラーと呼ぶ場合もありますので注意が必要です。

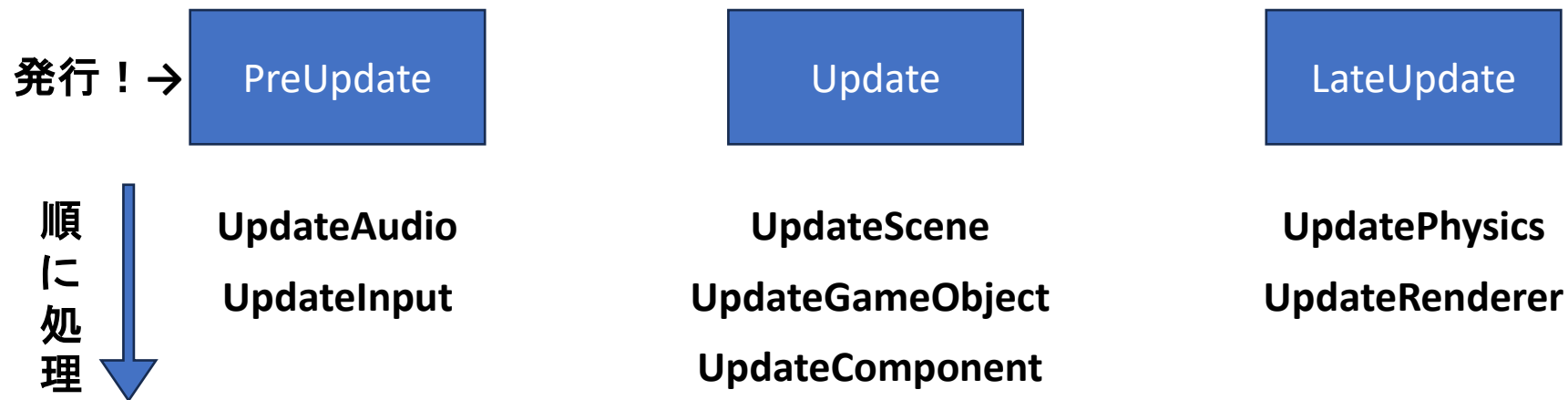
じゃあ、ジョブを使うとどんな感じになるの？（解説）

今回は異なる3つの更新タイミングを元に考えていきます。

PreUpdateは、更新の一番最初に呼ばれるものです。（先の例ではInputやAudioとかが該当します）

Updateは通常の更新タイミングです。（先の例ではScene、GameObject、Componentの更新が該当します）

LateUpdateは必ず更新の最後に呼び出されるものです。（先の例では物理や描画などですかね）



上記では先ほどの例から、ジョブを割り振って登録された後のイメージです。

システムはPreUpdate、UpdateやLateUpdateなどの実行が発行されると、登録されたジョブを処理していきます。

じゃあ、ジョブを使うとどんな感じになるの？1（実装例）

```
1  #include "JobSystem.h"
2
3  int main()
4  {
5      // ジョブを登録
6      Job job;
7      job.SetFunc([](float deltaTime) { printf("call method\n"); });
8
9      // 実行タイミングを指定。
10     JobSystem::Get().Get().AddJob(FunctionType::Update, &job);
11
12
13     while (true)
14     {
15         JobSystem::Get().Execute(0.01f, FunctionType::Update);
16     }
17 }
```

#完成イメージ

※「あれ、イベントシステムと一緒にじゃね？」と思う方いるかと思いますが、2ページ前に解説していますのでそちらを参照ください。

仮コードでは、Updateで実行されるジョブを一つ登録し、ゲームループの更新タイミングで実行しています。

簡単な例ではありますが、先ほどの例のような
**大量のUpdateを処理する時でも、設定さえすれば
簡単に処理を追加、変更できる**ことが分かります。
そのため、コードの維持がめっちゃ楽になります。

#プラスα

今回は更新タイミングだけしか設定項目がありませんが、
処理の実行優先度、前後関係、別スレッド動作などの
項目が増えていくとめっちゃくちゃ使いやすくなります。

↑後で解説します。

じゃあ、ジョブを使うとどんな感じになるの？2(実装例)

#簡単な実装例

```
1  #pragma once
2  #include <functional>
3
4  class Job
5  {
6  public:
7
8      void SetFunc(std::function<void(float)> func)
9      {
10         m_func = func;
11     }
12
13     void Execute(float deleteTime)
14     {
15         m_func(deleteTime);
16     }
17
18 private:
19     std::function<void(float)> m_func;
20
21 };
```

Jobクラスでは、
ジョブとして関数をセット出来るようにしており、
Execute関数が呼び出されると処理を実行します。

#プラスα

「Jobクラスに設定項目を持たせないの？」と
疑問に思う人もいるかもしれませんが、
このクラス内で設定変数を持たせていないのは、
このクラスはジョブを実行する役割のクラスだからです。

じゃあ、ジョブを使うとどんな感じになるの？3(実装例)

```
1  #pragma once
2  #include <vector>
3  #include "Job.h"
4
5  class JobContainer
6  {
7  public:
8
9      void Register(Job* job)
10     {
11         Jobs.push_back(job);
12     }
13
14     // JobSystem内で範囲for使用するために定義。
15     auto begin() { return Jobs.begin(); }
16     auto end() { return Jobs.end(); }
17
18 private:
19
20     std::vector<Job*> Jobs;
21 };
```

#簡単な実装例

JobContainerクラスでは、
ジョブリストを保持し、登録、解除等の処理を行います。

コードの下部分にあるbegin関数、end関数は、
ジョブシステム側で範囲for文を使えるようにしているため。

#プラスα

begin、end関数で戻り値がautoとなっていますが、
これは、型推論で型が見つかったものなら何でも使えます。
使い過ぎには注意ですが今回のような場合は非常に便利。

↑「何故使いすぎがよくない？」と思うかもしれませんが呼び出し側ならいいですが、
他の人が見て何の型を返すのか、パッと見で分からないのは結構問題だと思います。

じゃあ、ジョブを使うとどんな感じになるの？4(実装例)

#簡単な実装例

```
1  #pragma once
2  #include <map>
3  #include "Job.h"
4  #include "JobContainer.h"
5
6  enum class FunctionType
7  {
8      PreUpdate, // 更新の一番最初に呼ばれる
9      Update,    // 普通の更新タイミング
10     LateUpdate, // 更新中の最後で呼ばれる
11 };
12
13 class JobSystem
14 {
15 public:
16
17     static JobSystem& Get()
18     {
19         static JobSystem instance;
20         return instance;
21     }
22
23     void AddJob(FunctionType type, Job* job)
24     {
25         JobContainerMap[type].Register(job);
26     }
27
28     void Execute(float deltaTime, FunctionType type)
29     {
30         // 関数が登録されていれば処理する。
31         for (auto job : JobContainerMap[type])
32         {
33             job->Execute(deltaTime);
34         }
35     }
36
37 private:
38     std::map<FunctionType, JobContainer> JobContainerMap;
39 };
40
```

JobSystemクラスでは、
更新タイミングごとのコンテナマップを保持しており、
AddJob関数で渡された更新タイミングの設定から、
一致するコンテナへジョブの登録を行います。
そしてExecute関数でジョブを実行しています。

#プラスα

今回はExecute関数が呼び出されると、
登録された全てのジョブを実行していますが、
キューに登録にして遅延実行でジョブを処理することで、
前後処理など考慮することが出来ると思います。

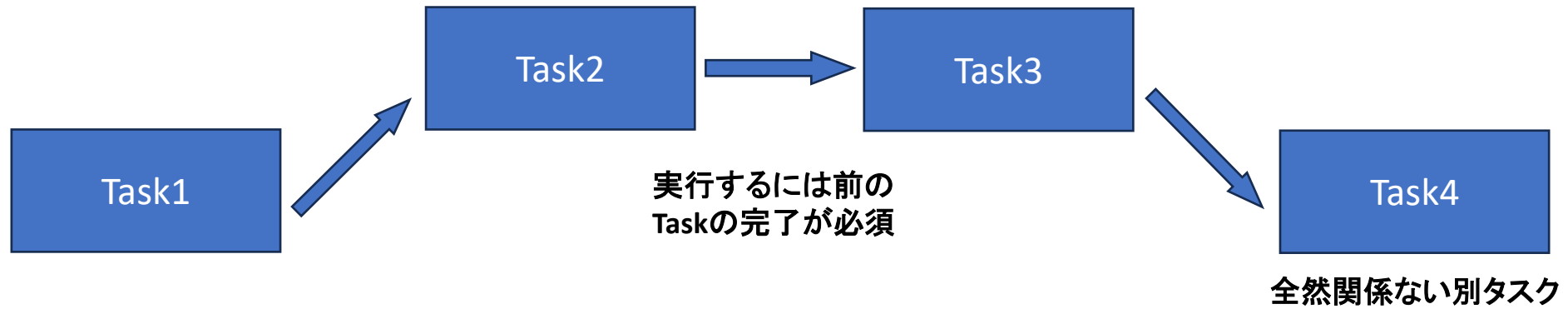
↑後で解説します。

ジョブシステム(応用)

1. 前後関係など実行順序を考慮するための解説
2. 前後関係など実行順序を考慮したジョブシステムへ

実行条件と違ってどうやって考慮するの？1（解説）

今回は処理の前後関係を考慮していく方法を解説します。



簡単ではありますが、

Task2はTask1が、同様にTask3はTask2の処理が完了しないと絶対に実行されない設定である時に、

Task4は全く関係のないTaskのためTask1の前に処理されても大丈夫なはずです。

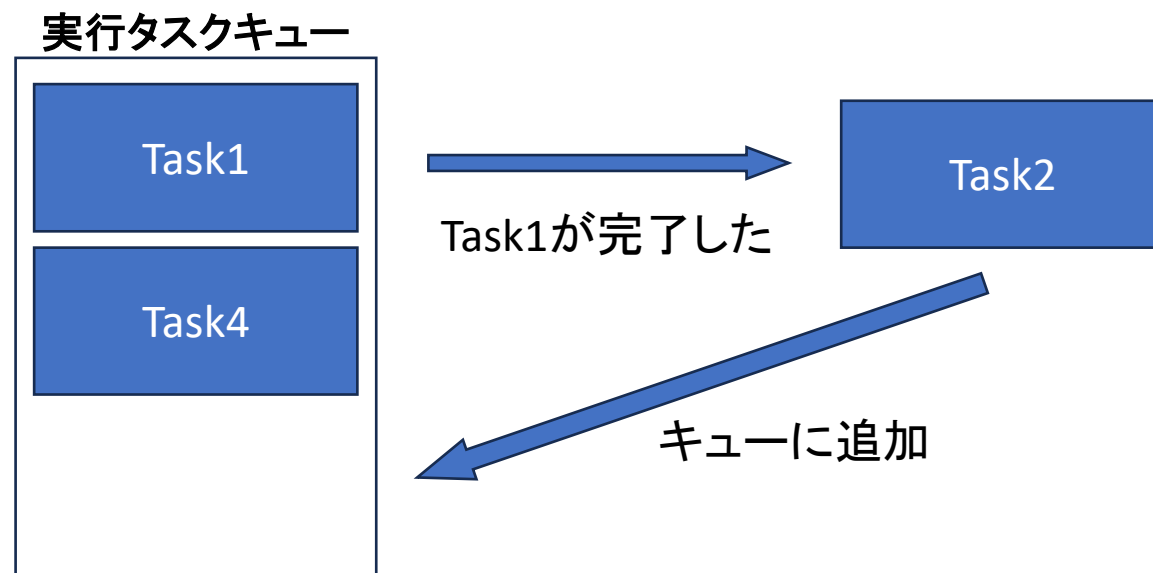
#プラスα

「あれ簡単だな？」と思うかもしれませんが、

前提条件だけでなく実行優先度や非同期処理など考慮していくとちょっと凝ったことをしなければいけません。

実行条件とかってどうやって考慮するの？2(解説)

ここからはキューを使って前提条件を作成していきたいと思います。



先ほどの例であった、Task2の部分の動作フローです。

Task1が完了して初めてTask2が実行キューに追加される点が重要な点です。

これで前提条件を考慮出来、実行条件を設定出来るようになると思います。

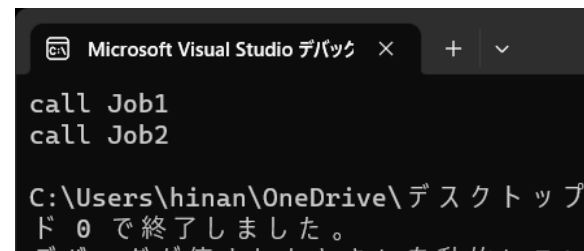
前提条件を考慮するジョブシステム1（実装例）

```
1 #include "JobSystem.h"
2
3 int main()
4 {
5     Job job1;
6     Job job2;
7
8     // ジョブを登録
9     job1.SetFunc([](float deltaTime) { printf("call Job1\n"); });
10    job2.SetFunc([](float deltaTime) { printf("call Job2\n"); });
11
12    // job2 の前に job1 が処理されるようにする。
13    job2.AddPrerequisite(&job1);
14
15    // 実行タイミングを指定。
16    // ここで job2 を先に登録して前提条件が正常に動作しているか確認している。
17    JobSystem::Get().AddJob(FunctionType::Update, &job2);
18    JobSystem::Get().AddJob(FunctionType::Update, &job1);
19
20    //while (true)
21    {
22        JobSystem::Get().Execute(0.01f, FunctionType::PreUpdate);
23        JobSystem::Get().Execute(0.01f, FunctionType::Update);
24        JobSystem::Get().Execute(0.01f, FunctionType::LateUpdate);
25    }
26 }
27
28
29
30
31
32
33
```

#完成イメージ

仮コードでは、
ジョブを二つ用意、前提条件を付けて登録して、
Updateのタイミングで実行しているテストです。

登録順を逆にしていることで、
本来はジョブ2が先に実行されるはずが、
前提条件が考慮されるので「call job1」が先に呼ばれる
という仕組みのものになっています。



```
Microsoft Visual Studio デバッグ × + ▾
call Job1
call Job2

C:\Users\hinan\OneDrive\デスクトップ\
ド0で終了しました。
デバッグが停止したときに自動的にコンソール
```

前提条件を考慮するジョブシステム2(実装例)

#簡単な実装例

```
1  #pragma once
2  #include <functional>
3  #include <vector>
4
5  class Job
6  {
7  public:
8
9      // ジョブの登録用関数
10     void SetFunc(std::function<void(float)> func);
11
12     // 前提条件を追加するための関数
13     void AddPrerequisite(Job* job);
14
15     // ジョブキューへの登録を試みる。
16     // 前提条件が設定されている場合はキューへ登録されない。
17     void TryAddQueue();
18
19     // ジョブの実行用関数
20     void Execute(float deleteTime);
21
22 private:
23
24     // 前提条件の登録用関数
25     void SetupPrerequisite();
26
27     // 前提条件が一つ完了するごとに呼ばれるイベント関数
28     void PrerequisiteCompleted();
29
30     // 自分自身をジョブキューへ登録する。
31     void AddQueue();
32
33 private:
34
35     // ジョブ処理
36     std::function<void(float)> m_func;
37
38     // ジョブ終了時イベント
39     std::vector<std::function<void()>> m_completeEvents;
40
41     // 前提条件となるJob一覧
42     std::vector<Job*> m_prerequisites;
43
44     // 残り完了待ち、前提条件数
45     int m_numPrerequisites;
46
47     };
```

まずJobクラスの変更点から、

前提条件となるジョブを持つ配列(m_prerequisites)、
ジョブの終了を通知するイベント(m_completeEvents)、
そして完了待ちの残り前提条件数(m_numPrerequisites)
が変数で追加されています。

その他、関数もいくつか増えていますが
これは関数の実装の所で解説します。

#プラスα

今回はサンプルなのでジョブの実行や前提条件等の設定を
全て実装していますが、クラス分けするともっと良くなります。

前提条件を考慮するジョブシステム3(実装例)

```
1  #include "Job.h"
2  #include "JobSystem.h"
3
4  void Job::SetFunc(std::function<void(float)> func)
5  {
6      m_func = func;
7  }
8
9  void Job::AddPrerequisite(Job* job)
10 {
11     m_prerequisites.push_back(job);
12 }
13
14 void Job::TryAddQueue()
15 {
16     if (m_prerequisites.size() == 0)
17     {
18         // 前提条件が無ければ直ぐにジョブをキューに登録する。
19         AddQueue();
20     }
21     else
22     {
23         // 前提条件があれば完了イベントに登録する。
24         SetupPrerequisite();
25     }
26 }
27
28 void Job::Execute(float deleteTime)
29 {
30     m_func(deleteTime);
31
32     // 登録された完了イベントを発行する。
33     for (auto completeEvent : m_completeEvents)
34     {
35         if (completeEvent)
36         {
37             completeEvent();
38         }
39     }
40     m_completeEvents.clear();
41 }
```

#簡単な実装例

Jobクラスのメンバ関数の実装側では、
特にTryAddQueue関数とExecute関数が大事です。

TryAddQueue関数では、前提条件が設定されていれば
キューへ登録せずにジョブ完了イベントに登録しています。

Execute関数では、ジョブの完了時にイベントとして
登録されたイベントへ通知を出しています。

#プラスα

もし鋭い方なら、
「TryAddQueue関数をいじったら非同期にもできる？」と
思うかもしれません。後で解説しますが、
キューの登録先を変更すれば良いのではないのでしょうか？

前提条件を考慮するジョブシステム4(実装例)

#簡単な実装例

```
44
45
46 void Job::SetupPrerequisite()
47 {
48     // 前提条件となるジョブへ自分のイベントを登録している。
49     for (auto prerequisite : m_prerequisites)
50     {
51         prerequisite->m_completeEvents.push_back([this]()
52         {
53             PrerequisiteCompleted();
54         });
55     }
56
57     // 前提条件数を保存
58     m_numPrerequisites = m_prerequisites.size();
59 }
60
61 void Job::PrerequisiteCompleted()
62 {
63     --m_numPrerequisites;
64     if (m_numPrerequisites == 0)
65     {
66         // 前提条件が全て完了するとキューに登録する
67         AddQueue();
68     }
69 }
70
71 void Job::AddQueue()
72 {
73     JobSystem::Get().AddQueue(this);
74 }
```

Jobクラスの残りのメンバ関数の実装です。

SetupPrerequisite関数は、
前提条件として設定したジョブに完了イベントの追加
を行っています。

そして完了イベントでは、
残り完了待ち数が0になれば、キューへ追加しています。

前提条件を考慮するジョブシステム5(実装例)

```
19 static JobSystem instance;  
20 return instance;  
21 }  
22  
23 void AddJob(FunctionType type, Job* job)  
24 {  
25     JobContainerMap[type].Register(job);  
26 }  
27  
28 void AddQueue(Job* job)  
29 {  
30     jobQueue.push(job);  
31 }  
32  
33 void Execute(float deleteTime, FunctionType type)  
34 {  
35     // 順にキューへの登録を試みます。ここでは、  
36     // 前提条件がないもののみキューへ追加されます。  
37     for (auto job : JobContainerMap[type])  
38     {  
39         job->TryAddQueue();  
40     }  
41  
42     // 追加された順にジョブを実行します。  
43     while (jobQueue.size())  
44     {  
45         auto job = jobQueue.front();  
46         jobQueue.pop();  
47  
48         job->Execute(deleteTime);  
49     }  
50 }  
51  
52 private:  
53     std::map<FunctionType, JobContainer> JobContainerMap;  
54  
55     // 実行待ちジョブキュー  
56     std::queue<Job*> jobQueue;  
57  
58 };
```

#簡単な実装例

そしてメインのJobSystemクラスの変更点です。

変数では、実行キューが追加されてます。

Execute関数の先頭で前提条件の無いジョブだけを、
実行キューへ追加しています。

前提条件先を含む、全てのジョブを処理すれば終了です。

#プラスα

先ほどのプラスαで書いてましたが、非同期処理にする場合、Execute関数で処理完了待ちのブロック処理を追加するのと、タスクスレッドを作成し、別キューの実装をすれば良いです。

↑後で解説します。

ジョブシステム（より発展的）

1. 複数スレッドを使ったジョブシステムへ

前提条件を考慮するジョブシステム1（実装例）

```
1 #include "JobSystem.h"
2
3 int main()
4 {
5     Job job1;
6     Job job2;
7
8     // ジョブを登録
9     job1.SetFunc([](float deltaTime) { printf("call Job1\n"); });
10    job2.SetFunc([](float deltaTime) { printf("call Job2\n"); });
11
12    // job2 の前に job1 が処理されるようにする。
13    job2.AddPrerequisite(&job1);
14
15    // 実行タイミングを指定。
16    // ここで job2 を先に登録して前提条件が正常に動作しているか確認している。
17    JobSystem::Get().AddJob(FunctionType::Update, &job2);
18    JobSystem::Get().AddJob(FunctionType::Update, &job1);
19
20    // 別スレッドで動作させる。(前提条件を考慮してるので絶対に[call Job1]の後に呼ばれる)
21    job2.SetThreadSafe(true);
22
23    // MainLoop開始前のおまじない
24    JobSystem::Get().InitSystem();
25
26    //while (true)
27    {
28        JobSystem::Get().Execute(0.01f, FunctionType::PreUpdate);
29        JobSystem::Get().Execute(0.01f, FunctionType::Update);
30        JobSystem::Get().Execute(0.01f, FunctionType::LateUpdate);
31    }
32
33    // MainLoop開始後のおまじない
34    JobSystem::Get().TermSystem();
35 }
```

#完成イメージ

仮コードでは先ほどまでのサンプルに、
別スレッドで動作させる設定(SetThreadSafe関数)をして、
別スレッドでタスク処理を行っています。

後追加されたのは、
ジョブシステムの初期化と終了くらいです。

Job1は通常スレッドで、Job2の処理は別スレッドで
実行されるため、「AnyThread!!」が間に実行されています。



```
Microsoft Visual Studio デバッグ × + ▾
call Job1
Any Thread!!
call Job2

C:\Users\hinan\OneDrive\デスクトップ\ド0で終了しました。
デバッグが停止したときに自動的にコンソールを開く。
```

前提条件を考慮するジョブシステム2(実装例)

```
44 // 前提条件となるJob一覧
45 std::vector<Job*> m_prerequisites;
46
47 // 残り完了待ち、前提条件数
48 int m_numPrerequisites;
49
50 // trueの場合に別スレッドの実行キューに追加する
51 bool m_isThreadSafe;
52 };
```

#簡単な実装例

まずJobクラスの変更点から、

別スレッドでの実行を許可する設定(m_isThreadSafe)が、変数で追加されています。

```
56 }
57
58
59 // 前提条件数を保存
60 m_numPrerequisites = m_prerequisites.size();
61
62
63 void Job::PrerequisiteCompleted()
64 {
65     --m_numPrerequisites;
66     if (m_numPrerequisites == 0)
67     {
68         // 前提条件が全て完了するとキューに登録する
69         AddQueue();
70     }
71 }
72
73 void Job::AddQueue()
74 {
75     JobSystem::Get().AddQueue(this, m_isThreadSafe);
76 }
```

#簡単な実装例

関数の変更点はAddQueue関数の所で、
スレッドの設定を引数として渡すだけです。

#プラスα

「何でスレッドの設定を引数で渡しているの？」と疑問に
思うかもしれませんが、前にも書いてますが後々、
実行とデータクラスを分ける時に楽かなと思っています。

↑そこまでは実装しないので分かりませんが、もっと良い方法はあると思います。

前提条件を考慮するジョブシステム3(実装例)

#簡単な実装例

```
14
15 class JobSystem
16 {
17 public:
18
19     static JobSystem& Get()
20     {
21         static JobSystem instance;
22         return instance;
23     }
24
25     // 初期化、終了用の関数
26     void Initialize();
27     void DeInitialize();
28
29     // ジョブの登録用関数
30     void AddJob(FunctionType type, Job* job);
31
32     // ジョブを実行待ちキューへ追加する関数
33     void AddQueue(Job* job, bool isThreadSafe);
34
35     // 実行関数
36     void Execute(float deleteTime, FunctionType type);
37
38 private:
39
40     // 別スレッド用のジョブ実行用関数
41     void ThreadLoop();
```

そしてメインのJobSystemクラスの変更点です。
変更点が多いので分割して説明していきます。

まず関数定義から、

別スレッドの起動と終了用に、初期化、終了関数を用意。

別スレッドでキューを実行するためのThreadLoop関数

後は、AddQueue関数の引き数を追加しています。

#プラスα

今回は作成も考慮もしていませんが、
別スレッドの待ちをする関数なども必要になると思います。

↑ そうじゃないと別スレッドの処理が終了する前に、次のExecute関数が実行されてしまいます。

前提条件を考慮するジョブシステム4(実装例)

#簡単な実装例

```
31
32 // ジョブを実行待ちキューへ追加する関数
33 void AddQueue(Job* job, bool isThreadSafe);
34
35 // 実行関数
36 void Execute(float deleteTime, FunctionType type);
37
38 private:
39
40 // 別スレッド用のジョブ実行関数
41 void ThreadLoop();
42
43 private:
44
45 std::map<FunctionType, JobContainer> m_jobContainerMap;
46
47 // 実行待ちジョブキュー
48 std::queue<Job*> m_jobQueue;
49
50 // 別スレッドの実行待ちジョブキュー
51 std::queue<Job*> m_anyThreadJobQueue;
52
53 // 別スレッドでThreadLoop関数を実行する用
54 std::thread m_thread;
55
56 // 排他制御用(別スレッドを使う際にほぼ必ず必要)
57 std::mutex m_mutex;
58
59 // ThreadLoopで実行キューに追加されるまで待機するためのクラス
60 std::condition_variable m_conditionVar;
61
62 // 別スレッドで実行されているか?
63 bool m_isRunning;
64
```

JobSystemクラスの変数定義の部分の変更点です。

別スレッド用の実行キュー(m_anyThreadJobQueue)、
別スレッドでジョブを実行するスレッド(m_thread)、
スレッド間の排他制御を行う(m_mutex)、
キューの追加までスレッドを待機させる(m_conditionVar)、
別スレッドの実行状況を保持する(m_isRunning)、
が変数で追加されています。

#プラスα

排他制御について詳しくは説明ませんが、
別のスレッド同士が同時に同じ変数へアクセスした際に、
発生するバグを対象するものだと思います。

前提条件を考慮するジョブシステム5(実装例)

#簡単な実装例

```
1  #include "JobSystem.h"
2
3  void JobSystem::Initialize()
4  {
5      // 起動！（これはm_threadの起動より前に呼ぶ必要がある）
6      m_isRunning = true;
7
8      // 別スレッド用ジョブループを開始します。
9      m_thread = std::thread(&JobSystem::ThreadLoop, this);
10 }
11
12 void JobSystem::DeInitialize()
13 {
14     // ThreadLoop関数に対して終了を通知する
15     std::unique_lock<std::mutex> lock(m_mutex);
16     m_isRunning = false;
17     lock.unlock();
18
19     // 待機中の可能性があるため起動させる。
20     m_conditionVar.notify_all();
21
22     // ThreadLoopが完全に終了するまで待機
23     m_thread.join();
24 }
25
26 void JobSystem::AddJob(FunctionType type, Job* job)
27 {
28     m_jobContainerMap[type].Register(job);
```

JobSystemクラスの関数の実装部分です。

初期化処理では、

フラグを立てた後にスレッドの起動を行っています。

左のサンプルでは、ThreadLoop関数を起動しています。

そして終了処理では、フラグを降ろして

ThreadLoop関数が終了するまで待機しています。

#プラスα

m_conditionVar.notify_allの所で、もしThreadLoop関数が待機中の場合はスレッドを強制的に起動しています。

↑後で、ThreadLoop関数の実装の所でも説明します。

前提条件を考慮するジョブシステム6(実装例)

```
29 }
30
31 void JobSystem::AddQueue(Job* job, bool isThreadSafe)
32 {
33     if (isThreadSafe == false)
34     {
35         // 別スレッド動作の設定が無ければ、普通の実行キューに追加
36         m_jobQueue.push(job);
37     }
38     else
39     {
40         // 別スレッド設定があれば、別スレッドの実行キューに追加
41
42         // 実行キューに追加する前に排他制御をかける（これなかったらバグります）
43         std::unique_lock<std::mutex> lock(m_mutex);
44         m_anyThreadJobQueue.push(job);
45
46         // 実行キューにジョブが追加されたことを通知します。
47         m_conditionVar.notify_one();
48     }
49 }
50
51 void JobSystem::Execute(float deleteTime, FunctionType type)
52 {
53     // 順にキューへの登録を試みます。ここでは、
54     // 前提条件がないもののみキューへ追加されます。
55     for (auto job : m_jobContainerMap[type])
56     {
```

#簡単な実装例

そして、AddQueue関数の実装です。

ポイントは

スレッド設定から、追加先の実行キューを選択

している点です。

別スレッド用の実行キューに追加する際、
実行中の可能性があるため、排他処理を入れてます。

#プラスα

更に良くしていくには、排他制御をした際のブロック処理で
処理が止まってしまう問題をどうするかとかですかね。

↑正直めっちゃ小さいので考慮しなくても良いと思っています。

前提条件を考慮するジョブシステム7(実装例)

そして一番メインのThreadLoop関数の実装です。

実行キューにアクセスする前に排他制御して、
ジョブの追加 Or 終了まで待機、その後wait以下処理に移ります。(元々ジョブがあれば処理されない)

追加だった場合、キューから取り出し、
実行前に排他制御を解除しています。←めっちゃ大事。

```
67 }
68 }
69
70 void JobSystem::ThreadLoop()
71 {
72     while (true)
73     {
74         // 実行キューへアクセスする前に排他制御する(これなかったらバグります)
75         std::unique_lock<std::mutex> lock(m_mutex);
76
77         // 実行キューにジョブがなかったら追加まで待機するプログラムです。
78         m_conditionVar.wait(lock, [this] { return !m_anyThreadjobQueue.empty() || !m_isRunning; });
79
80         // ThreadLoop関数の終了を検知されると、Loop終了
81         if (m_anyThreadjobQueue.empty() && !m_isRunning)
82         {
83             return;
84         }
85
86         auto job = m_anyThreadjobQueue.front();
87         m_anyThreadjobQueue.pop();
88
89         // ジョブの実行前に必要で、排他制御を解除します(ジョブの追加を許可します)
90         lock.unlock();
91
92         printf("Any Thread!!\n");
93
94         // 実行
95         job->Execute(0);
96     }
97 }
```

#簡単な実装例

#プラスα

今回はサンプルで適当にジョブ実行時に0を渡していますが、
皆さんがちゃんとした経過時間を取得する必要があります。

ジョブスケジューラーのまとめ

今回は簡単な例でしたが、
ジョブスケジューラーを実装してみました。

結構難しい内容になったかもしれませんが、
実行タイミング、前後関係、別スレッド処理など結構色んな機能を作れました。

いくつか処理を足さないと皆さんの作品にはまだ使えないと思います。
特に、ジョブの前提条件が実行済みの場合の処理とか（実行済みだとイベントが呼ばれない）
自分なりに実装、拡張してしてみてください。

皆さんの作品がより良いものになることを祈っております。