# Code Segment と Data Segment を持つ Gears OS の設計

小久保 翔平　並列信頼研

## 並列環境下におけるプログラミング

・マルチコア CPU の性能を発揮するには、処理をできるだけ並列化しなければならない
・アムダールの法則により、並列化による性能向上は並列化されていない部分の割合に大きく影響される

・先行研究である Cerium は Task ベースで並列処理を記述する。
・Cerium は Task 間の依存関係を設定することで複雑な並列処理を記述可能で、Task をパイプライン処理することでプロセッサの性能を発揮することができる
・GPU を用いた実行もサポートしている

・Task 間の依存関係だけではデータの正しさを保証できない
・汎用ポインタを用いてデータの受け渡しを行うので、型情報が落ちて型システムで Task の組み合わせを検査することができずプログラムの正しさを保証することができない
・Allocator を Thread 間で共有しており、ある Thread がメモリ確保している間は他の Thread はメモリ確保することができず並列度の低下に繋がる
・一般的に参照透過な処理は並列化を行いやすいが、オブジェクト指向ではオブジェクトの状態によって振る舞いが変わるため並列処理との相性が悪い

Cerium を開発して得られた知見を元に Code Segment と Data Segment を持つ Gears OS を設計した

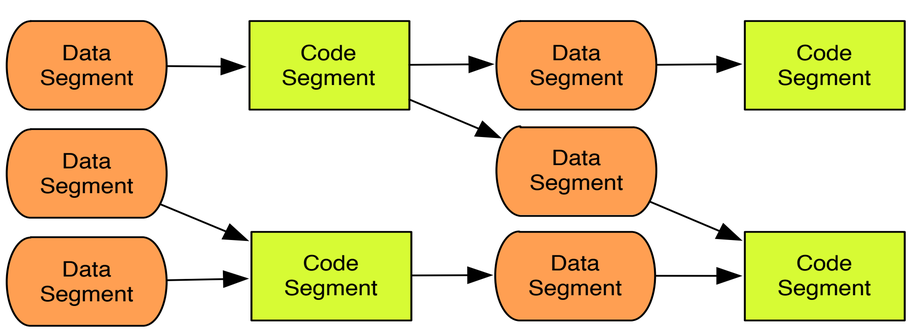
## Code Segment と Data Segment

・Code Segment は 処理の単位、Data Segment は型情報を持つ分割されたデータになる
・Code Segment は名前を指定して goto で遷移する
・Data Segment が揃うことで Code Segment は実行される
・Meta Data Segment である Context に名前と遷移先の Code Segment の組のリストが格納されている

```
// Code Segment Name
enum Code {
    Code1,
    Code2,
    Exit,
};

struct Context {
    int codeNum;
     __code (**code)
    (struct Context*);
    int dataNum;
    union Data **data;
};

// Data Segment definition
union Data {
    // size: 4 byte
    struct Data1 {
        int i;
    } data1;
    // size: 5byte
    struct Data2 {
        int i;
        char c;
    } data2;
};
```

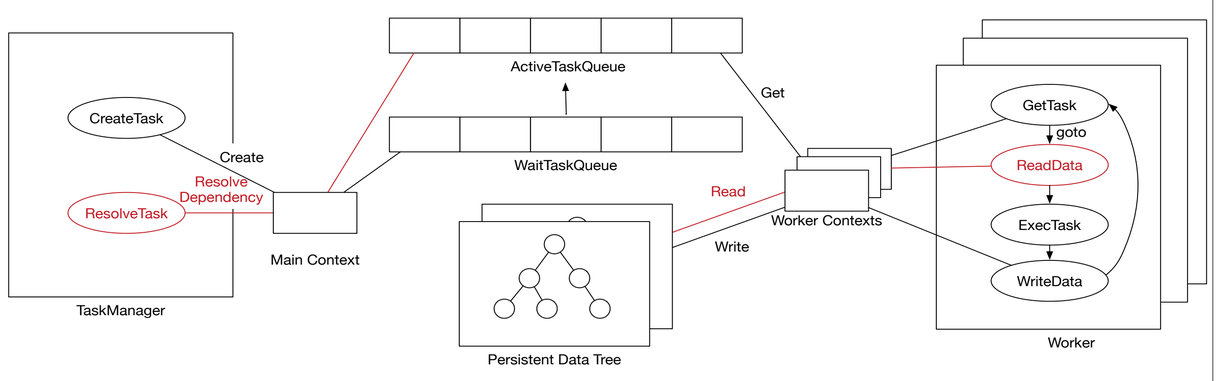
```
// Init Context
extern __code code1_stub
  (struct Context*);
extern __code code2_stub
  (struct Context*);
extern __code exit_code
  (struct Context*);

__code
initContext(struct Context* context)
{
   context->codeNum      = Exit;
   context->code         = calloc();
   context->code[Code1] = code1_stub;
   context->code[Code2] = code2_stub;
   context->code[Exit]  = exit_code;

   context->dataNum      = 0;
   context->data         = calloc();
}
```

同じ名前を使って遷移先を動的に変更することで Meta Computaion を実現可能にする




## Gears OS の構成

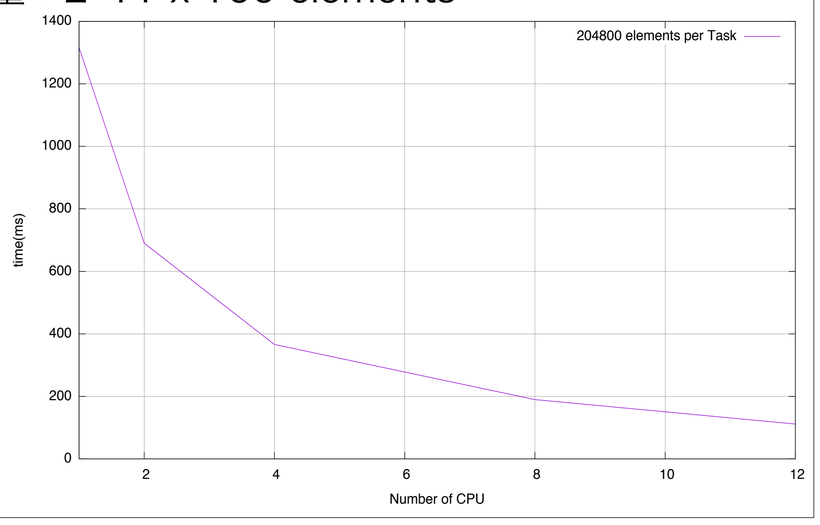
・Context
接続可能な Code/Data Gear のリスト、TaskQueue へのポインタ、Persistent Data Tree へのポインタ、独立したメモリ空間を持つ
・TaskQueue
ActiveTaskQueue と WaitTaskQueue を持つ
依存関係がある Task は WaitTaskQueue に挿入され、依存関係が解決されると実行可能な状態となり ActiveTaskQueue に移される
Compare and Swap 命令を用いることで平行に操作してもデータの一貫性を保つ
・Persistent Data Tree
非破壊木構造で構築されるので平行して読み書き可能
挿入・削除・検索における計算量を保証するため Red-Black Tree アルゴリズムを用いて平衡性を保つ
Persistent Data Tree への書き込みのみで相互作用を発生させる
・TaskManager
Persistent Data Tree を監視して Task の依存関係を解決する
Task の依存関係は Input/Output Data Gear から自動的に決定する
・Worker
TaskQueue から Task を取得し実行する
Task に必要な Data Gear は Persistent Data Tree から取得する




Gears OS のプロトタイプを実装し、並列実行の測定を行った

・要素数：2^17 x 1000 elements
・分割数：640 tasks
・1 Task 当たりの処理量：2^11 x 100 elements

・1 CPU と 12 CPU で約11.8倍の速度向上
・台数効果による十分な性能向上を確認した




## Cerium との比較

| | |
|---|---|
| Task 間の依存関係 | ・Input/Output Data Gear から依存関係を決定する |
| データの型情報 | ・Data Gear として定義されており型情報を持つ |
| メモリ確保 | ・Context は独立したメモリ空間を持ちお互いに干渉しない |
| 並列処理との相性 | ・Code/Data Gear を用いることで並列処理に向いた形で記述できる |

## 今後の課題

・複雑な並列処理を実行できるようにするために依存関係を解決する TaskManager を実装する必要がある
・GPU などの各プロセッサのアーキテクチャにマッピングした実行機構の実装