動的データ競合検出アルゴリズムの紹介

@chakku_000

データ競合(data race)とは?

```c
int counter=0;
void f(){// thread1
  for(int i=0;i<10000;i++)
    counter++;
}
void g(){// thread2
  for(int i=0;i<10000;i++)
    counter++;
}
int main(){
  pthread_t th1,th2;
  pthread_create(&th1,NULL,(void*)f,NULL);
  pthread_create(&th2,NULL,(void*)g,NULL);
  pthread_join(th1,NULL);
  pthread_join(th2,NULL);
  printf("counter = %d\n",counter);
  return 0;
}
```

データ競合とはマルチスレッドプログラミングで発生し得る並行バグの1つ。次の3条件を満たすとデータが不正な値になり得る。

- 特定のメモリ領域に
- 複数のスレッドから同時にアクセスがあり
- 少なくとも1つのアクセスが書き込み操作

左のコードでは、

- 特定のメモリ領域 = 変数counter
- 複数のスレッド = th1, th2
- 少なくとも1つの書き込み操作 = counter++

と条件を満たすのでデータ(counterの値)が不正になっている。

出力：counter = 117962

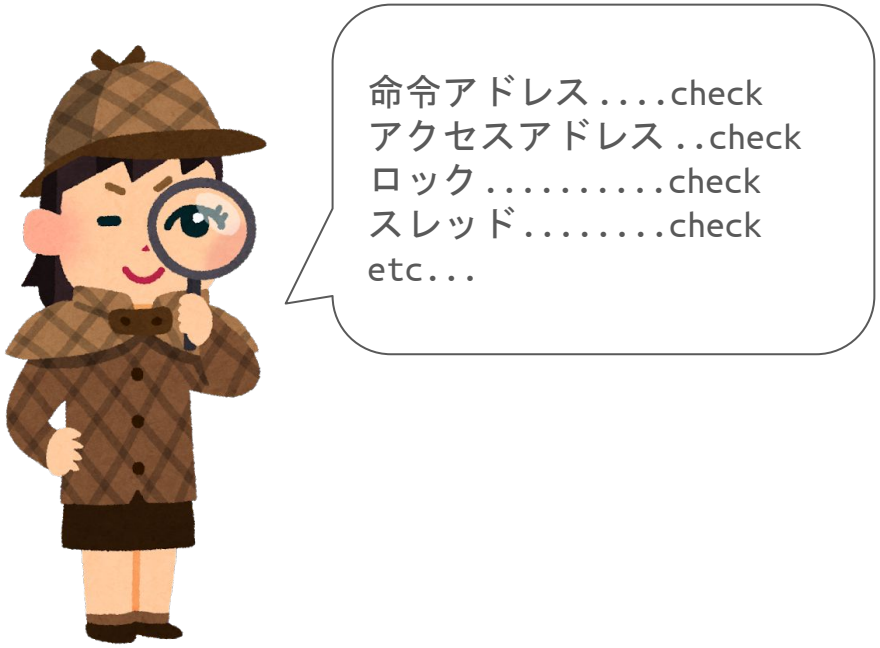
f,gで10000回ずつインクリメントするので
counterの値は200000になるはずだがデータ競合により値が定まらない
(もう一度実行しても117962になるとは限らない)。

動的データ競合検出とは?

検査対象のプログラムを実際に実行させ、トレース情報を収集し、
収集した情報をもとにデータ競合を検出する。

静的な検出に対する利点として精度が高い。

しかし欠点として実際に実行されたパスしか検査されないことが挙げられる。



検査対象のプログラムを実行

動的データ競合検査アルゴリズム

動的データ競合検査アルゴリズムは主に2つ

- Lockset Algorithm
  - ロックに注目してデータ競合を検査
- Happens-Before based Algorithm
  - アクセスの順序関係に注目してデータ競合を検査

Lockset Algorithm

「アドレスxを保護しているロックが存在するか？」をチェックする。
アドレスxを保護するロックが存在しない場合にデータ競合が存在すると判断。

データ競合を起こす疑似コード

```
int x = 0;
Mutex mtx1,mtx2;

// thread1
func f(){
      lock(mtx1);
      write(x); // アクセスα
      unlock(mtx1);
}
// thread2
func g(){
      lock(mtx2);
      write(x); // アクセスβ
      unlock(mtx2);
}
// thread0
int main(){
      fork(f,g);
      lock(mtx1,mtx2);
      write(x); // アクセスγ
      unlock(mtx1,mtx2);
      join(f,g);
}
```

アクセスαとアクセスγはmtx1により排他制御されるのでデータ競合を起こさない

アクセスβとアクセスγはmtx2により排他制御されるのでデータ競合を起こさない

アクセスαとアクセスβは排他制御されないのでデータ競合を起こす可能性がある

Lockset Algorithm

データ競合を起こす疑似コード

```
int x = 0;
Mutex mtx1,mtx2;

// thread1
func f(){
      lock(mtx1);
      write(x); // アクセスα
      unlock(mtx1);
}
// thread2
func g(){
      lock(mtx2);
      write(x); // アクセスβ
      unlock(mtx2);
}
// thread0
int main(){
      fork(f,g);
      lock(mtx1,mtx2);
      write(x); // アクセスγ
      unlock(mtx1,mtx2);
      join(f,g);
}
```

参考
Eraser: A dynamic data race detector for multi-threaded programs. (ACM Transactions on Computer Systems)

thread0	thread1	thread2	変数xを保護するロックの集合
			mtx1,mtx2
write(x) with mtx1,mtx2			{mtx1,mtx2}∩ {mtx1,mtx2} = mtx1,mtx2
	write(x) with mtx1		mtx1 ∩ {mtx1,mtx2} = mtx1
		write(x) with mtx2	mtx2 ∩ {mtx1} = empty

変数xを保護するロックが存在しないので
データ競合が存在すると判定

Happens-Before based Algorithm

メモリへのアクセスの順序を計算し、
異なる2つのアクセスに順序関係が無い場合にデータ競合が存在すると判断。
この順序関係を**Happens-Before関係**という。

Happens-Before関係の計算には***vector clock***を使用する。

- vector clockとは...?
  - 各スレッドの論理時刻を保存したベクトル
    - 例えばスレッド0の論理時刻が1,スレッド1の論理時刻が2,スレッド2の論理時刻が2の場合は〈1,2,2〉のように表現する
  - vector clockは比較可能かつ推移律も成り立つ
    - 2つのvector clock間で大小関係が成り立たないことと、そのvector clockに対応するイベント間に順序関係がないことは等価

Happens-Before based Algorithm

データ競合を起こす疑似コード

```
int x = 0;
Mutex mtx1,mtx2;

// thread1
func f(){
      lock(mtx1);
      write(x); // アクセスα
      unlock(mtx1);
}
// thread2
func g(){
      lock(mtx2);
      write(x); // アクセスβ
      unlock(mtx2);
}

// thread0
int main(){
      fork(f,g);
      join(f,g);
}
```

thread1	thread2	mtx1	mtx2
〈1,1,0〉	〈1,0,1〉	〈0,0,0〉	〈0,0,0〉
lock(mtx1)			
〈1,1,0〉			
	lock(mtx2)		
	〈1,0,1〉		
write(x)			
unlock(mtx1)			
〈1,2,0〉	write(x)	〈1,1,0〉	
	unlock(mtx2)		
	〈1,0,2〉		〈1,0,1〉

αのvector clockは〈1,1,0〉、βのvector clockは〈1,0,1〉。
α→βの順序を確かめるためにはαに対応するスレッド(thread1)のエントリ(赤字部分)。この場合α(1)がβ(0)よりも大きいので順序関係が存在しないのでデータ競合と判定する。

参考
Hybrid Dynamic Data Race Detection(PPoPP'03)