Twin-and-diff 设计

1.结构体

inc/spmc_defs.h

```
typedef struct spmc_space {
        ……
        write_record_t *twin_dif;
        ……
}spmc_space_t;
```

```
typedef struct write_record {
        address_map_t *write_set;
        void *snapdata;
        diff_arr_t *diff_set;
} write_record_t;
```

2.接口

twindif_init(spmc_space_t *space);	完成 space->record 中的数据结构的初始化
twindif_protect(void *args)	保护 args 中给出的地址范围，如果对该范围进行修改，将产生 SIGSEGV，进程捕获后调用 twindif_snapshot()处理
twindif_unprotect(void *args)	已经 snapshot 的 page，再次访问将不再产生 SIGSEGV
twindif_snapshot(off_t addr);	首先判断 addr 是否属于保护的范围: 如果是，记录下修改的 page 并拷贝 snapshot 到 twin_diff->snapshot; 否则什么都不做;
twindif_compare(write_record_t * twin_dif);	子进程完成 fun 的调用后，将自己的保护区与 snapshot 进行比对，如果不同，记录到 twin_dif->diff_set 中
twindif_merge(spmc_space_t *space);	将父进程的保护区域中的内容与子进程的 diff_set 内容做比对，更新父进程的保护区域
twindif_clear(spmc_space_t *space);	清除 space->twin_dif 的 mmap 空间

3.dlinux 中代码的框架

```
spaceid_t
space_alloc(int uid)
{
……
if ( spmc_g->nused < spmc_g->nspaces ) {
        twindif_init(space);
}
}
```

```
void
space_free(spaceid_t idx)
{
……
```

```
static void
handle_signal(int signum, siginfo_t * siginfo, void * ctx)
{
        off_t addr = (off_t)siginfo->si_addr;
        if (signum == SIGSEGV) {
#ifdef TWIN_AND_DIF
                twindif_snapshot(addr);
#endif
                BEGIN_TIMING2(pagefault, spmctime);
                spmcpmap_pagefault(addr);
                END_TIMING2(pagefault, spmctime);
        }
```

```
int
space_start(spaceid_t idx, void *(*fn)(void*), void *arg)
{
    if (pid == 0) { // this is child
        close(space->pipes.up[PIPE_WRITE]);
        spmc_s.idx = idx;
        spmc_s._space = space;
        spmc_s._space->state = SPACE_RUN;
        spmc_s.heap = NULL;
        memset(&spmc_s.heap_seg, 0, sizeof(segment_t));
        spmc_s.pipe_r = space->pipes.up[PIPE_READ];
#ifdef TWIN_AND_DIF
        twindif_protect(scope);
#endif
        if (fn) {
            init_tstat(space);
            BEGIN_TIMING_RUNTIME;
            fn(arg);
            END_TIMING_RUNTIME;
#ifdef TWIN_AND_DIF
            twindif_compare(space->twin_diff);
#endif
        }
        space_ret();
        return 0;
}
```

```
void
space_sync(spaceid_t idx, bool restart)
{
    spmc_space_t *space = &spmc_g->spaces[idx];
    assert(MYID == space->parent);
    if ( write(space->pipes.up[PIPE_WRITE], &restart,
            sizeof(restart)) != sizeof(restart))
        panic("space_sync: S%d write", idx);

    if ( !restart ) {
        // the child space is finished
        close(space->pipes.up[PIPE_WRITE]);
        SPMC_WAITPID(space->pid, NULL, 0);

//保证此时子进程已经完成了 twindif_compare()
#ifdef TWIN_AND_DIF
        twindif_merge(space);
#endif
        space_free(idx);
    }

}
```

4.详细方案

1) 保护范围的设置

一方面，不必对整个 master heap 中的内容都进行保护，设置保护范围的区域，这样可以提高 twin-and-diff 的效率，避免不必要的 twin-and-diff。

另一方面，保护的范围不应在应用程序中设置，而是又 DMR 库完成

以 kmeans 和 matrix_multiply 两个 app 为例，父子线程需要共享的 heap 对象有一定的**规律**：它们都是调用 map_reduce()之前的 heap 中分配的对象，主要用于存放结果

目前的做法：

只保护 master 在调用 map_reduce()之前的 heap 对象（由 malloc 分配的变量）

小实验证明：malloc()后不进行初始化，mprotect 该区域，可以对指定的 malloc 区域进行保护

DMR 库设置范围的流程：

map_reduce()

=》env_init()　(map_reduce.c)

=》thread_share()　(spmc_thread.c)

=》space_share()　(spmc_space.c)

```
typedef struct scope {
        void *begin;
        void *end;
} scope_t;
scope_t *scope = NULL;
```

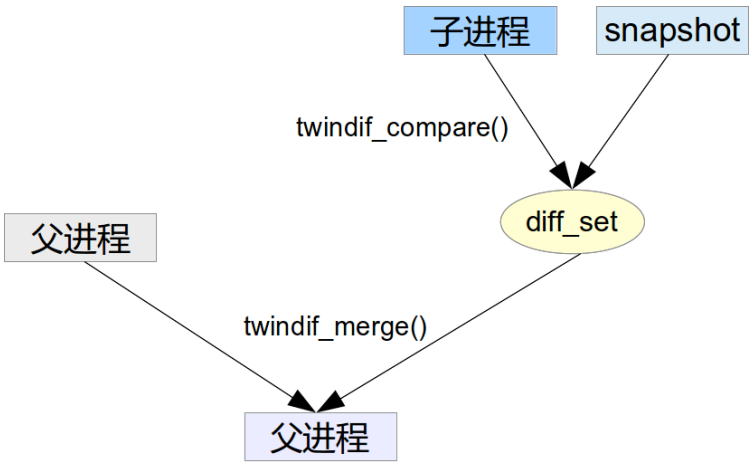
space_share()完成全局变量 scope 的设置

scope 是保护的范围

```
void space_share()
{
        if(spmc_s == NULL) return;
        scope = (scope_t *)malloc(sizeof(scope_t));
        scope->begin = spmc_s.heap_seg.start;
        scope->end = scope;//由 heap 的分配得到
}
```

2)　父进程、子进程和 snapshot 的比对

简单的实现方式如下图



考虑多种情况的发生，

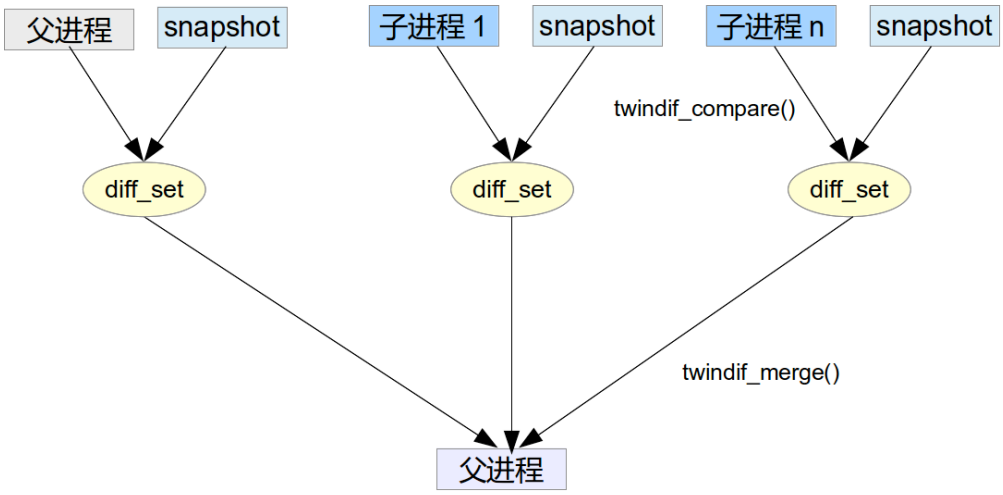
① 父进程可能修改保护区的内容

② 多个子进程修改保护区内容

③ 父进程和多个子进程修改同一个位置，发生 conflict!!!

问题 1：如何辨别父子进程对同一位置修改是 conflict 还是更新？

问题 2：如果发生 conflict，该如何处理？



存在一个问题，space_free()后，子进程的 space->twin_dif 结构已经被释放，无法做集中的对比

1.mapreduce 调用与 twin-and-diff 机制

(1)Applications 对 twin-and-diff 机制需求的分类

	map_reduce()调用前没有 malloc		map_reduce()调用前有 malloc	
	1 次 MR	n 次 MR	1 次 MR	N 次 MR
Applications	histogram linear_regression	word_count	string_match matrix_multiply	pca kmeans
Twin-and-diff 解决方案 (twindif_setscope())	twindif_enable = -1; 关闭 twin-and-diff 机制	第一次 MR： twindif_enable = -1; 之后的迭代： 判断 twindif_enable	twindif_enable = 1 打开开 twin-and-diff 机制	第一次 MR: twindif_enable = 1 设置 scope 之后的迭代： scope 无需重新设置

(2)代码实现

```
void twindif_setscope()
{
/* if application does't malloc before The function*/
    if(spmc_s.heap == NULL) {
        twindif_enable = -1;
        return;
    }
    /* if scope not NULL, return */
    if(scope || twindif_enable == -1) return;

    scope = (scope_t *)malloc(sizeof(scope_t));
    scope->start = ROUNDDOWN((void *)spmc_s.heap_seg.start, PAGE_SIZE);
    scope->end = ROUNDUP((void *)scope, PAGE_SIZE);

    /* fill rest of a page */
    void *blank = malloc(scope->end - (void *)scope);
    memset(blank, 0, scope->end - (void *)scope);
}
```

```
scope_t *scope = NULL;
/**
 * default = 1: enable
 * -1: disable
 */
int twindif_enable = 1;
```

2.页对齐和填充

mprotect 要求页对齐保护，因此将 end,start 进行对齐
但存在**页内碎片**（右图 Unused 部分），

为了避免非保护对象落入该区域，
采取的策略：将该区域填充为 0