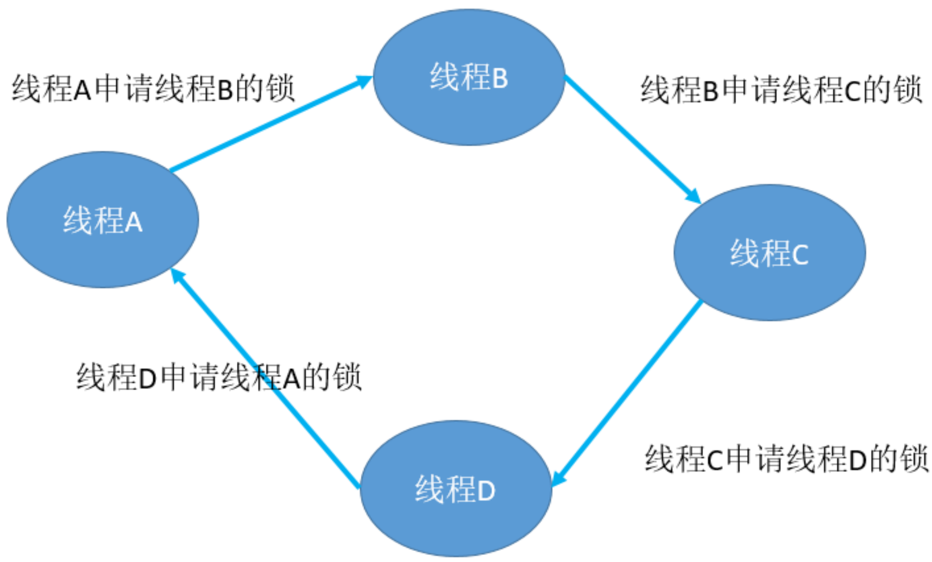
死锁检测组件实现

一．死锁存在的条件

死锁，是指多个线程或者进程在运行过程中因争夺资源而造成的一种僵局，当进程或者线程处于这种僵持状态，若无外力作用，它们将无法再向前推进。如下图所示，线程 A 想获取线程 B 的锁，线程 B 想获取线程 C 的锁，线程 C 想获取线程 D 的锁，线程 D 想获取线程 A 的。锁，从而构建了一个资源获取环。



死锁的存在是因为有资源获取环的存在，所以只要能检测出资源获取环，就等同于检测出死锁的存在。

二．死锁检测实现的原理

资源获取环可以采用图来存储，使用有向图来存储。线程 A 获取线程 B 已占用的锁，则为线程 A 指向线程 B。如何为线程 B 已占用的锁？运行过程线程 B 获取成功的锁。
检测的原理采用另一个线程定时对图进程检测是否有环的存在。

数据结构定义

```
enum Type {PROCESS, RESOURCE};

struct source_type {

    uint64 id;
```

```c
    enum Type type;

    uint64 lock_id;

};

struct vertex {

    struct source_type s;
    struct vertex *next;

};

struct task_graph {

    struct vertex list[MAX];
    int num;

    struct source_type locklist[MAX];
    int lockidx;

    pthread_mutex_t mutex;
};
```

图算法，检测成环

```c
struct task_graph *tg = NULL;
int path[MAX+1];
int visited[MAX];
int k = 0;
int deadlock = 0;

struct vertex *create_vertex(struct source_type type) {

    struct vertex *tex = (struct vertex *)malloc(sizeof(struct
vertex ));

    tex->s = type;
    tex->next = NULL;

    return tex;

}
```

```c
int search_vertex(struct source_type type) {

    int i = 0;

    for (i = 0;i < tg->num;i ++) {

        if (tg->list[i].s.type == type.type && tg->list[i].s.id ==
type.id) {
            return i;
        }

    }

    return -1;
}

void add_vertex(struct source_type type) {

    if (search_vertex(type) == -1) {

        tg->list[tg->num].s = type;
        tg->list[tg->num].next = NULL;
        tg->num ++;

    }

}


int add_edge(struct source_type from, struct source_type to) {

    add_vertex(from);
    add_vertex(to);

    struct vertex *v = &(tg->list[search_vertex(from)]);

    while (v->next != NULL) {
        v = v->next;
    }

    v->next = create_vertex(to);

}
```

```c
int verify_edge(struct source_type i, struct source_type j) {

    if (tg->num == 0) return 0;

    int idx = search_vertex(i);
    if (idx == -1) {
        return 0;
    }

    struct vertex *v = &(tg->list[idx]);

    while (v != NULL) {

        if (v->s.id == j.id) return 1;

        v = v->next;

    }

    return 0;

}


int remove_edge(struct source_type from, struct source_type to) {

    int idxi = search_vertex(from);
    int idxj = search_vertex(to);

    if (idxi != -1 && idxj != -1) {

        struct vertex *v = &tg->list[idxi];
        struct vertex *remove;

        while (v->next != NULL) {

            if (v->next->s.id == to.id) {

                remove = v->next;
                v->next = v->next->next;

                free(remove);
                break;

            }
```

```
            v = v->next;
        }

    }

}


void print_deadlock(void) {

    int i = 0;

    printf("deadlock : ");
    for (i = 0;i < k-1;i ++) {

        printf("%ld --> ", tg->list[path[i]].s.id);

    }

    printf("%ld\n", tg->list[path[i]].s.id);

}

int DFS(int idx) {

    struct vertex *ver = &tg->list[idx];
    if (visited[idx] == 1) {

        path[k++] = idx;
        print_deadlock();
        deadlock = 1;

        return 0;
    }

    visited[idx] = 1;
    path[k++] = idx;

    while (ver->next != NULL) {

        DFS(search_vertex(ver->next->s));
        k --;

        ver = ver->next;
```

```
    }
    return 1;

}


int search_for_cycle(int idx) {

    struct vertex *ver = &tg->list[idx];
    visited[idx] = 1;
    k = 0;
    path[k++] = idx;

    while (ver->next != NULL) {

        int i = 0;
        for (i = 0;i < tg->num;i ++) {
            if (i == idx) continue;

            visited[i] = 0;
        }

        for (i = 1;i <= MAX;i ++) {
            path[i] = -1;
        }
        k = 1;

        DFS(search_vertex(ver->next->s));
        ver = ver->next;
    }

}
```

图算法的测试入口函数

```
int main() {


    tg = (struct task_graph*)malloc(sizeof(struct task_graph));
    tg->num = 0;

    struct source_type v1;
    v1.id = 1;
    v1.type = PROCESS;
    add_vertex(v1);
```

```
    struct source_type v2;
    v2.id = 2;
    v2.type = PROCESS;
    add_vertex(v2);

    struct source_type v3;
    v3.id = 3;
    v3.type = PROCESS;
    add_vertex(v3);

    struct source_type v4;
    v4.id = 4;
    v4.type = PROCESS;
    add_vertex(v4);


    struct source_type v5;
    v5.id = 5;
    v5.type = PROCESS;
    add_vertex(v5);


    add_edge(v1, v2);
    add_edge(v2, v3);
    add_edge(v3, v4);
    add_edge(v4, v5);
    add_edge(v3, v1);

    search_for_cycle(search_vertex(v1));

}
```

通过另外开启线程，检测资源环。

```
void check_dead_lock(void) {

    int i = 0;

    deadlock = 0;
    for (i = 0;i < tg->num;i ++) {
        if (deadlock == 1) break;
        search_for_cycle(i);
    }
```

```c
    if (deadlock == 0) {
        printf("no deadlock\n");
    }

}


static void *thread_routine(void *args) {

    while (1) {

        sleep(5);
        check_dead_lock();

    }

}


void start_check(void) {

    tg = (struct task_graph*)malloc(sizeof(struct task_graph));
    tg->num = 0;
    tg->lockidx = 0;

    pthread_t tid;

    pthread_create(&tid, NULL, thread_routine, NULL);

}
```

Hook pthread_mutex_lock/pthread_mutex_unlock 函数接口

```c
int pthread_mutex_lock(pthread_mutex_t *mutex) {

    pthread_t selfid = pthread_self(); //
    //printf("pthread_mutex_lock : %ld\n", selfid);

    lock_before(selfid, (uint64)mutex);
    pthread_mutex_lock_f(mutex);
    lock_after(selfid, (uint64)mutex);

}
```

```c
int pthread_mutex_unlock(pthread_mutex_t *mutex) {


    pthread_t selfid = pthread_self();

    pthread_mutex_unlock_f(mutex);
    unlock_after(selfid, (uint64)mutex);


}

static int init_hook() {

    pthread_mutex_lock_f = dlsym(RTLD_NEXT, "pthread_mutex_lock");

    pthread_mutex_unlock_f = dlsym(RTLD_NEXT, "pthread_mutex_unlock");

}
```