mobibench源码阅读文档

In Mobibench, one can generate two types of workloads: file operations and database operations. Mobibench specifies spatial locality (random vs. sequential), I/O mode ( read vs. write), file size, I/O unit size, and synchronization mode. There exist five synchronization modes: buffered, synchronous, direct, mmap, and write()+fsync().

Mobibench generates three performance values: throughput, CPU utilization, and the number ofcontext switches. In file I/O test, units of throughput are “KB/s” for sequential operations and “IOPS”for random operations. Unit of throughput in SQLite operation is “transaction/ sec”. Utilization of CPU distinguishes ACTIVE, IDLE, and IO-WAIT to understand how the test utilizes the CPU. Mobibench also counts the number of context switches to measure the context switch overheads.

mobibench和mobigen在IO栈中的位置如图1所示

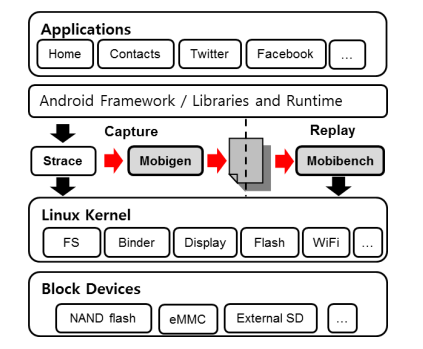


图1 mobibench和mobigen在IO栈中的位置

## Usage (shell version)

# mobibench [-p pathname] [-f file\_size\_Kb] [-r record\_size\_Kb] [-a access\_mode] [-h]

[-y sync\_mode] [-t thread\_num] [-d db\_mode] [-n db\_transcations]

[-j SQLite\_journalmode] [-s SQLite\_syncmode] [-g replay\_script] [-q]

* -p set path name (default=./mobibench)
* -f set file size in KBytes (default=1024)
* -r set record size in KBytes (default=4)
* -a set access mode (0=Write, 1=Random Write, 2=Read, 3=Random Read) (default=0)
* -y set sync mode (0=Normal, 1=O\_SYNC, 2=fsync, 3=O\_DIRECT, 4=Sync+direct, 5=mmap, 6=mmap+MS\_ASYNC, 7=mmap+MS\_SYNC 8=fdatasync) (default=0)
* -t set number of thread for test (default=1)
* -d enable DB test mode (0=insert, 1=update, 2=delete)
* -n set number of DB transaction (default=10)
* -j set SQLite journal mode (0=DELETE, 1=TRUNCATE, 2=PERSIST, 3=WAL, 4=MEMORY, 5=OFF) (default=1)
* -s set SQLite synchronous mode (0=OFF, 1=NORMAL, 2=FULL) (default=2)
* -g set replay script (output of MobiGen)
* -q do not display progress(%) message

# mobibench关键数据结构

## file\_test\_mode\_t;

文件测试模式，写、随机写、读、随机读

typedef enum

{

MODE\_WRITE,

MODE\_RND\_WRITE,

MODE\_READ,

MODE\_RND\_READ,

} file\_test\_mode\_t;

## file\_sync\_mode\_t

文件同步模式

typedef enum

{

NORMAL,

OSYNC,

FSYNC,

ODIRECT,

SYDI,

MMAP,

MMAP\_AS,

MMAP\_S,

FDATASYNC,

} file\_sync\_mode\_t;

## thread\_status\_t

线程的状态

typedef enum

{

NONE,

READY,

EXEC,

END,

ERROR,

} thread\_status\_t;

## struct script\_entry

存放输入文件的信息，命令和命令的参数，命令实际执行过程中启动的线程数。

struct script\_entry {

int thread\_num;

long long time;

char\* cmd;

char\* args[3];

int arg\_num;

};

## struct script\_thread\_time

线程时间

struct script\_thread\_time {

int thread\_num;

int count;

int started;

int ended;

long long start;

long long end;

long long io\_time;

int io\_count;

int write\_size;

int read\_size;

};

## struct script\_thread\_info

线程信息

struct script\_thread\_info {

int thread\_num;

int line\_count;

int open\_count;

};

## struct script\_fd\_conv

fd\_conv暂时不知道具体意义，后续添加

struct script\_fd\_conv {

int\* fd\_org;

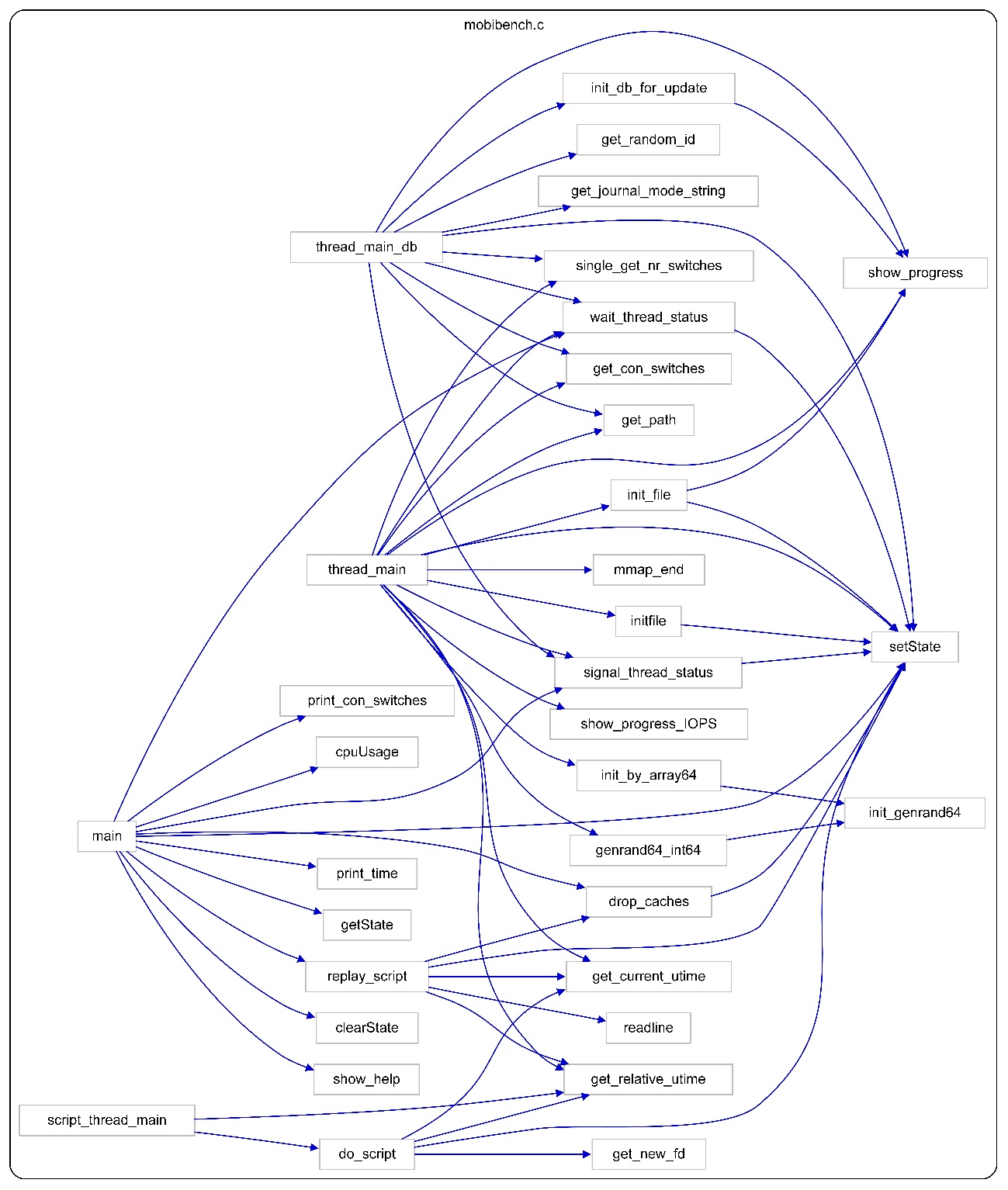
int\* fd\_new;

int index;

};

# mobibench关键函数流程

在mobibench分两种测试模式，若提供测试脚本则按照则是脚本测试，否则按照配置的参数进行测试。mobibench的整体调用图如下：

amin

# 关键函数代码解析

mobibench在提供输入脚本时会对输入脚本的命令进行回放，若未提供输入脚本，则对按照配置参数进行回放，在提供脚本时参与回放的关键的函数有replay\_script, script\_thread\_main, do\_script；指定配置参数参与回放的关键函数有thread\_main，thread\_main\_db

## replay\_script函数

replay\_script首先会对读取输入脚本信息，然后将输入脚本中命令和参数存放在gScriptEntry结构体中，分配线程内存，清空缓存，创建线程(pthread\_create)，并且将script\_thread\_main作为线程入口函数，换句话说，创建线程的时候会调用script\_thread\_main。

创建线程：

while(1) {

time\_current = get\_relative\_utime(time\_start);

for(i = 0; i < script\_thread\_num; i++)

{

if(gScriptThreadTime[i].started == 0)

{

/\* If the start time of a thread reached, create the thread. \*/

if(gScriptThreadTime[i].start <= (time\_current+1000000)) /\* 1sec margin \*/

{

thread\_info[i].thread\_num = gScriptThreadTime[i].thread\_num;

thread\_info[i].line\_count = line\_count;

thread\_info[i].open\_count = open\_count;

ret = pthread\_create((pthread\_t \*)&thread\_id[i], NULL, (void\*)script\_thread\_main, &thread\_info[i]);//线程的入口函数

pthread\_detach(thread\_id[i]);

gScriptThreadTime[i].started = 1;

thread\_start\_cnt++;

}

}

}

if(thread\_start\_cnt >= script\_thread\_num) {

break;

}

}

## pthread\_create函数

pthread\_create函数是Linux与线程创建有关的线程函数

函数声明：

int pthread\_create (pthread\_t \*tidp,const pthread\_attr\_t \*attr,(void\*)(\*start\_rtn)(void\*), void \*arg);

函数参数：

第一个参数为指向线程标识符的指针。

第二个参数用来设置线程属性。

第三个参数是线程运行函数的起始地址。

最后一个参数是运行函数的参数。

## script\_thread\_main

script\_thread\_main为启动线程的入口函数，统计线程的IO时间、写请求和读请求大小，当脚本命令的下发时间晚于mobibench创建线程时间时，线程会选择休眠。通过do\_script函数执行线程的实际动作。

函数声明：

int script\_thread\_main(void\* arg)

参数：

提供script\_thread\_info结构变量

gScriptThreadTime[thread\_num].io\_time = 0;

gScriptThreadTime[thread\_num].write\_size = 0;

gScriptThreadTime[thread\_num].read\_size = 0;

//printf("thread[%d] started at %lld, org %lld\n", thread\_num, get\_relative\_utime(time\_start), gScriptThreadTime[thread\_num].start);

for(i = 0; i < thread\_info->line\_count; i++)

{

if(gScriptEntry[i].thread\_num == thread\_num)

{

long long time\_diff = gScriptEntry[i].time - get\_relative\_utime(time\_start);

if(time\_diff > 1)

{

//printf("sleep %lld\n", time\_diff);

usleep(time\_diff-1);

}

//printf("%lld\n", get\_relative\_utime(time\_start) - gScriptEntry[i].time);

io\_time = do\_script(&gScriptEntry[i], &gScriptThreadTime[thread\_num]);

usleep(0);

if(io\_time >= 0)

{

gScriptThreadTime[thread\_num].io\_time += io\_time;

gScriptThreadTime[thread\_num].io\_count++;

//printf("thread[%d] %s\n", thread\_num, gScriptEntry[i].cmd);

}

}

}

## do\_script

函数声明：

int do\_script(struct script\_entry\* se, struct script\_thread\_time\* st)

下图是do\_script的结构图，在do\_script函数中处理输入脚本的命令有：open, close, write, pwrite, read, pread, fsync, fdatasync, access, stat, lstat, fstat, unlink。这些命令依次调用系统函数下发实际操作。

int fd\_new = get\_new\_fd(atoi(se->args[0])); //获取新的 file descriptor

ret = write(fd\_new, script\_write\_buf, atoi(se->args[1]));

int fd\_new = get\_new\_fd(atoi(se->args[0])); //获取新的 file descriptor

ret = pwrite(fd\_new, script\_write\_buf, atoi(se->args[2]), atoi(se->args[1]));

int fd\_new = get\_new\_fd(atoi(se->args[0])); //获取新的 file descriptor

ret = read(fd\_new, script\_read\_buf, atoi(se->args[1]));

int fd\_new = get\_new\_fd(atoi(se->args[0])); //获取新的 file descriptor

ret = pread(fd\_new, script\_read\_buf, atoi(se->args[2]), atoi(se->args[1]));

int fd\_new = get\_new\_fd(atoi(se->args[0])); //获取新的 file descriptor

ret = fsync(fd\_new);

int fd\_new = get\_new\_fd(atoi(se->args[0])); //获取新的 file descriptor

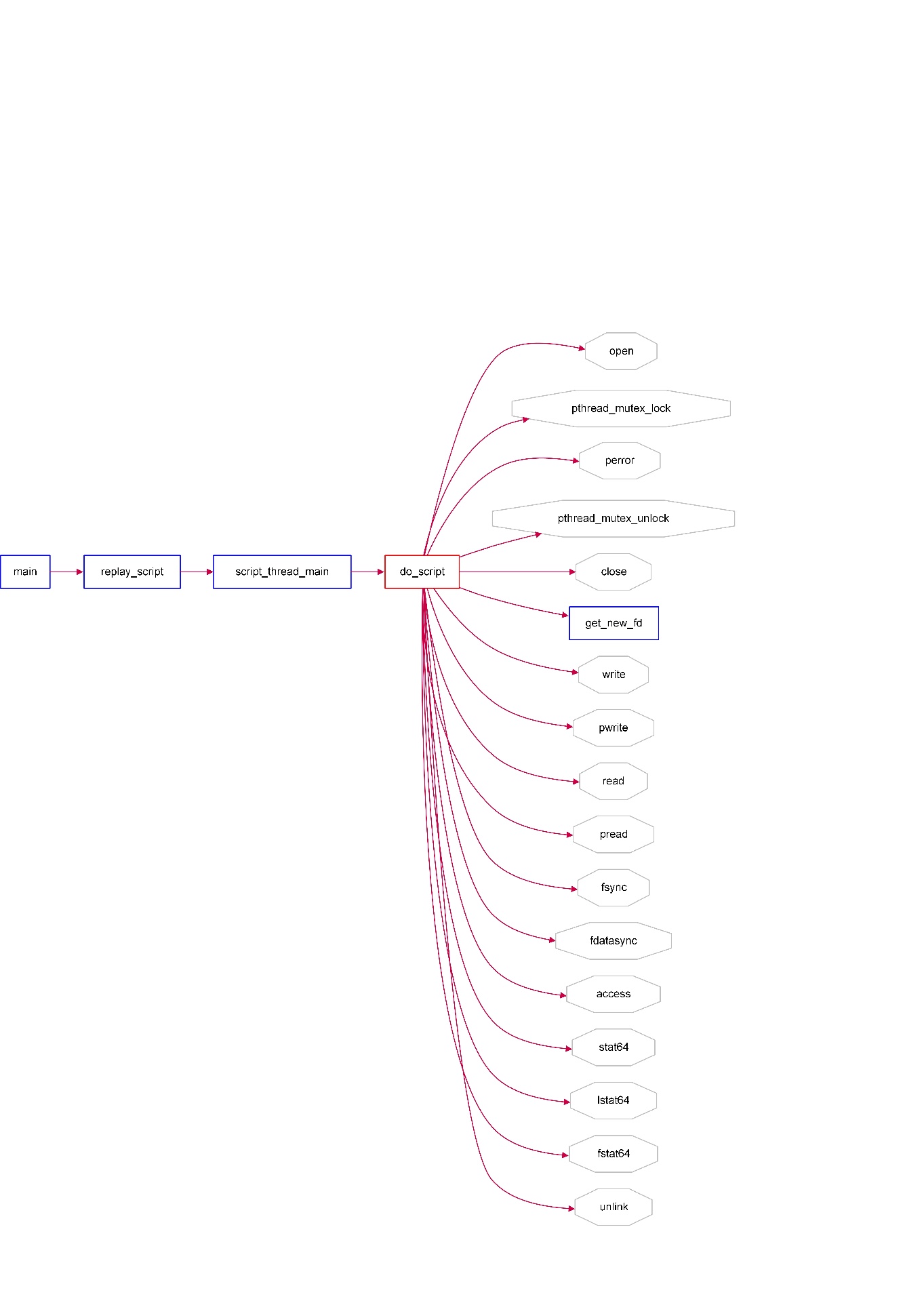
ret = fstat64(fd\_new, &stat\_buf);

ret = access(replay\_pathname, 0777);

ret = stat64(replay\_pathname, &stat\_buf);

ret = lstat64(replay\_pathname, &stat\_buf);

ret = unlink(replay\_pathname);



## pthread\_mutex\_lock和pthread\_mutex\_unlock

函数声明：

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

当pthread\_mutex\_lock()返回时，该互斥锁已被锁定。线程调用该函数让互斥锁上锁，如果该互斥锁已被另一个线程锁定和拥有，则调用该线程将阻塞，直到该互斥锁变为可用为止。

函数声明：

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

pthread\_mutex\_unlock是可以解除锁定 mutex 所指向的互斥锁的函数。

mobibench在按照指定参数的下回放时，会判断是否使能SQlite数据库参与回放，这里存在两个与线程有关的回放函数thread\_main，thread\_main\_db，这两个函数均作为创建线程函数的入口函数。

ret = pthread\_create((pthread\_t \*)&thread\_id[i], NULL, (void\*)thread\_main, &thread\_info[i]);

ret = pthread\_create((pthread\_t \*)&thread\_id[i], NULL, (void\*)thread\_main\_db, &thread\_info[i]);