SQLite 数据库在 μC/OS-III操作系统上的移植①

杨有波[1], 富 坤[1], 吴 清[1], 刘 旭[2]

[1](河北工业大学 计算机科学与软件学院, 天津 300401)
[2](石家庄铁道大学 外语系, 石家庄 050043)

摘 要: 为提高基于嵌入式操作系统μC/OS-III平台的应用软件对数据的处理能力, 本文通过对 SQLite 数据库的操作系统接口的分析, 重点探讨了互斥信号量、内存分配、虚拟文件系统三个子系统的移植方法, 给出了 SQLite 数据库在μC/OS-III操作系统上的移植技术, 并结合实例应用验证了该数据库移植的可行性.

关键词: 数据库; 操作系统接口; 互斥信号量; 内存分配; 虚拟文件系统

Porting SQLite Database to μC/OS-III Operating System

YANG You-Bo[1], FU Kun[1], WU Qing[1], LIU Xu[2]

[1](School of Computer Science and Engineering, Hebei University of Technology, Tianjin 300401, China)
[2](Department of Foreign Language, Shijiazhuang Tiedao University, Shijiazhuang 050043, China)

Abstract: In order to improve the data processing capacity of application software based on embedded operating system μC/OS-III. This paper focused on the porting method of mutex, memory allocation and VFS three subsystems by analyzing SQLite′s operating system interface, provided the technology of porting SQLite to μC/OS-III, and validated the feasibility of the database porting combined with instance.

Key words: database; OS interface; mutex; memory allocation; VFS

1 引言

目前, 传统的小型嵌入式系统的数据处理主要采用文件方式, 它主要存在以下缺点: 一是加大了应用软件开发的难度和代价; 二是数据共享性差; 三是独立性、语义性、可移植性差, 造成了软件的可重用性差、系统成本高; 四是数据管理能力有限, 对大量数据的查询和统计能力不足[1]. 随着小型嵌入式系统的广泛应用以及用户对数据处理和管理需求的日渐提高, 基于文件方式的嵌入式系统已很难满足用户的需求. 而嵌入式数据库可以很好地解决文件处理方式的不足. 嵌入式数据库 SQLite 在便携性、易用性、紧凑性、高效性和可靠性方面有突出的表现, 并且它在设计时特别注重移植性[2]. 它没有独立运行的进程, 而是与所服务的应用程序在应用程序进程空间内共生共存. 它的代码与应用程序代码也是在一起的, 或者说是嵌入其中的, 作为托管它的程序的一部分. SQLite 已成功的移植到许多嵌入式平台, 如 QNX、VxWorks、Symbian 和 Windows CE 等操作系统, 并在嵌入式系统中发挥了重要作用. 由于嵌入式实时操作系统μC/OS-III在小型嵌入式系统中的广泛应用, 本文重点针对在μC/OS-III上移植 SQLite 数据库时, 对其操作系统接口的配置问题进行了研究.

2 SQLite数据库的体系结构

SQLite 体系结构如图 1 所示, 主要由以下几个主要的子系统组成: 接口是一个 C 语言库, 即使上层使用的是不同语言的 API(应用程序编程接口), 在底层执行的都是 C 语言库[3]; 从接口接收到命令后传到编译器, 由编译器将这些命令翻译成一种 SQLite 专用的汇编语言代码, 并最终将这种汇编语言代码编写的微程序交给虚拟机处理; 虚拟机主要是用来执行一个为操作数据库文件而设计的抽象的计算机引擎; 后端由

① 基金项目:河北省高等学校科学技术研究青年基金(20111122)
收稿时间:2013-10-16;收到修改稿时间:2013-11-15

B-Tree、pager(SQLite 的一种数据结构)以及操作系统接口组成. B-Tree 模块的职责是排序, 保证快速定位并找到一切有关系的数据. pager 模块根据 B-Tree 的请求从磁盘读取页面或者向磁盘写入页面, 其使用特殊的算法来预测使用哪些页, 从而保证 B-Tree 尽快地工作. 操作系统接口主要是为不同平台的操作而执行的一个底层与操作系统有关的抽象层, 这样使得 SQLite 很容易移植到不同的操作系统上.

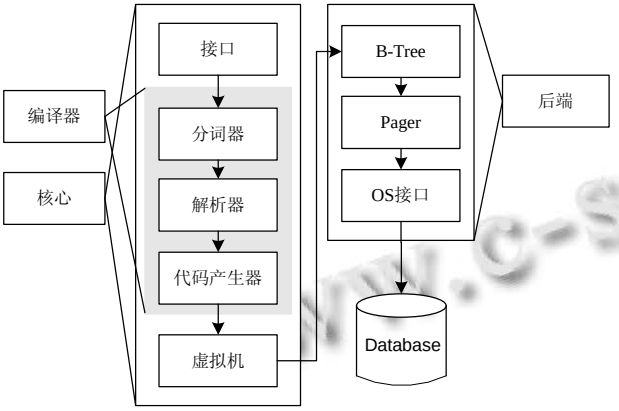

图 1 SQLite 体系结构

3 SQLite的操作系统接口配置

SQLite 数据库的操作系统接口部分实现了一个操作系统抽象层, 为兼容多种操作系统平台而设计. 它由 Mutex(互斥信号量)子系统、内存分配子系统和 VFS(虚拟文件系统)子系统三部分组成[4]. 内存分配子系统用于实现 SQLite 对内存的分配和回收; Mutex 子系统用于多线程环境中各线程间线性使用 SQLite 资源, 信号量的创建需要通过内存分配子系统为其分配内存资源; VFS 子系统用于在 SQLite 和底层操作系统之间提供一个统一的文件操作接口, 对文件的创建、删除等操作同样需要内存分配子系统的支持.

2.1 替换 Mutex 子系统

多线程环境中, SQLite 使用 Mutex 串行访问共享资源. 由于 μC/OS-Ⅲ支持多任务(也称称作线程)管理, 允许应用程序有任意多个任务, 因此, 需设定变量 SQLITE_THREADSAFE=1 以启用 SQLite 对多线程的支持. μC/OS-Ⅲ使用关中断、禁止任务调度、信号量、互斥信号量四种机制实现对共享资源的互斥访问. 而使用互斥信号量是访问共享资源的首选方法, 特别是当某些任务对共享资源的访问有时间要求时[5]. 互斥信号量是一种被定义为 OS_MUTEX 数据类型的内核对象, 该数据类型源自于结构体 os_mutex. μC/OS-Ⅲ操作系统允许用户递归调用互斥信号量以避免优先级的翻转问题, 因而, 配置时可设定变量 DSQLITE_HOMEGROWN_RECURSIVE_MUTEX=1, 以使 SQLite 支持递归信号量. 对互斥信号量操作的 API 函数如表 1 所示.

表 1 互斥信号量 API 函数

函数名称	函数功能
OSMutexCreate()	建立一个互斥信号量
OSMutexDel()	删除一个互斥信号量
OSMutexPend()	等待一个互斥信号量
OSMutexPendAbort()	取消等待
OSMutexPost()	释放或者发布一个互斥信号量

为实现 SQLite 数据库的 Mutex 子系统, 需重新定义 sqlite3_mutex 结构体, 并用 μC/OS-Ⅲ操作系统的信号量操作函数实现对 SQLite 数据库的信号量操作函数的封装. 重新定义后的 sqlite3_mutex 结构体和需要封装的指针函数 sqlite3DefaultMutex()如下:

```
struct sqlite3_mutex {
  /* μC/OS-Ⅲ互斥信号量*/
  OS_MUTEX  ucos_mutex;
  int  id;      //mutex 类型
  int  trace ;  //mutex 改变标志
};
sqlite3_mutex_methods const
*sqlite3DefaultMutex(void){
  static const sqlite3_mutex_methods sMutex = {
    ucosMutexInit,  //初始化 mutex 子系统
    ucosMutexEnd,   //关闭 mutex 子系统
    ucosMutexAlloc,  //创建信号量
    ucosMutexFree,   //释放信号量
    ucosMutexEnter,  //请求信号量
    ucosMutexTry,    //请求信号量
    ucosMutexLeave,  //发送信号量
  };
  return &sMutex;
}
```

SQLite 提供了动态分配和静态分配 mutex 两种机制, 并且有 8 种 mutex 类型(2 种动态类型、6 种静态类型). 为实现动态分配 mutex, 需使用 sqlite3 Malloc Zero()函数创建一个指向 sqlite3_mutex 的指针, 并通

过 μC/OS-Ⅲ提供的 OSMutexCreate()函数创建 OS_MUTEX 对象. 对于静态分配 mutex, 可创建一个静态的 sqlite3_mutex 类型的数组来存储. mutex 的分配函数主要代码如下:

```
sqlite3_mutex *ucosMutexAlloc(int iType){
  /*指向 mutex 结构体的指针*/
  sqlite3_mutex *p = NULL;
  /*定义存储 6 种静态指针的数组*/
  static sqlite3_mutex staticMutexes[6] = {
    SQLITE3_MUTEX_INITIALIZER,
      ……
    SQLITE3_MUTEX_INITIALIZER
};
switch( iType ){
  /*动态分配 mutex*/
  case SQLITE_MUTEX_FAST:
  case SQLITE_MUTEX_RECURSIVE: {
    p = sqlite3MallocZero( sizeof(*p) );
    if( p ){
      p->id = iType;
  /*创建 OS_MUTEX 类型的互斥信号量*/
      OSMutexCreate(&p-> ucos_mutex, “My Mutex”, &err);
      }
  break;
}
/*静态分配 mutex*/
default: {
        p = &staticMutexes[iType-2];
        p->id = iType;
        break;
    }
  }
return p;
}
```

2.2 配置内存分配子系统

SQLite 默认的内存分配器可适应绝大多数应用程序, 虽然可使用其提供的标准 C 函数 malloc()、realloc()和 free()来进行内存分配和回收等功能, 但在嵌入式实时操作系统中, 这样做的结果会产生大量的内存碎片, 将一块连续的内存区域逐渐地分割成许多非常小而且彼此又不相邻的内存区域. 另外, 由于内存管理算法的原因, malloc()和 free()函数执行时间是不确定的, 影响了系统的实时性. 显然, SQLite 默认的内存分配器不能满足嵌入式实时操作系统的要求.

在 SQLite 源码中包含了几种不同的内存分配器模块, 可以在编译时选择, 或者在启动时做有限扩展. 当 SQLite 使用 SQLITE_ENABLE_MEMSYS5 选项编译时, 会包含一个不使用 malloc()的可选内存分配器“memsys5”, 这是一个为嵌入式系统而设计的内存分配器[6]. 应用程序在启动时给 SQLite 提供一个足够大的内存缓冲区, 所有内存分配请求的大小都舍入到 2 的整数次幂, 本算法可以为避免内存碎片和内在崩溃提供数学保证[7].

为启用“memsys5”内存分配器, 应用程序必须在启动时调用 SQLite 数据库提供的 sqlite3_config (SQLITE_CONFIG_HEAP, pBuf, szBuf, mnReq)接口. sqlite3_config()接口用以配置 SQLite 以适应特殊应用的需要, 其中第一个参数是一个整型的配置选项以标识内存分配器, 随后的参数变化情况取决于第一个参数. 当配置选项是 SQLITE_CONFIG_HEAP 时, 应用程序须根据需要指定请求内存的起始地址 pBuf, 内存字节数 szBuf 和一次分配的最小字节数 mnReq 来完成内存申请.

2.3 添加新的虚拟文件系统

从 3.5.0 版本开始, SQLite 支持虚拟文件系统 VFS. SQLite 为了能够更方便地做到支持各种操作系统, 设计了一套统一的文件操作接口. SQLite 数据库的 VFS 接口主要是在 os.h 文件中声明, 并在 os.c 文件中进行封装. 其 VFS 通过编写 sqlite3_vfs、sqlite3_io_methods 和 sqlite3_file 三个结构体来实现. sqlite3_vfs 定义了 VFS 的名称和核心方法, 如检查文件是否存在、删除文件、创建文件、打开文件和规范文件名, 同时还包括挂起进程和获取当前日期和时间等方法. sqlite3_file 代表一个打开的文件, sqlite3_vfs 的 xOpen()方法在文件被打开时构建一个 sqlite3_file 对象, 并保持跟踪打开文件的状态. sqlite3_io_methods 包含方法执行具体的操作, 例如从文件读取和写入, 截取文件, 清空缓存, 获取文件大小, 上锁和解锁文件, 关闭文件并销毁 sqlite3_file 对象等.

由于 μC/OS-Ⅲ操作系统是面向中小型嵌入式系统的, 所以系统本身没有提供文件系统. 然而, μ

C/OS-Ⅲ具有良好的扩展性能，用户可以根据需要自行添加文件系统. 嵌入式文件系统μC/FS 是为资源限制型嵌入式应用而设计，它采用模块化结构，为不同的硬件体系提供多样化的配置模块[8]，它具有紧凑、可靠、高性能等特点，并且默认提供支持μC/OS-Ⅲ操作系统的接口，可以很好的移植到μC/OS-Ⅲ操作系统上. μC/FS 文件系统的架构如图 2 所示.

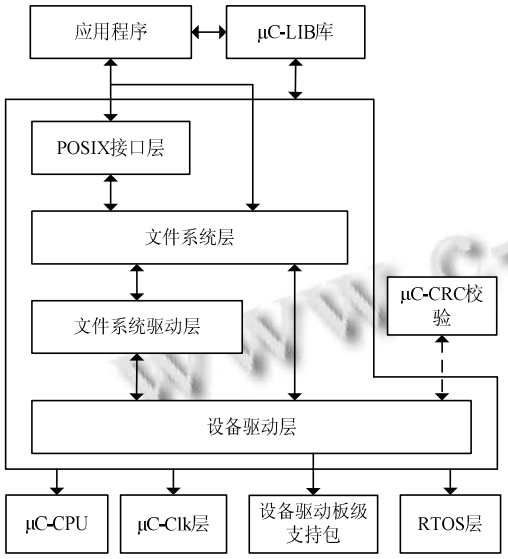

图 2 μC/FS 文件系统架构

μC/FS 文件系统为用户提供了丰富的文件操作接口，移植时需利用μC/FS 的接口函数实现对 VFS 文件操作函数的封装. 大量的文件操作函数的实现无疑会加大 VFS 移植的难度. 然而，由于μC/FS 的 POSIX API 遵循 POSIX 标准，并与 UNIX 的文件操作函数接近，并且 SQLite 实现了对类 UNIX 操作系统(如 Linux, MacOSX, VxWorks 等)的支持，为避免逐一实现文件系统的抽象接口，可充分仿照和利用现有的 SQLite 的 UNIX 系统实现接口，以减少移植工作量. 对 ucosFile 结构体重新封装后的代码如下:

```
struct ucosFile {
  //sqlite3_io_methods 结构体指针文件操作方法
  const sqlite3_io_methods *pMethod;
  sqlite3_vfs  *pVfs;          //sqlite3_vfs 结构体指针用以打开文件
  FS_FILE *p_file;          //访问文件的句柄
  const char *zPath;          //文件路径
  unsigned char locktype;    //当前文件锁的类型
};
```

SQLite 利用操作系统的功能来完成对数据库的锁机制实现，其实现机制在不同操作系统上是不同的. 它在 Linux 上使用 POSIX 建议锁，在 Windows 上使用 LockFile()、LockFileEx()、UnLockFile() 系统调用. SQLite 提供了共享锁、保留锁、未定锁、排他锁 4 种锁实现对数据库的共享读和互斥写操作. μC/FS 提供的文件锁函数 fs_flockfile()、fs_ftrylockfile()、fs_funlockfile()不能完全满足 SQLite 的 4 种锁机制的实现. 为此仿照 SQLite 对嵌入式操作系统 VxWorks 的锁实现机制，将 SQLite 的 4 种锁归为一种排他锁，同一时刻只允许一个任务对数据库进行读或写操作. 这样虽然降低了文件操作的并发性，却简化了锁机制的实现. 其中重新编写的对文件上锁函数的主要代码如下:

```
static int ucosLock(sqlite3_file *id, int eFileLock) {
/*文件上锁只能由低级别锁向高级别锁转变*/
    unixFile *pFile = (unixFile*)id;
    FS_FILE *p_file = pFile->p_file;
    int    rc = SQLITE_OK;
    /*文件已经获得排他锁*/
      if (pFile->eFileLock > NO_LOCK) {
        pFile->eFileLock = eFileLock;
        rc = SQLITE_OK;
        goto ucos_end_lock;
    }
/*调用μC/FS 提供的文件上锁函数 fs_flockfile()*/
    if( fs_flockfile(p_file) == 0 ){
      /*文件上锁失败*/
      rc = SQLITE_BUSY;
      goto ucos_end_lock;
    }
    /*给文件上锁*/
    pFile->eFileLock = eFileLock;
    ucos_end_lock:
    return rc;
}
```

3 移植测试

以基于 Zigbee 技术的温室环境监控系统作为数据库移植测试用例，通过对温室环境中的温度、湿度的采集存储和读取，并以上位机软件 LabVIEW 显示来验证 SQLite 数据库移植的可行性. 系统由以 PC 作为上

位机、以 STM32F103 控制芯片和 Zigbee 网络组成下位机. 系统总体架构如图 3 所示.

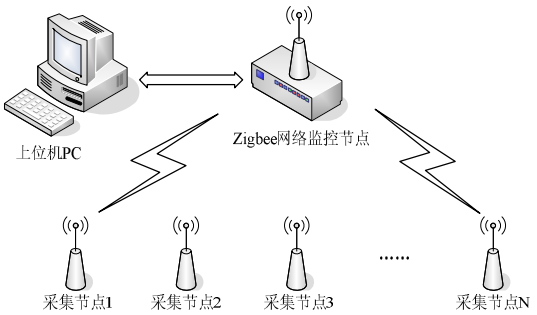

图 3 监控系统架构

其中, 监控节点以 STM32F103 处理器和 CC2530 芯片为硬件平台, 移植 μC/OS-III操作系统、μC/FS 文件系统和 SQLite 数据库组成软件平台. 系统新建任务包括系统主任务, 串口通信任务、温度值读任务和写任务、湿度值读任务和写任务, 系统空闲任务以及统计运行时间任务, 并为不同任务设置不同的优先级以检验 SQLite 在多任务环境中对共享资源的访问控制性能. 系统以规定时间间隔(30 分钟)采集数据, 以 1 号采集节点数据为例, 一天的统计结果如图 4 所示.

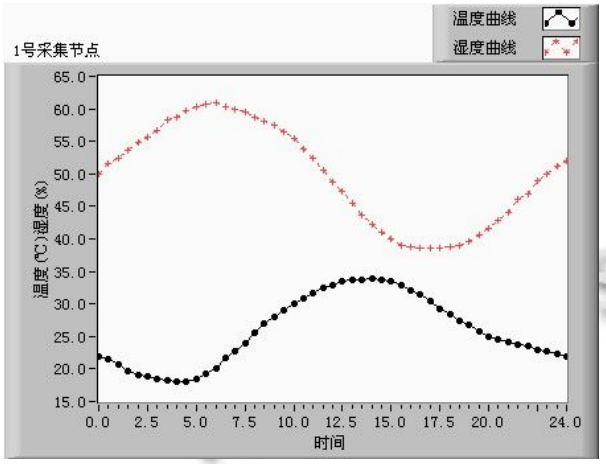

图 4 测试结果

测试结果显示的数据值与实际测量统计数据相同, 说明系统完成了对数据库的读和写操作, 并且各个任务之间对数据库的操作是互斥的, 保证了数据的完整性. 与其它嵌入式平台(如 Windows CE、Linux)相比, 虽然本移植工作简化了文件的锁实现机制, 文件操作的并发性有所降低, 但对于数据量较小的小型嵌入式系统而言, SQLite 数据库的高效性可以一定程度上的弥补其不足.

4 结语

通过对 SQLite 数据库的操作系统接口的配置, 将其移植到μC/OS-III操作系统上, 并通过对温室环境中的温度、湿度的读写控制验证了移植的可行性. 然而, 对于移植后的数据库效率和稳定性等性能还有待进一步验证, 移植良好的用户交互接口也是后续工作的主要研究内容. 总之, 将 SQLite 数据库移植到μC/OS-III操作系统上无疑将提高小型嵌入式系统对数据的管理能力, 降低嵌入式应用软件的开发难度和成本.

参考文献

1 韩善锋,曹凤海,易昌华.SQLite 数据库在嵌入式程序开发中的应用.物探装备,2011,21(3):170–173,178.

2 Allen G, Owens M.杨谦,刘义宣,谢志强,译. SQLite 权威指南(第二版).北京:电子工业出版社,2012.

3 韩太东,卢秉亮,朱健,等.嵌入式数据库 SQLite 在 Windows 程序中的应用.沈阳航空工业学院学报,2009,26(4):61–64.

4 Hipp DR. Custom Builds Of SQLite or Porting SQLite To New Operating Systems. http://www.sqlite.org/custombuild.html. 2013-06-26.

5 Labrosse JJ.宫辉,曾鸣,龚光华,等译. 嵌入式实时操作系统μC/OS-III(第二版).北京:北京航天航空大学出版社,2012.

6 Hipp DR. Dynamic Memory Allocation In SQLite. http://www.sqlite.org/malloc.html. 2013-07-18.

7 Robson JM. Bounds for Some Functions Concerning Dynamic Storage Allocation. ACM, 1974, 21(3): 491–499.

8 李家良,慕德俊.基于 SD 卡的μC/FS 文件系统移植研究. 微处理器,2010,(6):79–81.