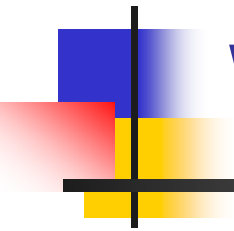
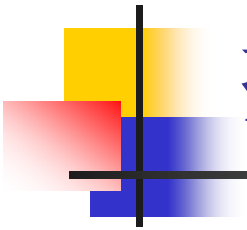


Windows CE 嵌入式系统实例分析



提纲

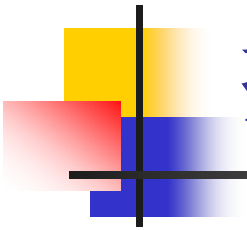
- WinCE 嵌入式系统原理
- 动态链接库的加载分析
- 线程在队列之间转换分析
- 虚存分配
- 文件系统的建立及访问过程
- 驱动程序加载

WinCE 嵌入式系统原理

- 概念：
  - 嵌入式系统是不同于常见计算机系统的一种计算机系统，它不以独立设备的物理形态出现。
- 分类：
  - 通用型的嵌入式操作系统如Windows CE、VxWorks、
  - 嵌入式Linux等和专用型的嵌入式操作系统如Palm OS、Symbian等

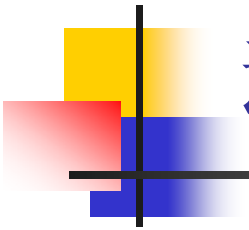
WinCE 嵌入式系统原理

- 组成及原理
  - 从体系结构上看，嵌入式系统主要由嵌入式处理器、支撑硬件和嵌入式软件组成。
  - 其中嵌入式处理器通常是单片机或微控制器;支撑硬件主要包括存储介质、通信部件和显示部件等;嵌入式软件则包括支撑硬件的驱动程序、操作系统、支撑软件以及应用中间件等。
  - 它的部件根据主体设备以及应用的需要嵌入在设备的内部，发挥着运算、处理、存储以及控制作用。



提纲

- WinCE 嵌入式系统原理
- 动态链接库的加载分析
- 线程在队列之间转换分析
- 虚存分配
- 文件系统的建立及访问过程
- 驱动程序加载



动态链接库的加载分析

- loader.c中程序代码的组织结构
- module structure
- LoadOneLibraryPart2加载DLL的过程
- DLL加载过程 — InitModule的执行
- 实例分析

loader.c中程序代码的组织结构

- **Win32 LoadLibrary call**

  HANDLE SC_LoadLibraryW(LPCWSTR lpszFileName)

  HINSTANCE SC_LoadLibraryExW(LPCWSTR lpLibFileName, HANDLE hFile, DWORD dwFlags)

  HANDLE SC_LoadDriver(LPCWSTR lpszFileName)

  HANDLE SC_LoadKernelLibrary(LPCWSTR lpszFileName)

  HANDLE SC_LoadIntChainHandler(LPCWSTR lpszFileName, LPCWSTR lpszFunctionName, BYTE bIRQ)

loader.c中程序代码的组织结构

- **Win32 FreeLibrary call**
  - HANDLE SC_FreeIntChainHandler(HANDLE hLib)

    BOOL SC_FreeLibrary(HANDLE hInst)
  - 主要是FreeLibrary函数。这个API函数负责卸载DLL。它呼叫FreeOneLibrary这个函数，函数原型如下：

    BOOL FreeOneLibrary(PMODULE pMod,
    BOOL bCallEntry)

    卸载的过程主要是由FreeOneLibraryPart2负责

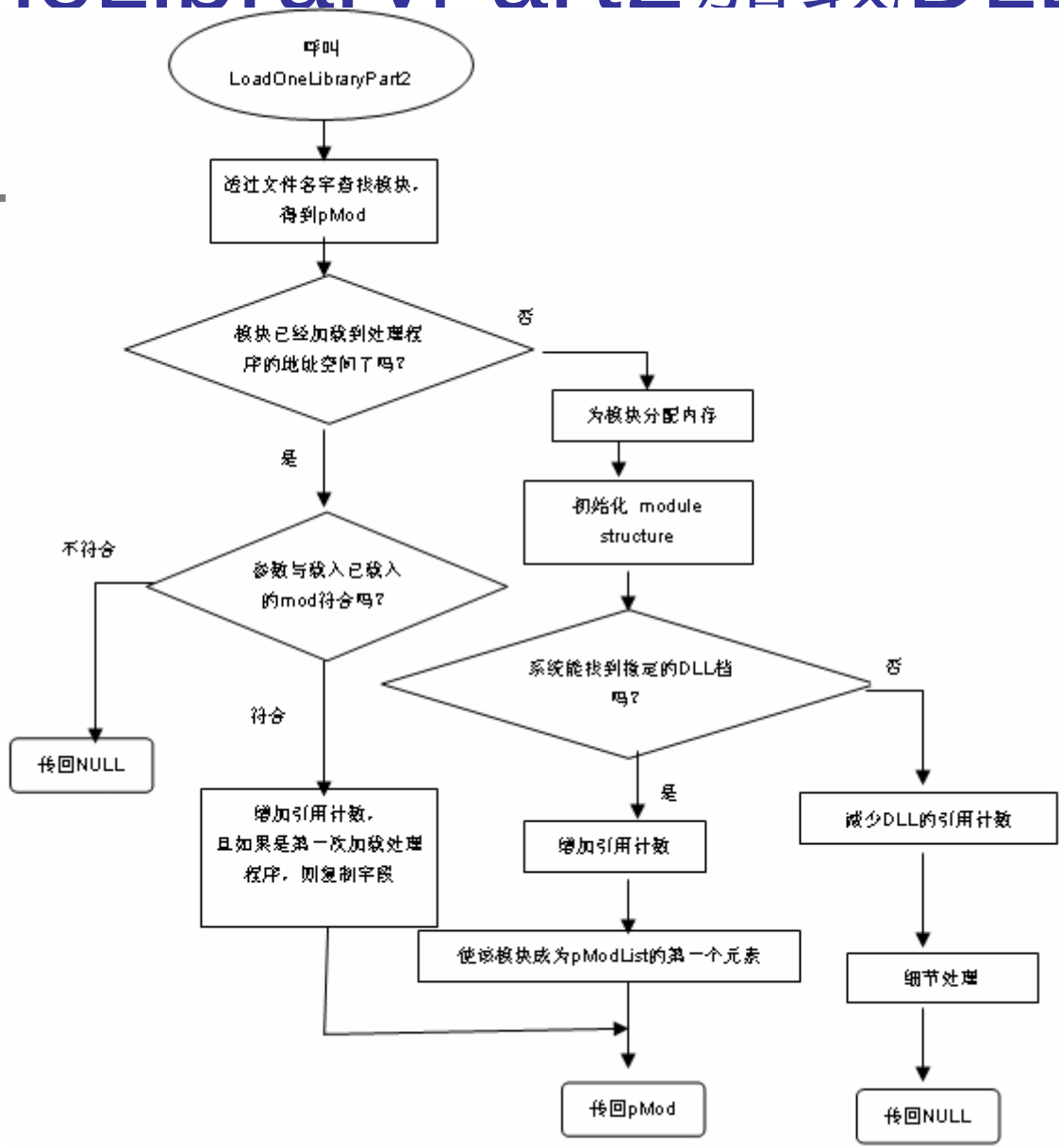
loader.c中程序代码的组织结构

- **Win32 GetProcAddress call**
  - LPVOID SC_GetProcAddressA(HANDLE hInst, LPCSTR lpszProc)
  - LPVOID SC_GetProcAddressW(HANDLE hInst, LPCSTR lpszProc)

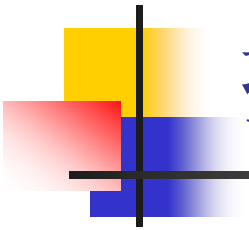
module structure

```
typedef struct Module {
    LPVOID lpSelf;                  /* 用于验证，指向自己的指标 */
    PMODULE pMod;                   /* 链接串行中的下一个模块 */
    LPWSTR lpszModName;             /* 模块名字 */
    DWORD inuse;                    /* 使用状况的位向量 */
    DWORD calledfunc;               /* 被呼叫的进入点，但不退出 */
    WORD refcnt[MAX_PROCESSES];     /* 处理程序的引用计数 */
    LPVOID BasePtr;                 /* DLL 载入基址 */
    DWORD DbgFlags;                 /* 侦错旗标 */
    LPDBGPARAM ZonePtr;             /* Debug zone 指标 */
    ulong startip;                  /* 基于 0 的进入点 */
    openexe_t oe;                   /* 执行档指标 */
    e32_lite e32;                   /* e32 标头 */
    o32_lite *o32_ptr;              /* O32 串行指标 */
    DWORD breadcrumb;
    DWORD dwNoNotify;               /* 每个处理程序对应一个位，当 Notify 被禁用时设为 1 */
    WORD wFlags;
    BYTE bTrustLevel;
    BYTE bPadding;
    PMODULE pmodResource;           /* 包含资源的模块 */
    DWORD rwLow;                    /* ROM DLL 中可擦写段的基地址 */
    DWORD rwHigh;                   /* ROM DLL 中可擦写段的高位地址 */
} Module;
```

LoadOneLibrarvPart2加载DLL的过程



DLL加载过程 — InitModule的执行



- 呼叫OpenADll，产生线程指标 (openexe_t)
- 设定module的e32 标头信息
- DLL的内存配置与Module->BasePtr的取得
- name和o32对象的内存配置
- 复位位映射
- EXE的起始IP

呼叫OpenADII，产生线程指标(openexe_t)

openexe_t structure

```
typedef struct openexe_t {
    union {
        int hppfs;                  // ppfs handle
        HANDLE hf;                  // 对象储存指针
        TOCentry *tocptr;           // rom entry pointer
    };
    BYTE filetype;                  //档案类型
    BYTE bIsOID;
    WORD pagemode;                  //分页模式
    DWORD offset;                   //偏移
    union {
        Name *lpName;
        CEOID ceOid;
    };
} openexe_t;
```

呼叫OpenADll，产生线程指标(openexe_t)（续）

- OpenADll呼叫OpenExe，由OpenExe呼叫SafeOpenExe，设定现在指标的工作基本上都在SafeOpenExe中完成。函数如下：

```
BOOL SafeOpenExe(LPWSTR lpszName,
    openexe_t *oeptr, BOOL bIsDLL, BOOL
    bAllowPaging,          OEinfo_t *poeinfo)
```

呼叫OpenADll，产生线程指标(openexe_t)（续）

- SafeOpenExe还执行以下的工作：
  - 寻找EXE文件所在的目录
  - 按照指定路径寻找文件
  - 在Windows目录中寻找文件
  - 在根目录中寻找文件

设定module的e32 标头信息

e32_lite structure

```
typedef struct e32_lite {
        unsigned short    e32_objcnt;          /* PE 32-bit .EXE header */
        BYTE              e32_cevermajor;      /*  内存对象个数  */
        BYTE              e32_ceverminor;      /*  版本信息  */
        unsigned long     e32_stackmax;        /*  版本信息  */
        unsigned long     e32_vbase;           /*  堆栈的最大值  */
        unsigned long     e32_vsize;           /* module 的虚拟内存基地址  */
        unsigned long       e32_sect14rva;     /*  整个映射的 Virtual  大小  */
        unsigned long       e32_sect14size;    /* section 14 rva */
        struct    info      e32_unit[LITE_EXTRA];  /* section 14 size */
} e32_lite, *LPe32_list;                       /* Array of extra info units */
```

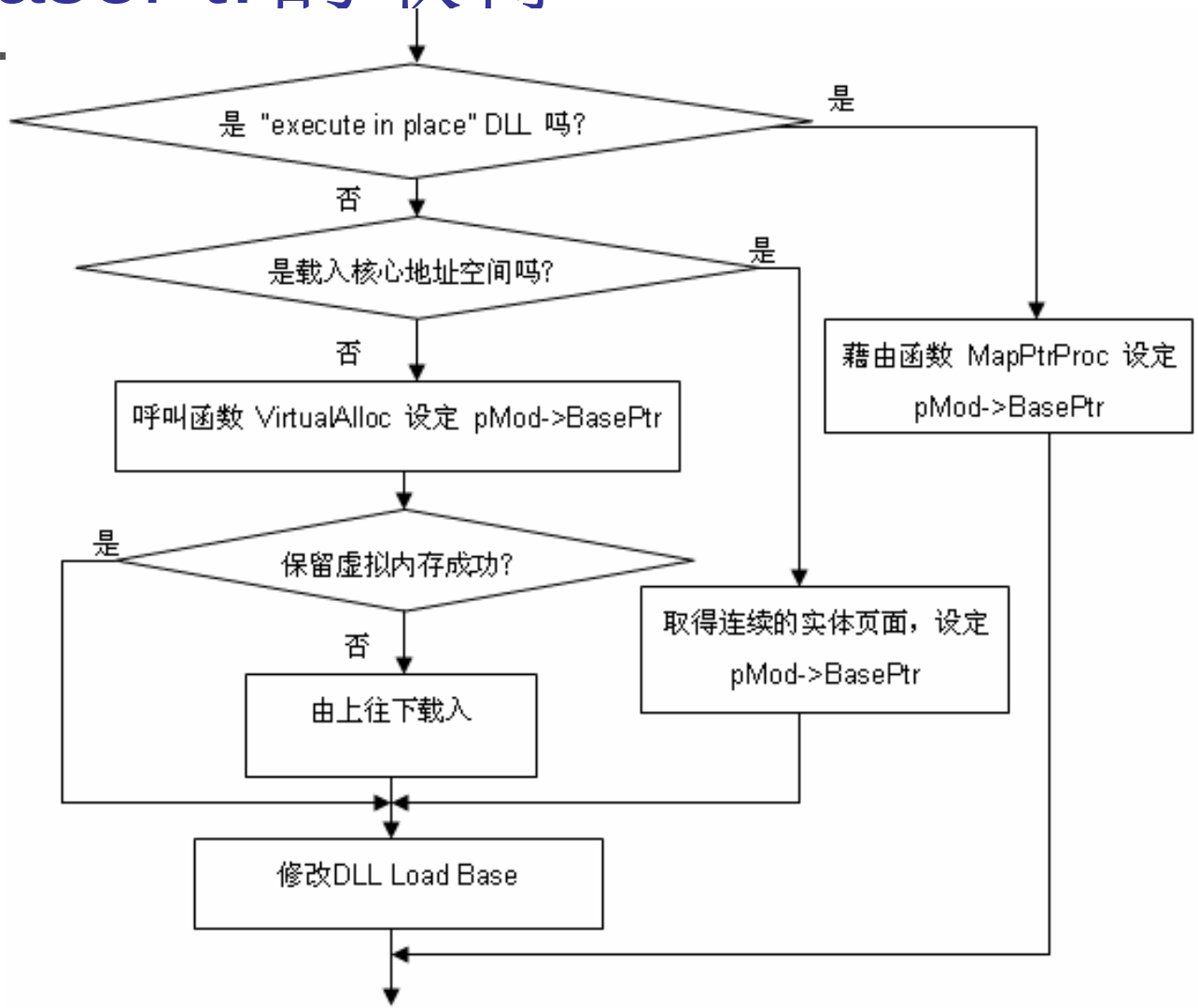
设定module的e32 标头信息（续）

- 这是与内存相关的一组数据，在InitModule中有一段程序代码如下：

```
// load O32 info for the module
eptr = &pMod->e32;
if (retval = LoadE32(&pMod->oe, eptr, &flags, &entry,
        (pMod->wFlags &
        LOAD_LIBRARY_AS_DATAFILE) ? 1 : 0,
bAllowPaging, &pMod->bTrustLevel)) {
        return retval;
}
```

它由呼叫LoadE32函数读取DLL文件，设定e32标头的各部分内容，设定堆栈、虚拟内存基地址、映像等正确的值，使接下来DLL的内存配置工作能够顺利进行。

DLL的内存配置与Module->BasePtr的取得



DLL的内存配置与Module->BasePtr的取得（续）

- DLL加载到核心地址空间 (kernel space) 或是使用者空间 (user space) 也有差别

```
if (wFlags & LOAD_LIBRARY_IN_KERNEL) {
        PHYSICAL_ADDRESS paRet;
        // Loading in the kernel address space
        paRet = GetContiguousPages((DWORD) (eptr->e32_vsize + PAGE_SIZE - 1) / PAGE_SIZE,
                0, 0);
        if (paRet == INVALID_PHYSICAL_ADDRESS || !(pMod->BasePtr =
                (PVOID) Phys2Virt(paRet))) {
                return ERROR_OUTOFMEMORY;
        }
}
```

DLL的内存配置与Module->BasePtr的取得（续）

- 如果不是加载核心地址空间，则必须为其保留地址空间，以避免其它DLL加载同样的地址空间，而使该DLL的卸载发生问题。程序代码如下：

```
} else {
    // try to honor the Dll's relocation base to prevent relocation
    if ((pTOC->ulKernelFlags & KFLAG_HONOR_DLL_BASE) && (eptr->e32_vbase +
        eptr->e32_vsize < ROMDllLoadBase)) {
        pMod->BasePtr = VirtualAlloc ((LPVOID)(ProcArray[0].dwVMBase +
            eptr->e32_vbase), eptr->e32_vsize, MEM_RESERVE | MEM_IMAGE,
            PAGE_NOACCESS);

        DEBUGMSG (pMod->BasePtr, (L"Loading DLL '%s' at the preferred loading address
            %8.8lx\n", lpszFileName, ZeroPtr (pMod->BasePtr)));
    }
```

name和o32对象的内存配置

o32 对象是与存取控制有关的对象。结构定义如下。

```
typedef struct o32_lite {
        unsigned long           o32_vsize;
        unsigned long           o32_rva;
        unsigned long           o32_realaddr;
        unsigned long           o32_access;
        unsigned long           o32_flags;
        unsigned long           o32_psize;
        unsigned long           o32_dataptr;
} o32_lite, *LPo32_lite;
```

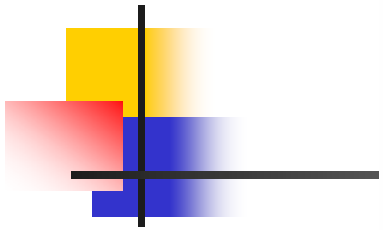
复位位映射

非 XIP 映射需要重新寻址（Relocate）。

```
if (pMod->oe.filetype != FT_ROMIMAGE) {
    //
    // Non-XIP image needs to be relocated.
    //
    if ((pMod->oe.pagemode == PM_NOPAGING) &&
        !(pMod->wFlags & LOAD_LIBRARY_AS_DATAFILE) &&
        !Relocate (eptr, pMod->o32_ptr, (ulong)pMod->BasePtr,
        ((wFlags & LOAD_LIBRARY_IN_KERNEL) ? 0 : ProcArray[0].dwVMBase))) {
        return ERROR_OUTOFMEMORY;
    }
```

复位位映射（续）

- module被加载内存的Slot1 (DLL高地址区域)
- 或者加载到核心中，则需要记录为这个module而寻址的读写区
- 程序如下页：

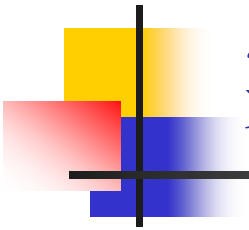


```
o32_lite *optr = pMod->o32_ptr;
//
// If the module is loaded into the Slot 1 (DLL High) area or into
// the kernel we need to record where the read/write section has
// been located for this module.
//
if (IsModCodeAddr (pMod->BasePtr) || IsKernelVa(pMod->BasePtr)) {
        // find the high/low of RW sections
        for (loop = 0; loop < eptr->e32_objcnt; loop ++, optr ++) {
                if ((optr->o32_flags & IMAGE_SCN_MEM_WRITE) && !(optr->o32_flags &
                        IMAGE_SCN_MEM_SHARED)) {
                        if (pMod->rwLow > optr->o32_realaddr)
                                pMod->rwLow = optr->o32_realaddr;
                        if (pMod->rwHigh < optr->o32_realaddr + optr->o32_vsize)
                                pMod->rwHigh =    optr->o32_realaddr + optr->o32_vsize;
                }
        }
}
```

EXE的起始IP

这里是藉由呼叫函数 FindEntryPoint ， 来设定 pMod->startip。

```
if (entry) {
        if ((wFlags & LOAD_LIBRARY_IN_KERNEL) || !pMod->e32.e32_sect14rva)
                pMod->startip = FindEntryPoint(entry,&pMod->e32,pMod->o32_ptr);
        else {
                HANDLE hLib;
                if (!(hLib = LoadOneLibraryW (L"mscoree.dll",0,0)) || !(pMod->startip =
                        (DWORD)GetProcAddressA (hLib,"_CorDllMain"))) {
                        return ERROR_DLL_INIT_FAILED;
                }
        }
}
```



实例分析

- 范例环境的建立过程
- 启动时加载DLL
- 使用者DLL加载过程小结

范例环境的建立过程

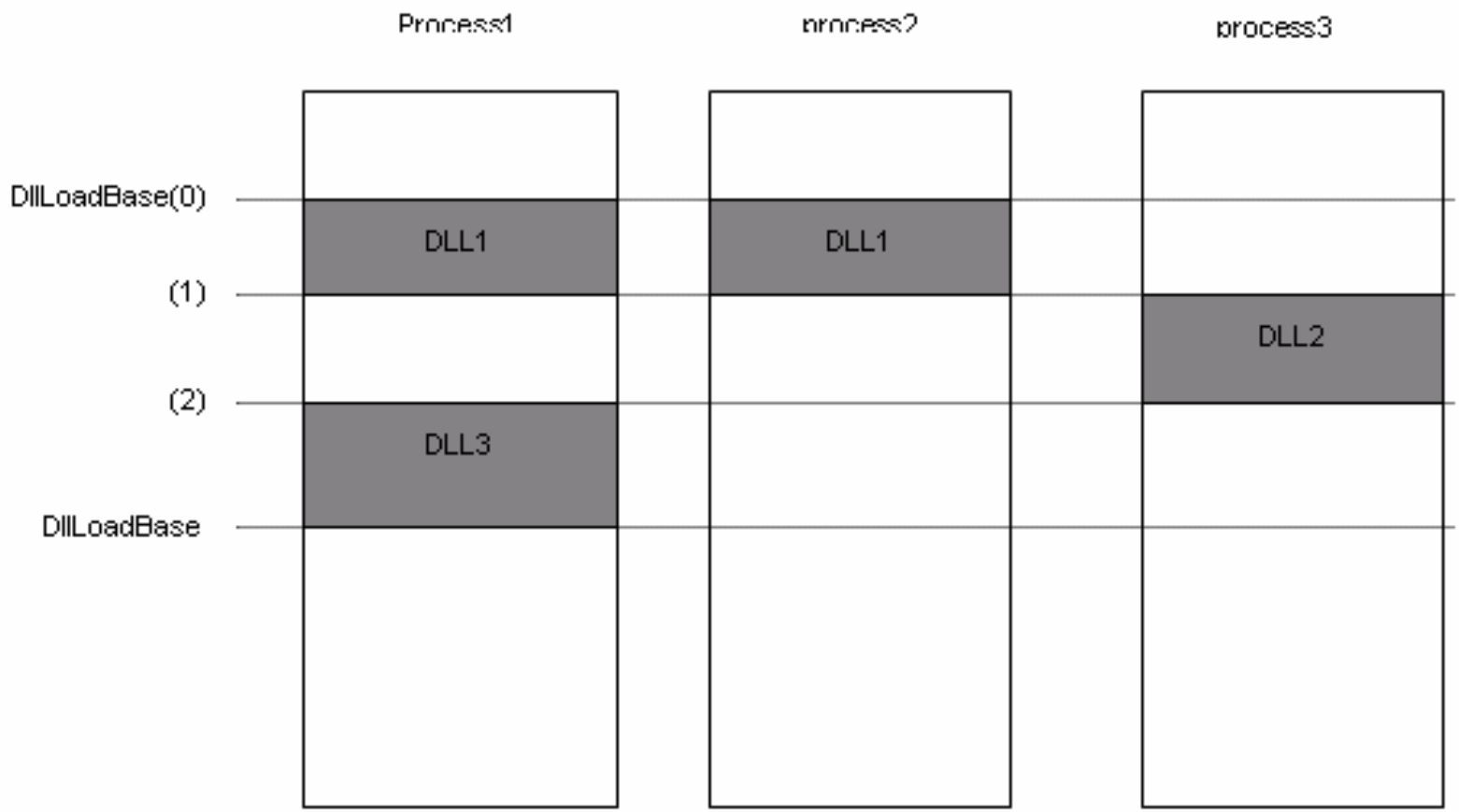
- 在Platform Builder 4.0下，使用其所提供的emulator作为platform的BSP，建立新的platform — tiny kernel。
- Build后产生新的Platform — loader_test，它同时产生debug和release版本。用debug版本侦错，追踪loader.c，可以看到这个用作测试的loader_test启动时加载各个DLL的过程。
- 建立控制台应用程序，编译产生loader_test上的应用程序 — console_test。要注意的是，因为这里建立起来的是tiny kernel，所以不支持一些C链接库函数。当然，你也可以建立其它类型的platform。
- 建立空的动态链接库dll_test，用console_test来呼叫dll_test，追踪DLL载入的过程。主要是看其加载的地址pMod->BasePtr。在Platform Builder的target中看Modules and Symbols窗口，可以看到DLL加载的映像地址范围和重新寻址后的地址范围。

启动时加载DLL

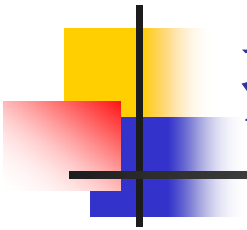
载入的 DLL 和 EXE

Module	Image Address Range	Relocated Data Address Range	Status
coredll.dll	0x03FC0000-0x03FF6FFF	0x01FFF000-0x01FFF780	loaded
filesys.exe	0x04010000-0x04049FFF		loaded
fsdmgr.dll	0x03F80000-0x03F92FFF	0x01FF9000-0x01FF98F0	loaded
Kd.dll	0x80293000-0x802A9FFF	0x80354000-0x8035A584	loaded
Nk.exe	0x80220000-0x802B8FFF	0x80330000-0x8034A7E6	Loaded
relfsd.dll	0x03FB0000-0x03FB7FFF	0x01FFD000-0x01FFDABC	Loaded
shell.exe	0x06010000-0x0601CFFF		Loaded
toolhelp.dll	0x03FA0000-0x03FA3FFF	0x01FFB000-0x01FFB050	loaded

使用者DLL加载过程小结



处理程序加载 DLL 的示意图



提纲

- WinCE 嵌入式系统原理
- 动态链接库的加载分析
- 线程在队列之间转换分析
- 虚存分配
- 文件系统的建立及访问过程
- 驱动程序加载

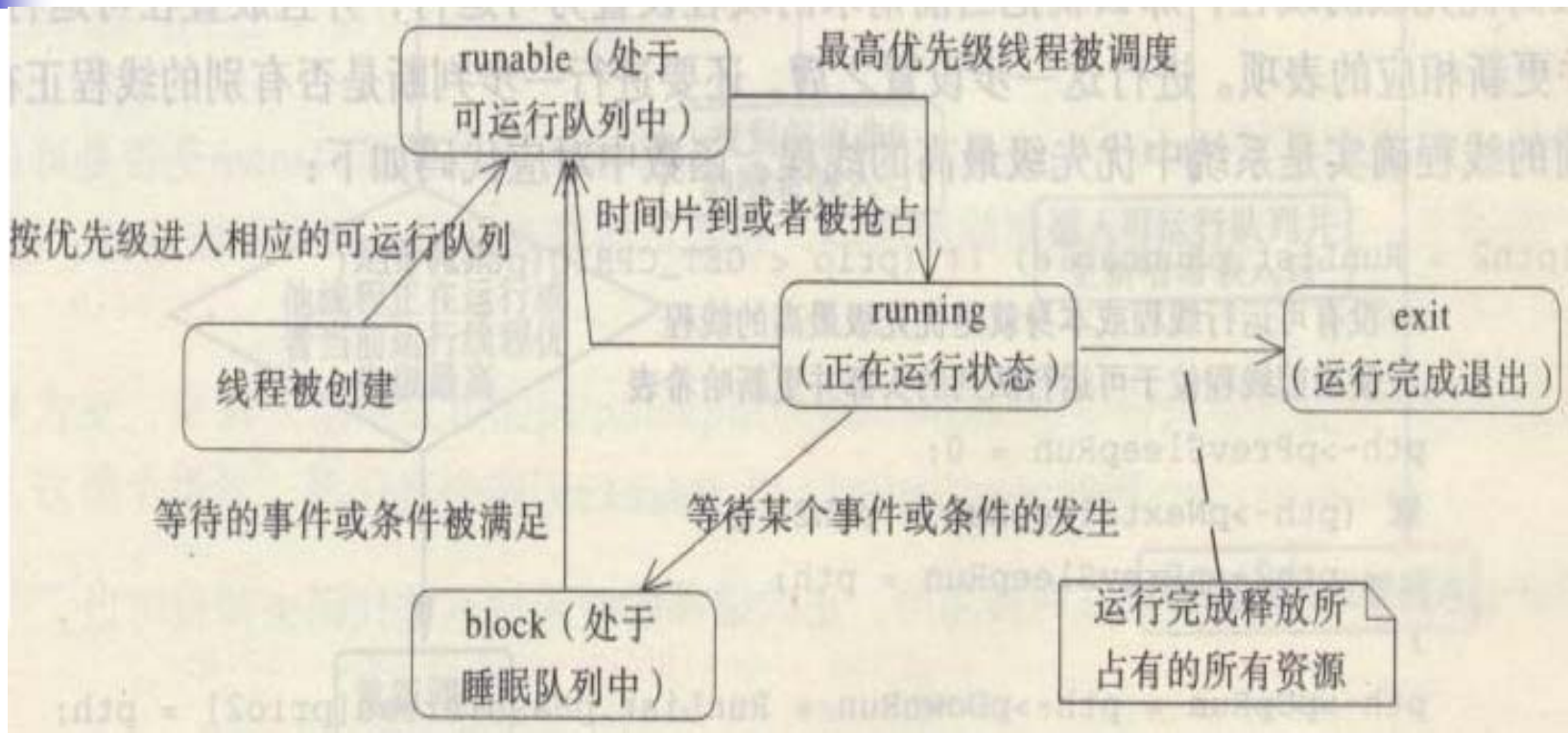
线程在队列之间转换分析

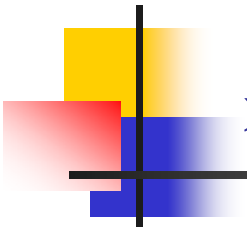
- 概念
- 具体分析

线程在队列之间转换分析

- 线程状态：
  - RUNSTATE_RUNNING　　正在执行
  - RUNSTATE_RUNNABLE　　可以执行
  - RUNSTATE_BLOCKED　　可执行态的停滞态,可能是自愿进入停滞态
  - RUNSTATE_NEEDSRUN　　即将进入可执行状态
  - WAITSTATE_SIGNALLED　　等待某个信号的唤醒
  - WAITSTATE_PROCESSING　　重新处理等待态
  - WAITSTATE_BLOCKED　　等待状态的停滞态

线程状态切换图

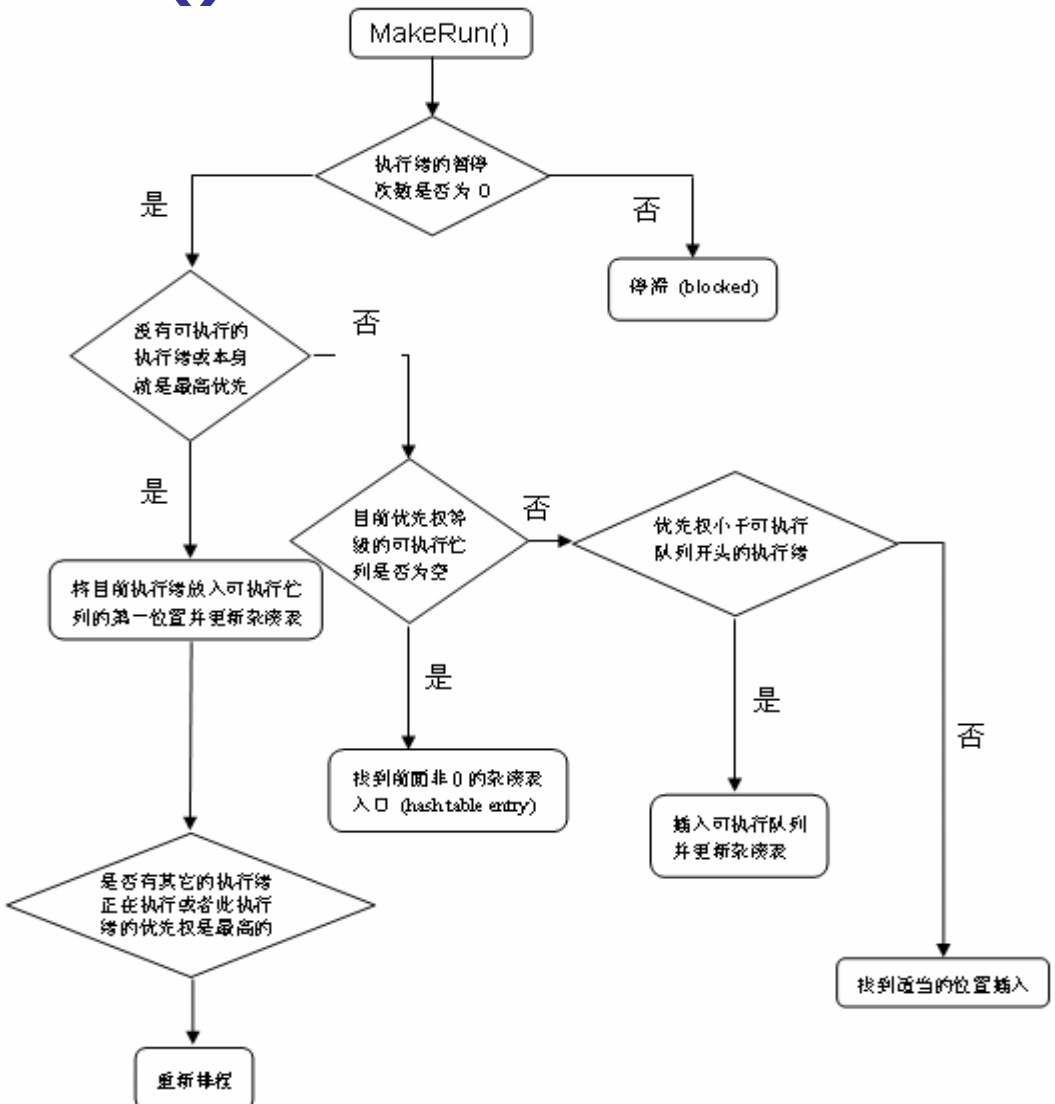

runable（处于
可运行队列中）
最高优先级线程被调度
时间片到或者被抢占
按优先级进入相应的可运行队列
线程被创建
running
（正在运行状态）
exit
（运行完成退出）
等待的事件或条件被满足
等待某个事件或条件的发生
block（处于
睡眠队列中）
运行完成释放所
占有的所有资源



具体分析

- MakeRun()函数分析
- RunqDequeue()函数分析
- SleepqDequeue()函数分析

MakeRun()函数分析



MakeRun()函数分析 （续）

- 对应的程序：

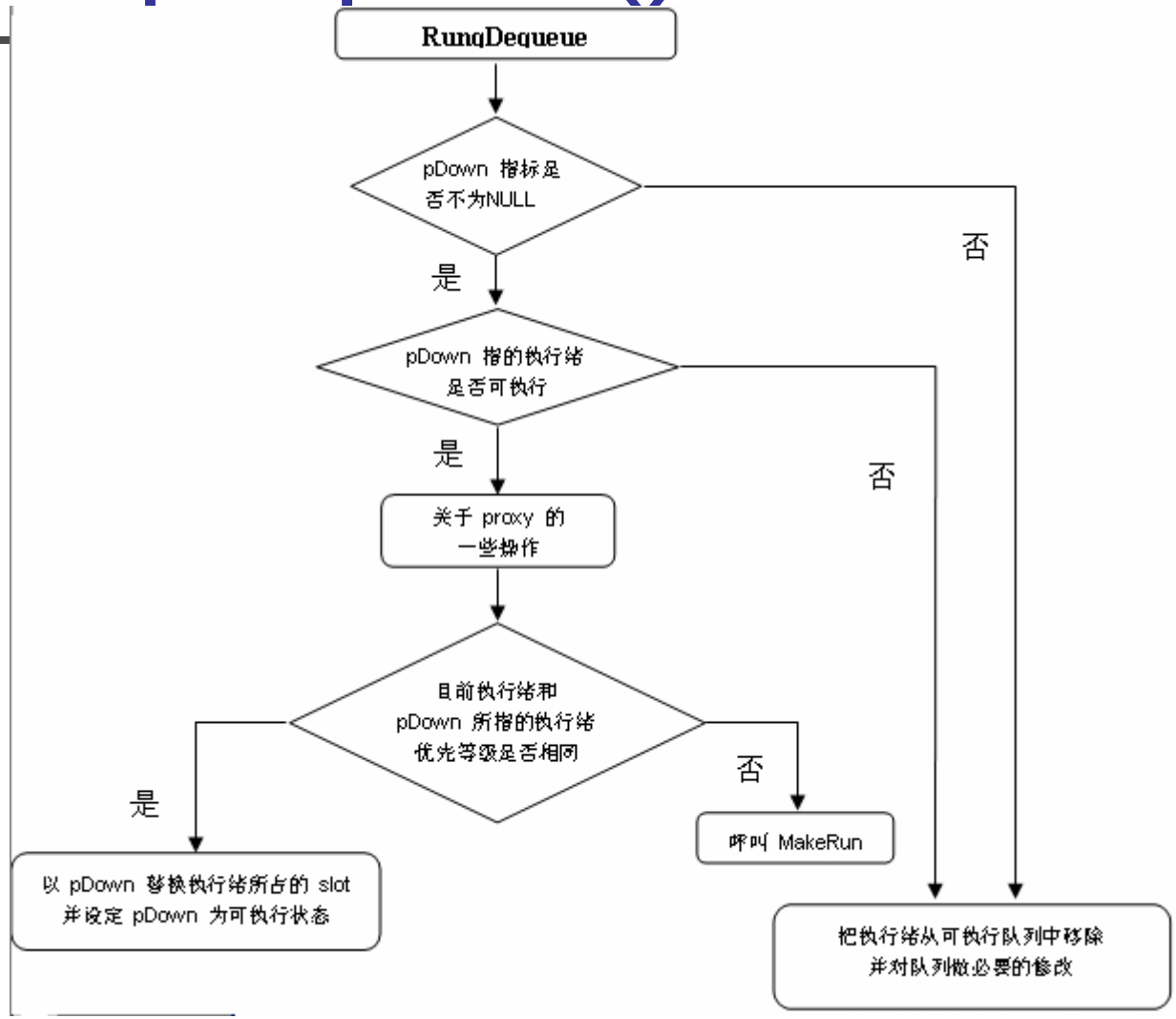
```
if (!(pth2 = RunList.pRunnable) || (prio < GET_CPRIO(pth2))) {
        //没有其它可执行的执行绪或本身就是优先权最高的执行绪
        //使目前执行绪位于可执行队列的开头并更新 hash table
        pth->pPrevSleepRun = 0;
        if (pth->pNextSleepRun = pth2) {
                pth2->pPrevSleepRun = pth;
        }
        pth->pUpRun = pth->pDownRun = RunList.pHashThread[prio2] = pth;
        RunList.pRunnable = pth;

        // 判断是否需要重新排程
        if (!RunList.pth || (prio < GET_CPRIO(RunList.pth)))
                SetReschedule();
}
```

MakeRun()函数分析 （续）

- 最后一步应该考虑到的是：
  - 在设定之后是否有中断发生
  - 是否改变了系统的状态

RunqDequeue()函数分析



RunqDequeue()函数分析 （续）

- 如果pDown与线程的pDownSleep指标所指的线程不相同
  - 则直接删除线程
  - 同时对队列作一些必要的修改。
- 如果相等
  - 下一步要做的就是判断pDown是否是可执行的，
  - 如果不是，则同上一步，直接删除要删除的执行绪，作必要的修改即可。
  - 如果不能执行，则处理一些与proxy相关的操作。

SleepqDequeue()函数分析

- 从等待队列中删除一个线程

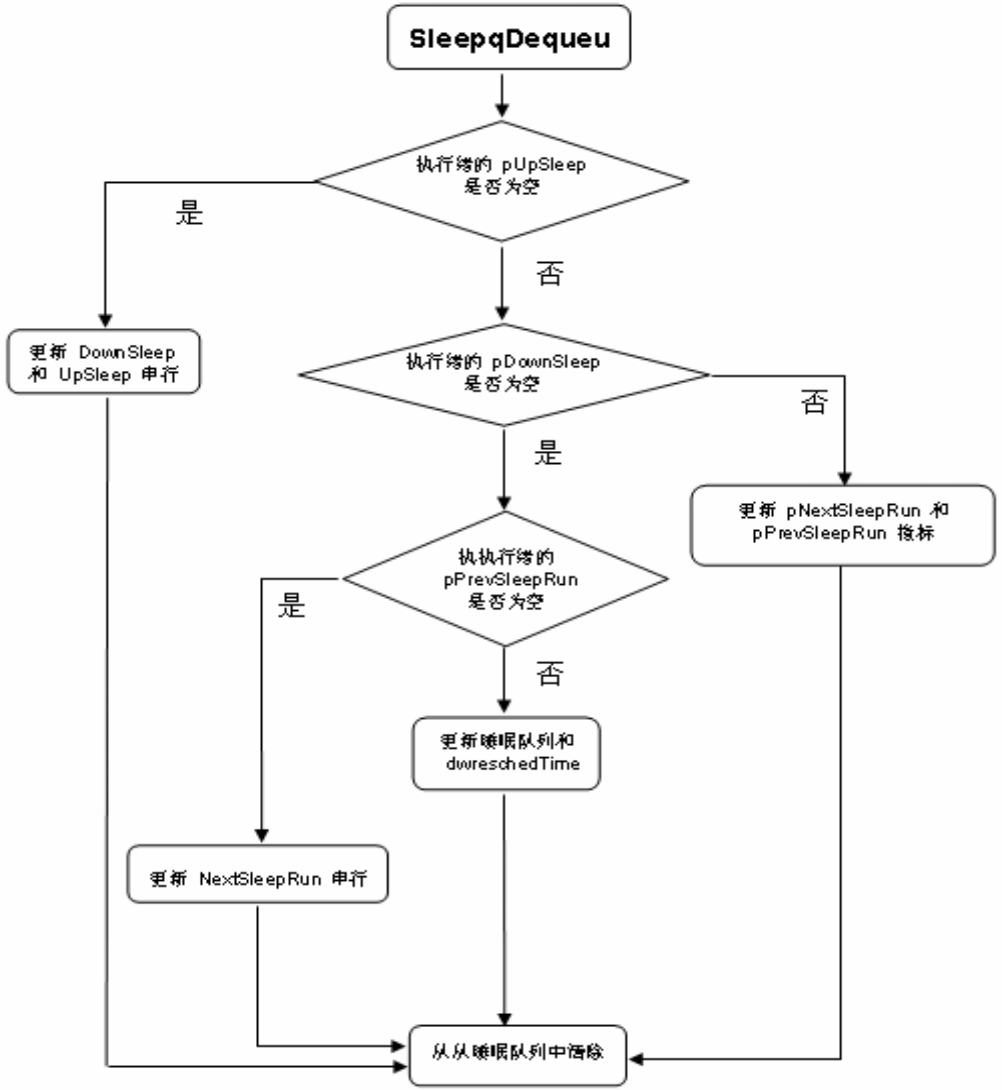
```
if (pth2 = pth->pUpSleep) {
    ……
} else if (pth2 = pth->pDownSleep) {
    ……
} else if (pth2 = pth->pPrevSleepRun) {
    ……
} else {
    ……
}
```

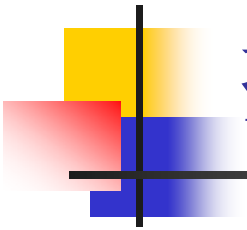
SleepqDequeue()函数分析（续）

- 如果为NULL，更新要删除执行绪的pUpSleep以及pDownSleep，然后直接呼叫CLEAR_SLEEPING(pth)
  - if (pth2->pDownSleep = pth->pDownSleep) {
  - pth2->pDownSleep->pUpSleep = pth2;
  - pth->pDownSleep = 0;
  - }
  - pth->pUpSleep = 0;
  - ……
  - CLEAR_SLEEPING(pth);
- 如果不为NULL，接着判断pth->pDownSleep参数是否为NULL，即下面所示的程序代码：

  } else if (pth2 = pth->pDownSleep) { …… }

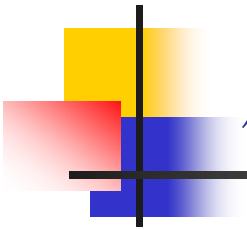
SleepDequeue 函数流程图





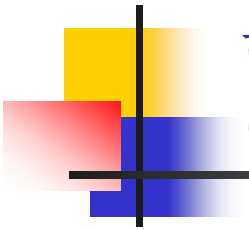
提纲

- WinCE 嵌入式系统原理
- 动态链接库的加载分析
- 线程在队列之间转换分析
- 虚存分配
- 文件系统的建立及访问过程
- 驱动程序加载



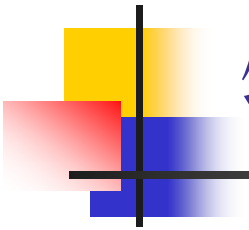
虚存分配

- 配置过程概述
- 物理内存的获取
- 虚拟内存配置原始码片段



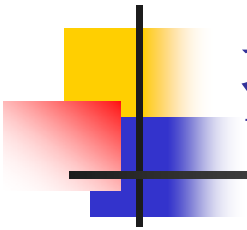
配置过程概述

- 参数验证
- 扫描虚拟内存区域找到合适大小的空闲区块
- 在得到的虚拟内存区块中写入控制信息
- 获取足够的实体区块并建立映像
- 小实验：虚拟内存配置的直观印象



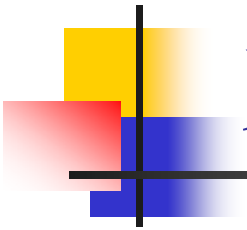
物理内存的获取

- 分配过程
- 小实验：HoldPages函数跟踪



提纲

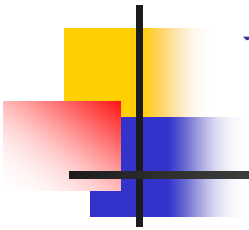
- WinCE 嵌入式系统原理
- 动态链接库的加载分析
- 线程在队列之间转换分析
- 虚存分配
- 文件系统的建立及访问过程
- 驱动程序加载



文件系统的建立及访问过程

- 储存管理器分层结构的建立及可安装文件系统的加载
- FAT文件系统中CreateFile的具体流程

储存管理器分层结构的建立及可安装文件系统的加载

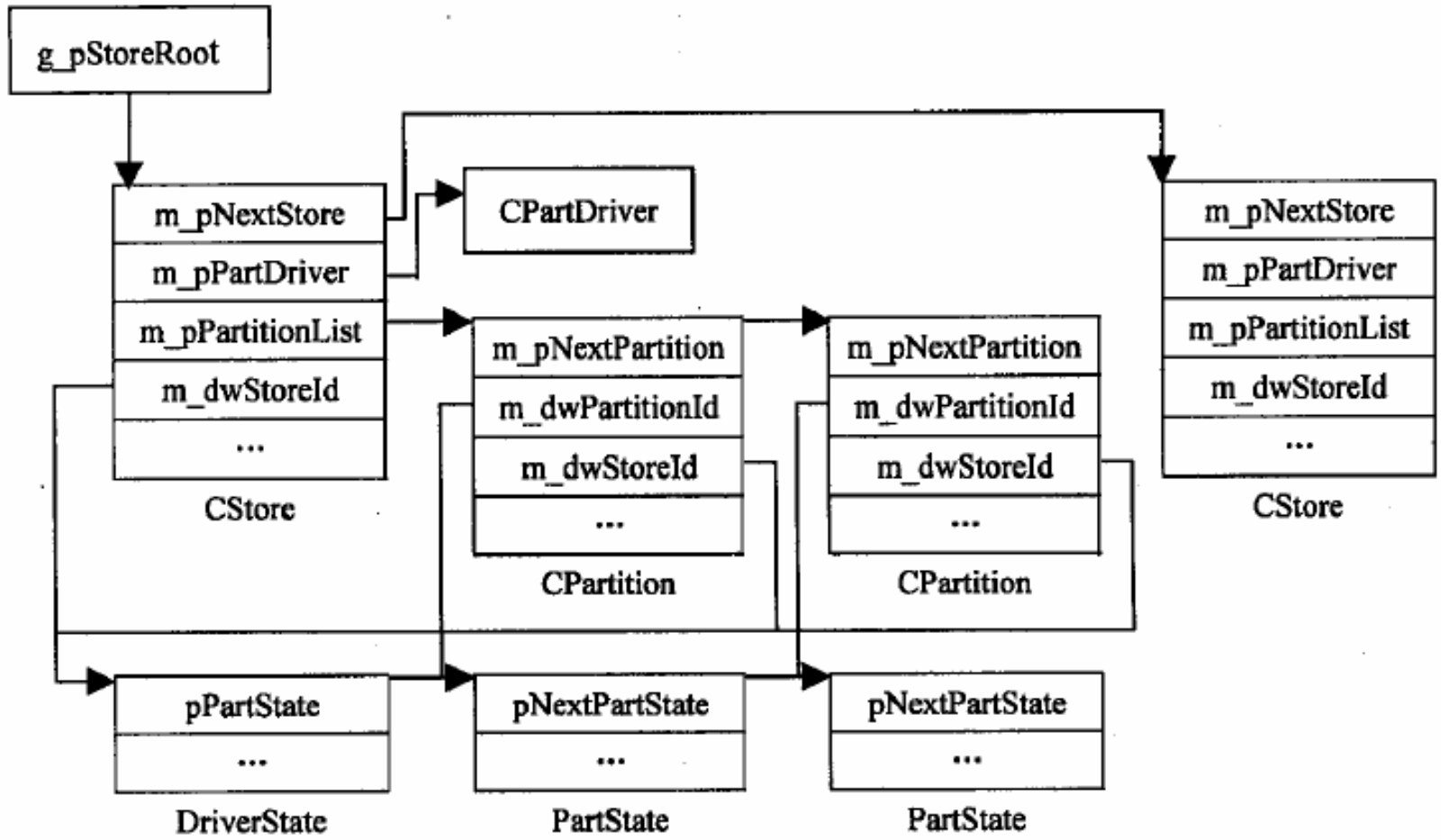


- 几个重要的数据结构
- 情景分析

几个重要的数据结构

- CStore类别
- DriverState由分区管理器负责建立并维护
- ＣＰａｒｔｉｔｉｏｎ类别
- ＰａｒｔＳｔａｔｅ是分区管理器为维护分区信息所使用的数据结构

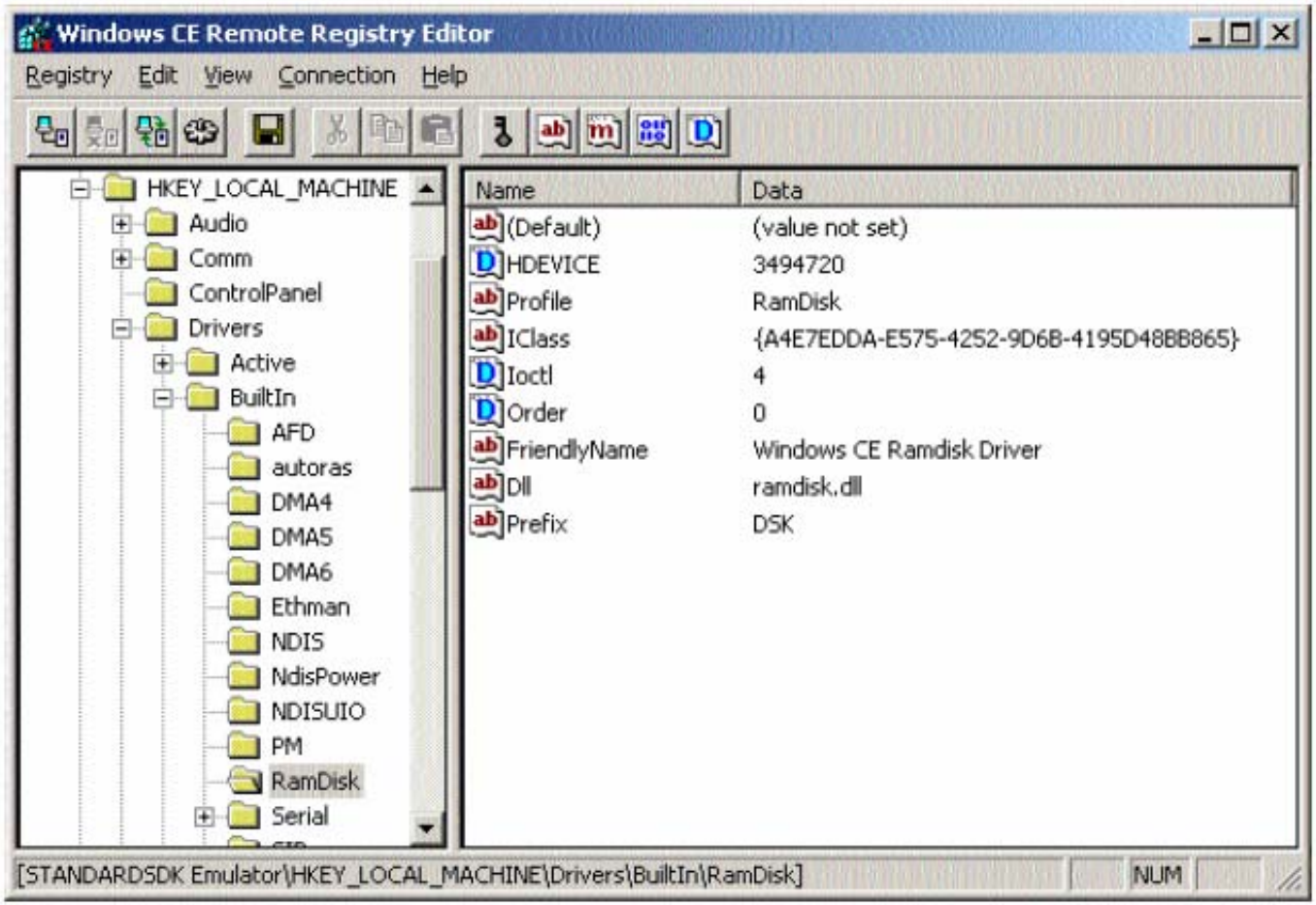
这些数据结构的关系


g_pStoreRoot
m_pNextStore
m_pPartDriver
m_pPartitionList
m_dwStoreId
...
CStore
CPartDriver
m_pNextPartition
m_dwPartitionId
m_dwStoreId
...
CPartition
m_pNextPartition
m_dwPartitionId
m_dwStoreId
...
CPartition
m_pNextStore
m_pPartDriver
m_pPartitionList
m_dwStoreId
...
CStore
pPartState
...
DriverState
pNextPartState
...
PartState
pNextPartState
...
PartState

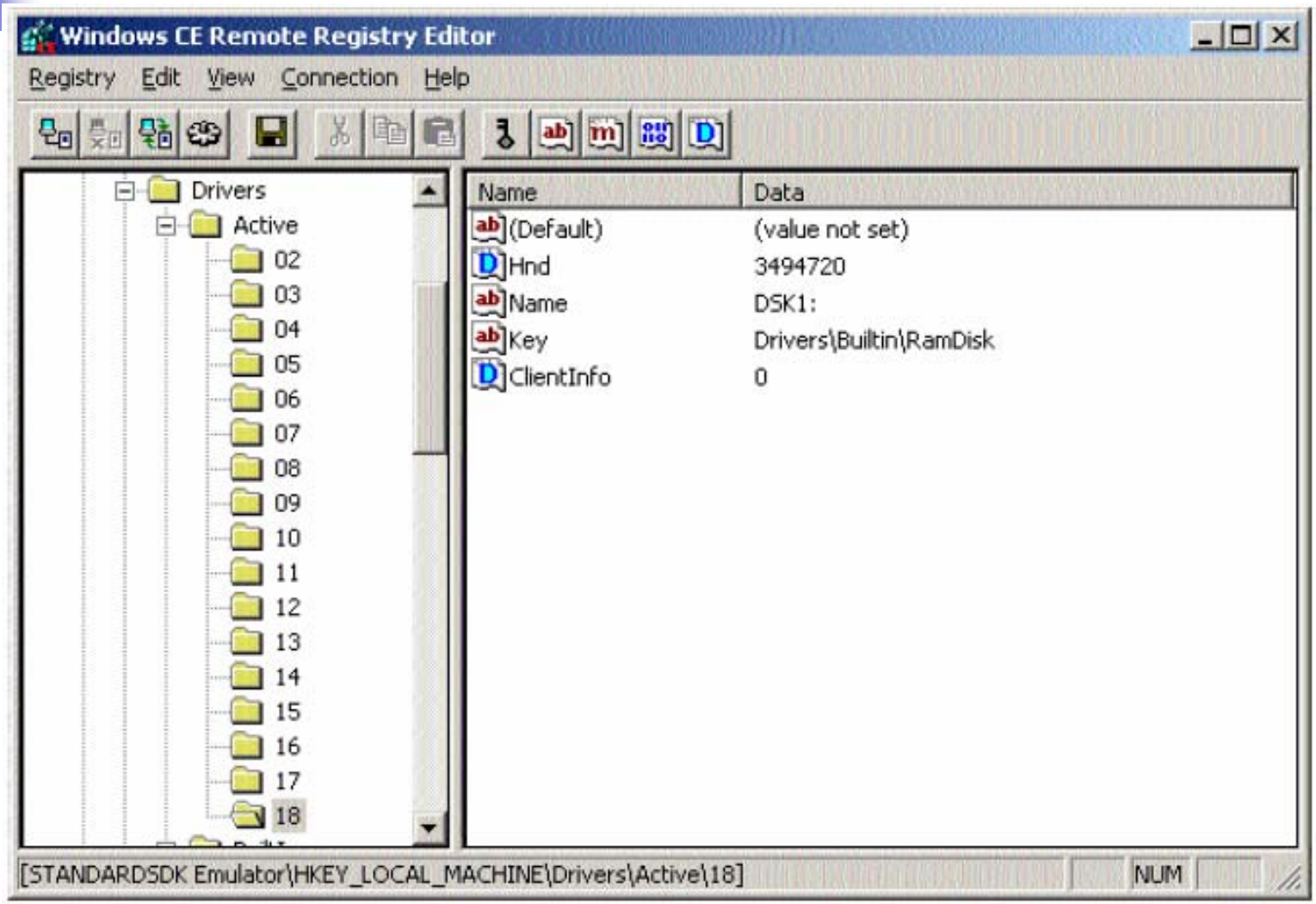
情景分析

- 在系统的启动程序中会自动加载两个模块，
  - 一个是filesys.exe，
  - 另一个是fsdmgr.dll
- 成功为其装入区块装置驱动程序之后，设备管理器会将Ramdisk作为已经启动（Active）的设备填写进注册表，并通过WriteMsgQueue将Ramdisk的设备名及其GUID等信息写入讯息队列。

注册表中有关Ramdisk的信息



Ramdisk启动之后的注册表

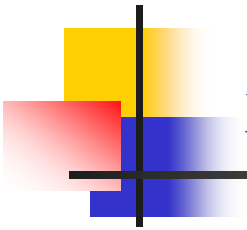


最后填写FSDINTDATA数据结构，并呼叫InitEx

- 进入InitEx之后首先会呼叫FSDLoad，由其完成两项任务：
  - 呼叫AllocFSD完成FSD结构的分配及初始化，并将其纳入文件系统驱动程序管理器的管理当中
  - 呼叫AllocDisk为该FSD分配并初始化一个DSK结构

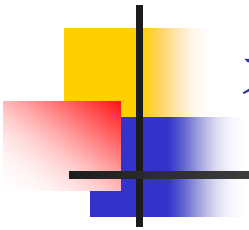
FAT文件系统中CreateFile的具体流程

- 实验环境
- 参数介绍
- 局部变数
- 过程跟踪



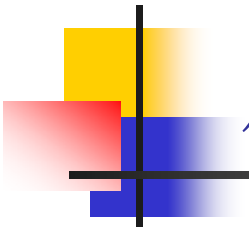
实验环境

- Platform Builder 4.0
- 目标机由Platform Builder自带的仿真器代替



参数介绍

- PVOLUME pvol
- 扇区（Sector）和簇（Cluster）概念比较

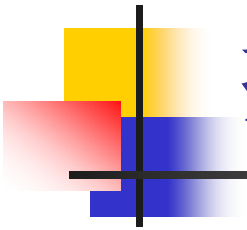


局部变数

- BYTE mode 文件打开的模式
- int flName 文件属性
- HANDLE hFile 文件识别码
- PFHANDLE pfh = NULL FHANDLE类型的指标
- PDSTREAM pstm = NULL DSTREAM类型指标
- DWORD dwError = ERROR_SUCCESS 出错信息
- BOOL bExists = FALSE 文件是否存在的标识

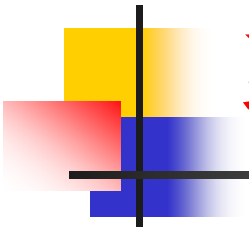
过程跟踪

- 1) 呼叫FATEnter。
- 2) 呼叫BufEnter。
- 3) 一系列的参数合法性判断和一些局部变量的初始化和赋值。
- 4) 呼叫OpenName。
- 根据回传的PDSTREAM pstm对流的共享性检查CheckStreamSharing。



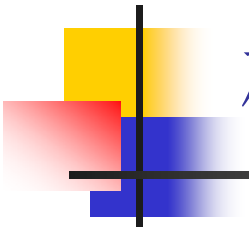
提纲

- WinCE 嵌入式系统原理
- 动态链接库的加载分析
- 线程在队列之间转换分析
- 虚存分配
- 文件系统的建立及访问过程
- 驱动程序加载



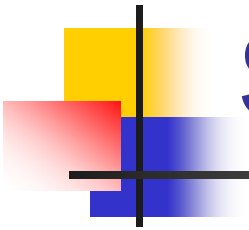
驱动程序加载

- 概述
- StartOneDriver
- 主要来源程序清单以及情景注释



概述

- 过程的基本步骤：
  - 1) 从注册表中读取特定的值，准备并校验参数。
  - 2) 设置Active键下的若干内容。
  - 3) 根据是否有前缀来分别处理。
  - 4) 呼叫装置管理器根据前面准备的参数注册驱动程序（呼叫RegisterDeviceEx函式）。
  - 5) 设置Active下的相关键的值。
  - 6) 对实作档案数据流接口的驱动做额外的初始化工作。
  - 7) 将系统接口变化通知到相关的行程。



StartOneDriver

- 装置信息的获取和维护
- 存取核心装置管理数据表
- 系统事件的传递

主要来源程序清单以及情景注释

- 原始码来自[CEROOT]\Private\Winceos\Coreos\Device\Lib\Devl oad.c