摘　要

本文面向嵌入式实时应用需求，在深入研究嵌入式实时系统和内存管理技术，国内外的研究现状和研究方向，系统的内存模式和内存管理的特点的基础上，提出了本文的研究目标和研究内容，提出并实现了一系列有助于提高系统安全性和可靠性的内存管理技术解决方法，包括：动态内存分配,内存保护等，在静态配置UB的基础上引入了“交换块（S Block）”的概念，并对其实现技术展开较为深入的研究。

文中提出要解决的问题并给出涉及到的技术理论，进行系统的需求分析给出了系统中内存数据的组织方式，描述了系统实现，并进行了测试分析。为保证嵌入式实时系统数据存储的安全性和可靠性，对内存管理技术的研究具有重要意义。

关键词：　嵌入式系统　内存管理　UB

Abstract

With the requirement of real-time embedded application, a systematic survey of research in the domain of embedded real-time and memory management is given, including the concept of real-time embedded system, the research status and tendency f real-time embedded system, the characteristic of memory model and memory management in reai-time embedded system. Then the study objectives and cotents of the thesis are presented.A series of memory mamagement Technology to improve safety and reliability for embedded system are proposed and implemented in the thesis, including memory redundancy allocation,memory redundancy encoding and memry protection . Then the key technology of these is researched and implemented.

Here puts forward the problem, carries through requirement analysis and buildup data in the main memory, describes the design of the architecture, records the testing and debugging data and proving the running environment.In order to improve the safety and reliability of data storage in real-time embedded system, research on memory management for embedded system is particularly signficant.

Keyword: **embedded real-time** **memory management** **UB**

创新性声明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果；也不包含为获得西安电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在文中作了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名：张金金　　日期：2008. 1. 15

关于论文使用授权的说明

本人完全了解西安电子科技大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属西安电子科技大学。本人保证毕业离校后，发表论文或使用论文工作成果时署名单位仍然为西安电子科技大学。学校有权保留送交论文的复印件，允许查阅和借阅论文；学校可以公布论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存论文。（保密的论文在解密后遵守此规定）

本学位论文属于保密，在____年解密后适用本授权书。

本人签名：张金金　　日期：2008. 1. 15

导师签名：　　日期：

第一章 绪论

本章首先简要介绍了内存管理模块在3G统一平台支撑层的位置，传统内存管理方法的的应用、现状及前景，由此引出在内存管理中存在的问题，指出了内存管理的重要性。最后给出了本文的主要工作以及内容安排。

内存管理模块在OSS系统中的位置如图1.1所示：

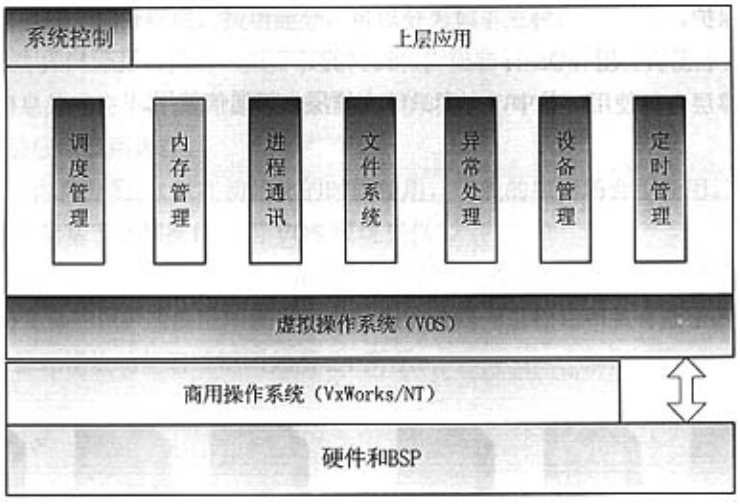


图1.1 内存管理在OSS中的位置

内存管理完成两个主要功能。一是内存分配管理，二是内存保护。对于长期运行、实时性要求高的系统，必须自己对内存进行管理。

VxWorks自身提供的内存管理函数采用first-fit算法。该算法在搜索可用的内存块时，从空闲队列头开始进行搜索，当找到第一个空间足够的空闲内存块时，即将该空闲块分配给其使用，剩余部分重新加入空闲队列。这种算法不适用于我们的系统。采用first-fit算法，一是内存分配的实时性得不到保证，具体的搜索时间和已分配的内存块个数成O(N)的关系，二是会产生内存碎片。

现存的内存管理模块具有设计简单、实时性强、效率高等优点，但使用静态配置UB的方法很难避免配置上不合理所带来问题（UB缺失或浪费）。内存管理的辅助功能(向上申请及系统对内存扩展)在一定程度上缓解了UB配置问题，但又会带其它一些问题。比如向上申请会引起实时性变弱；系统堆内存扩展将VxWorks本身内存管理的碎片问题暴露出来。

为解决上述问题，在保证现有UB管理优点的基础上，并考虑尽量合理有效的利用内存资源，将“交换块（S Block）”的概念引入到UB管理中来。引入“交换块”的概念后，实际上是在目前UB管理的基础上增加了一个层次，向内存池申请

UB 时，先要获取一个可用的交换块，进而获取已可用 UB。交换块可以被多个内存池交替使用，根据 UB 尺寸的大小，一个交换块被分解成若干 UB。交换块 UB 动态管理机制可以解决 UB 池间内存不能共享等问题。

对于 UB 保护，考虑到原来设计方案在打开 UB 保护功能时会浪费很多内存，甚至导致系统由于内存不够而不能正常启动，在设计时考虑支持动态打开 UB 保护功能，并支持对个别内存池进行保护，当然在内存资源够用时可以对全部 UB 池进行保护。

本模块引用 VOS 标准模块，提供的三套接口作为标准模块为上层应用模块和支撑层内部使用，其中，一套只给支撑层内部通信使用。

第二章 内存管理模块

2.1 概 述

内存管理模块主要由内存分配和内存保护两个相互独立的子模块组成。内存分配子模块由三个部分组成。按用途分，可以分为如下三种：

1.支撑内部使用，比如，进程间通信（消息构造时由通信模块为消息分配内存空间，消息处理完毕，由调度模块释放占用的内存空间）。

2.上层应用使用内存。

3.全局内存分配。（只在初始化的时候使用，一般的应用场合不使用。）

内存分配基于队列操作（由 VOS 模块提供）。

内存保护包括三个部分，一个是 UB 的保护，一个是进程栈和私有数据区的保护，全局变量的保护需要应用配合完成，保护流程同进程栈和私有数据区相同。

内存管理模块功能组成框图如图 2.1 所示：

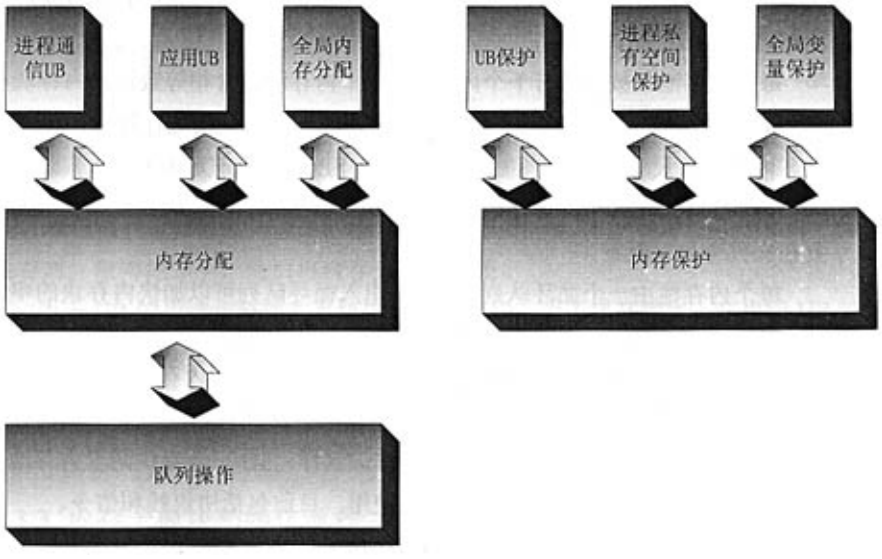


图 2.1 内存管理框架

2.2 核心数据区初始化

支撑内部的核心数据统一排布，所谓的核心数据，是指支撑内部使用的全局变

量。所有的核心数据都放在一个大的数据结构里面，先定义 T_CoreData *pTCoreData。对于大块数据，在 T_CoreData 内部加一结构指针，然后在内存初始化的时候动态分配。这样，通过 pTCoreData，就可以定位所有的核心数据。

2.3 内存分配子模块

2.3.1 概述

内存分配子模块主要实现对堆的动态申请和释放。VxWorks 也提供堆的动态管理，但其不适用于我们的系统。内存分配子模块只给上层应用提供两类接口，一个用于申请内存，一个用于释放内存。

内存分配子模块的设计目标是保证动态申请内存和释放内存的实时性，将内存碎片限制在可控的范围之内。内存分配子模块基于内存池集机制，对不同的应用场合，使用不同的内存池集。3G 统一平台使用两套完全独立的内存池集，即内部内存池集和应用内存池集。内部内存池集为支撑内部提供内存申请支持，这是支撑内部自己使用的内存池集，不对外开放。应用内存池集为上层应用提供高效的动态内存申请机制。

每个内存池集被分成若干个内存池，每个内存池包含相等数目的内存块和内存头，同一内存池中的内存块尺寸相同。内存池由循环队列进行管理，以达到实时性。内存块的尺寸的最小单位为 MEM_UB_SPAN（目前定为 64），其余内存块的大小为 MEM_UB_SPAN 日整数倍，其大小没有上限。内存块个数可以根据应用需要在文件中自行配置。

每个内存池由一个循环队列来管理。引入循环队列可以加快内存块的申请和释放。除去信号量等待的情况，每次申请内存块的时间是一样的。

上层应用如果需要使用到内存的动态申请，仍然采用内存池集管理机制。内部内存池集和应用内存池集完全独立，互不干扰，这样做的优点是一套内存池集的耗竭不会影响另一套内存池集的正常工作。这样，3G 平台提供两套内存池集，一套只给支撑内部使用，一套给上层应用使用，目前包括协议栈和信令。

另外，支撑层还提供对 malloc 函数的封装。作一层封装的目的只是为了加入统计功能。通过自封装的函数，支撑层就可以完全掌握应用对内存的使用情况。此函数只在支撑内部和初始化的时候预分配内存使用。

下图是队列、内存头和内存体之间的关系图。

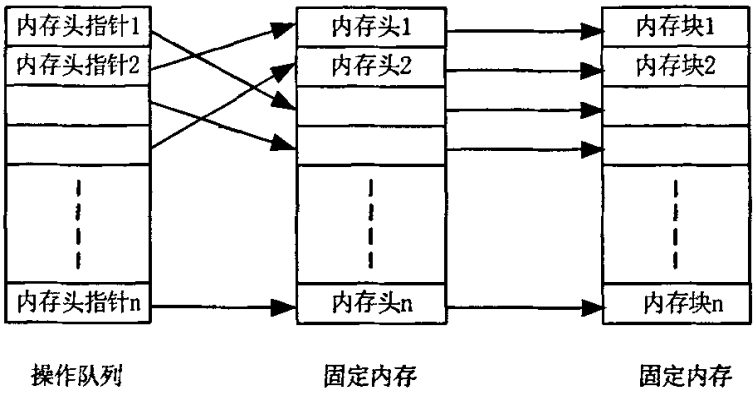


图 2.1 队列、内存头和内存块之间的关系

从图 2.2 中可以看出，内存头和内存块的排列关系是有规律可循的，这种对应关系在运行过程中不会改变。第一个内存头永远被用来控制第一个内存块，而不可能去控制别的内存块。但是内存头指针在队列中的位置是会不断改变的。从队列中取出的一个内存头指针，可以指向任何一个内存头。

2.3.2 队列操作

对每个内存池，初始化的时候先创建一块连续内存，用以存放内存头指针。并用头、尾两个索引来分别代表当前可用第一个内存头和当前可用的最后一个内存头。取可用内存时，从内存池控制块中取得头索引，而后取得内存头指针。释放内存时，取得尾索引，而后将内存头指针归还。

由于 UB 的申请和释放在系统中会频繁使用，为了提高效率，在申请 UB 的时候，将申请者申请的内存大小增加 8 字节，增加的部分用以存放释放信息，包括队列头和内存头指针。当释放内存时，可以得到指针值，将指针前移 8 个字节，就可以得到内存头指针和队列头。从而避免了释放信息的搜索过程。

2.3.3 内存子模块初始化流程

该初始化流程负责整个内存子模块的初始化工作，并向其它子模块提供一个函数接口 Mem_SysInit。

传入参数：无。

内存初始化流程如下：

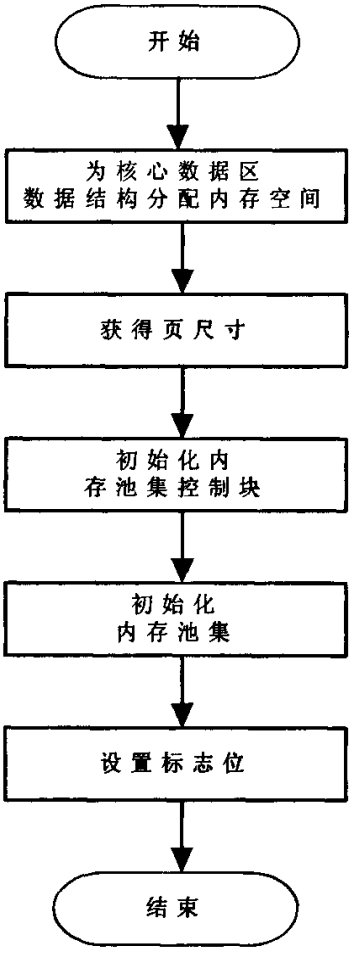


图 2.2 内存子模块初始化流程

流程描述：

1、为核心数据区数据结构分配内存空间，并清零。

2、获得页尺寸大小。

3、初始化内存池集控制块，标记所有的内存池集控制块没有被使用。

4、初始化内部内存池集。

5、初始化用户内存池集。

6、标记初始化已经完成。

其中，第 4、5 步的流程完全相同。

2.3.4 初始化内存池集流程

传入参数：

T_UbInfo *ptUbInfo——指向内存池集的配置信息。

WORD32 dwCfgTblLen——配置表的长度，其值由如下两个宏决定。

内部内存池集由宏 MEM_INNERUB_POOL_NUM 决定，用户内存池集的长度由宏 MEM_UB_POOL_NUM 决定。

概况来讲，初始化流程主要完成下面三部分的工作：

- 根据内存配置表进行内存预分配，内存配置表有两套。
- 创建控制队列，引用 VOS 标准模块的队列操作。
- 对内存头和内存体加页保护。

下面描述对内部内存池集的初始化过程，应用内存池集的初始化过程与之相同。

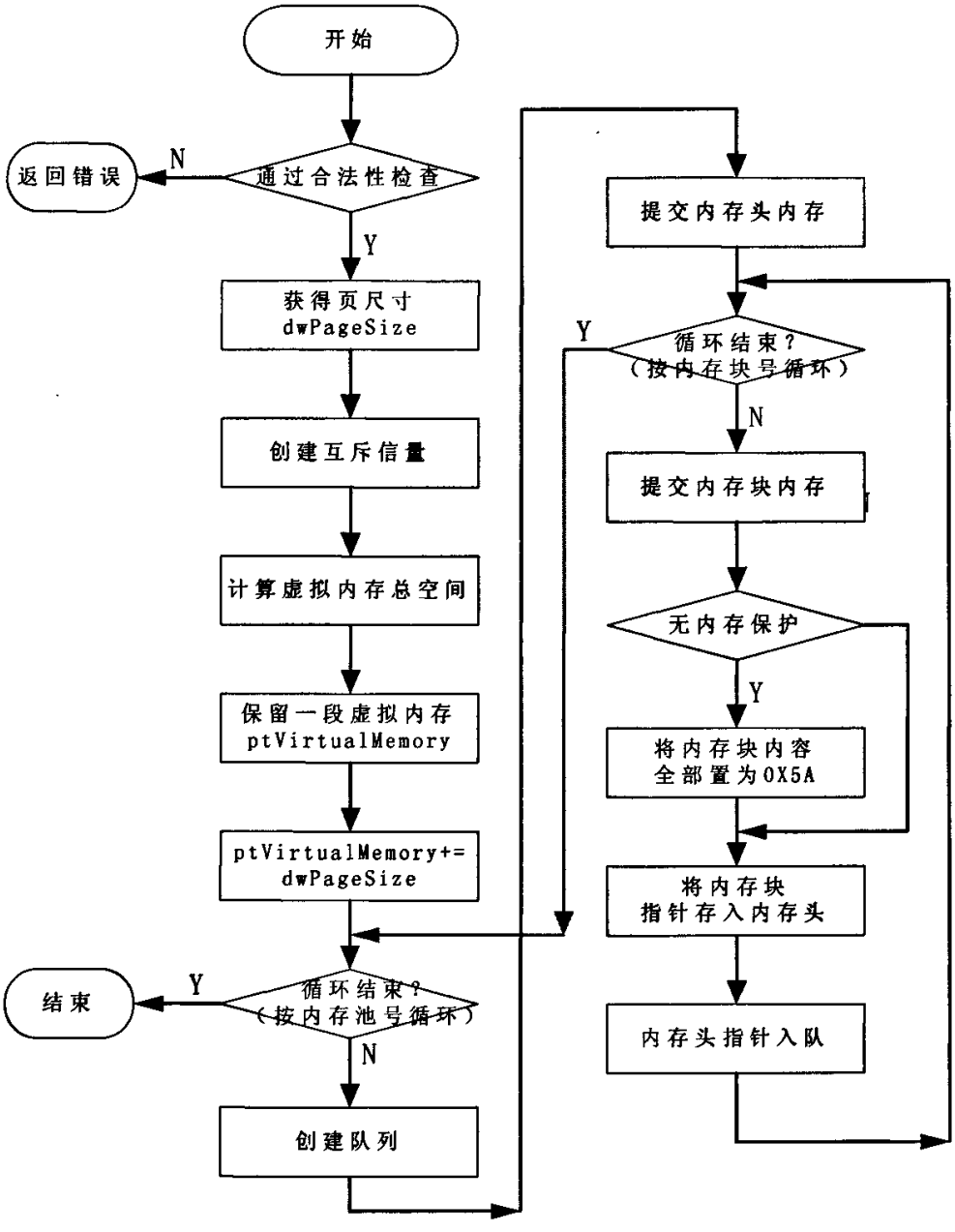


图 2.4 内存分配子模块初始化

流程描述：

1、进行合法性检查(按内存池号循环)，主要是检查用户的数据是否配置正确。包括，内存块大小是否为 MEM_UB_SPAN 的倍数，内存池中内存块大小是否按序增长。

2、获得页尺寸 dwPageSize，如果页保护开关打开，则返回的是 4096 个字节，否则，返回 64 个字节。

3、创建互斥信号量，一个内存池集有一个互斥信号量。

4、计算虚拟内存总空间，总空间的计算根据 UB 头尺寸和 UB 块大小来确定。同一个内存队列的 UB 头放在一起，间隔 MEM_PROTECT_PAGE_COUNT 个保护页。在内存头之后，排放内存块。内存块之间同样插入 MEM_PROTECT_PAGE_COUNT 个保护页。

5、调用 VOS 提供的虚拟页申请函数，保留一段虚拟内存（已经包括保护页），这段内存初始状态为不可访问，由 ptVirtualMemory（临时变量）指定。

6、ptVirtualMemory 加一个页，为第一个内存池的内存头前面加虚拟保护页。

7、从 ptVirtualMemory 处提交内存头内存，这段内存可以读写。

8、调用 VOS 提供的队列创建接口，创建一个无信号量、不可扩展的队列，将队列句柄存入对应的队列控制块。

9、提交内存块（按内存块号循环）。保护的格局见内存池集的保护（图）。

10、如果内存保护开关没打开，则将该内存块全部置为 0x5A，以便以后检测使用。

11、将内存块指针存入内存头。

12、将内存头指针入队。

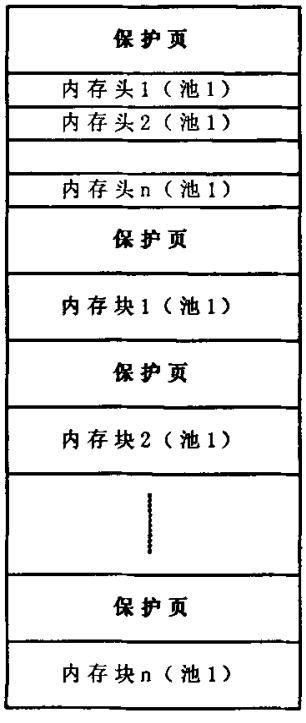


图 2.5 内存池集的保护

图 2.5 只画了一个内存池的保护格局，其它内存池的保护格局与此完全相同。

2.3.5 申请内存流程

传入参数：

T_PoolGroupCtrl *pGrpCtl——指定对哪个内存池集进行操作。

WORD32 dwSize——申请 UB 大小。

WORD32 dwIfUseReserve——是否使用保留内存。

USE_RESEVER_MEM——使用

NO_USE_RESEVER_MEM——不使用

WORD32 dwTimeOut——超时值（目前定为 500ms）。

下图是内存申请的分配流程：

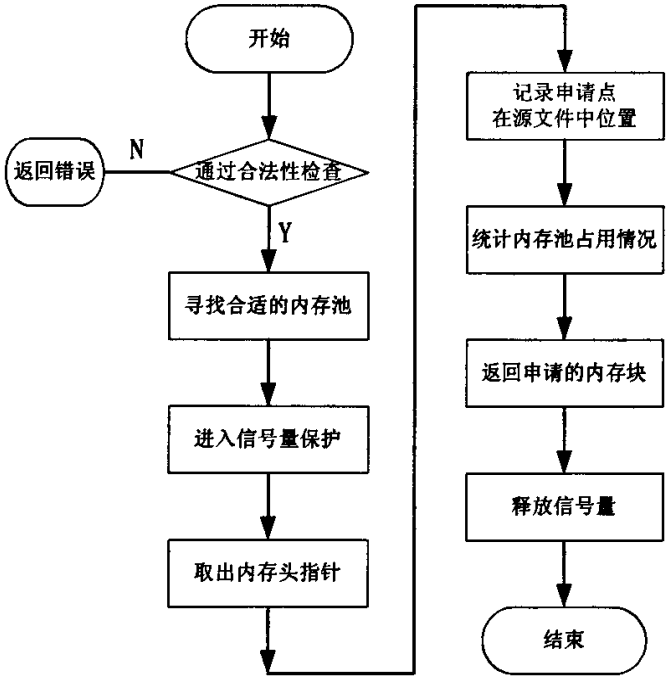


图 2.6 内存申请流程

流程描述：

1、 参数的合法性检查，尺寸必须介于 0 和最大尺寸之间。

2、 根据尺寸从内存池集中寻找合适的内存池。

3、 进入信号量保护。

4、 如果该内存池没有可用内存，则向更大的内存池申请可用内存。

5、 从队列头中取得内存块头指针。

6、 记录申请点在源文件中的位置。存申请函数在源文件的位置，包括文件名和行号。

7、 统计内存池占用情况，包括内存池中当前占用的内存块的个数，内存池中最多占用的内存块的个数。

8、 返回申请的内存块。一般来说，分配给应用的内存比实际需要的要多。返回指针的方式有两种，一种是将低端地址返回，另外一种是将高端地址减去实际所需要的内存大小，而后返回。由于内存越界最多的情况是向后越界引起，因此，采用第二种方法，只要越界，就可以触发异常。

9、 释放信号量。

2.3.6 释放内存流程

传入的参数：

T_PoolGroupCtrl *pGrpCtl——指定对哪个内存池集进行操作。

BYTE *pucBuf——指向待释放内存块。

图 2.7 是内存释放的流程：

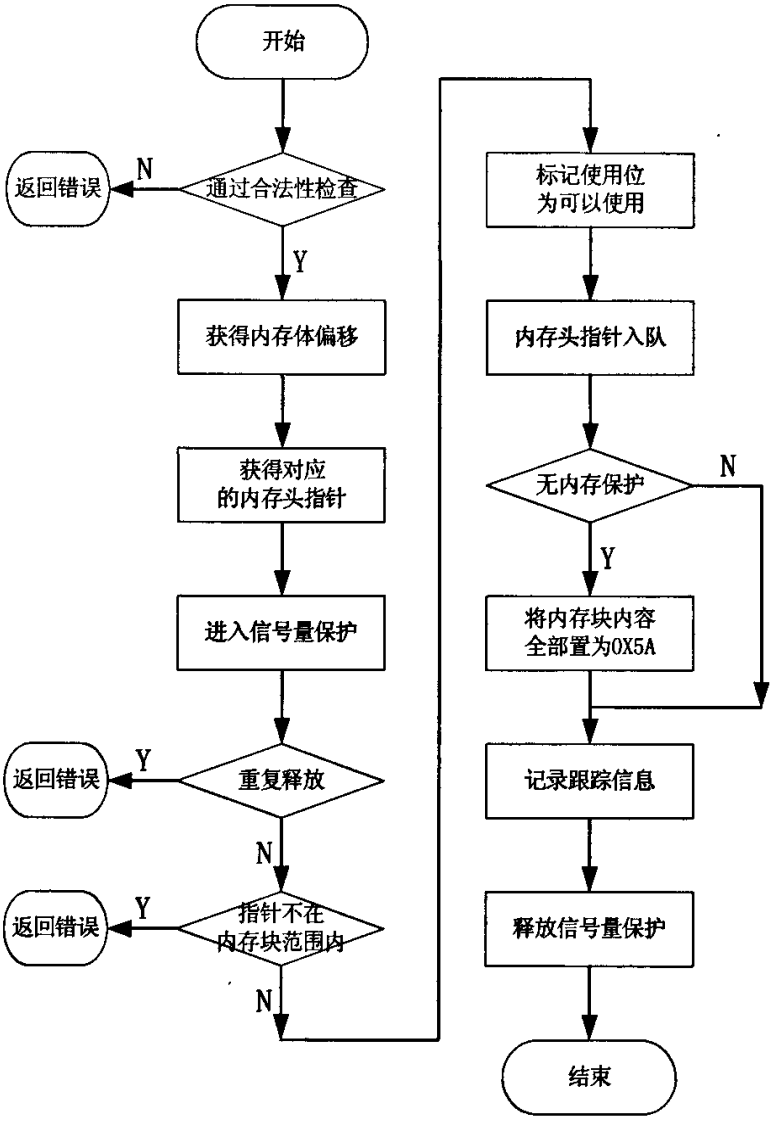


图 2.7 内存释放流程

1、合法性检查，指针必须在内存池集的范围之内。

2、获得内存体偏移，根据 pucBuf 获得其离内存体头部的偏移位置。（由 pucBuf 同各内存池控制头中记录的内存范围进行比较，如果落入该内存池范围，则认为找到对应的内存池。）

3、获得对应的内存头指针，根据偏移位置得到内存头指针。

4、进入信号量保护。

5、根据使用标识，判别是否重复释放。

6、根据内存头，判别返回指针是否落在内存块限定的范围内。

7、置使用标识位为可以使用，然后将内存头指针加入队列。

8、如果没有内存保护，则将内存块中相应的内容重置为 0X5A。

9、记录跟踪信息。

10、 释放信号量保护。

2.3.7 统计功能

统计功能包括如下部分：

1、 支撑占用的内存总量，包括两套内存池集，所有使用 OSS_Malloc 分配内存的情况。对于 VxWorks 内核自身使用的内存空间可以使用内核自身提供的 memShow 命令来显示。

2、 UB 的使用情况。包括当前 UB 池的使用情况，UB 池最大的使用个数。

2.3.8 可测试性设计

2.3.8.1 测试范围

测试重点包括下面几个方面：

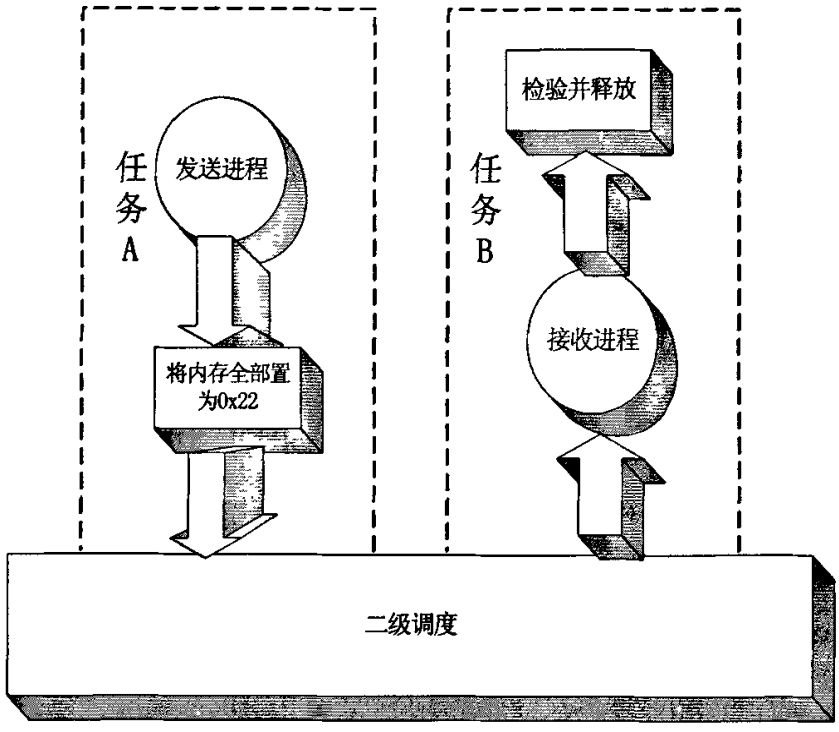
- 初始化
- 申请内存
- 释放内存

另外，由于 UB 块保护在内存分配子模块中实现，因此，测试设计也考虑了 UB 块的保护。

2.3.8.2 测试方法

创建两个进程，假设进程名分别为 Sender 和 Receiver，分属不同的调度任务，任务优先级不同（可以测试内存池集的信号量保护是否正确）。Sender 进程以时间间隔 dwSendInterval 申请大小为 dwMemSize 的内存，并且内存内容全部置为 0x22，而后将内存指针、内存大小发送给 Receiver 进程。接收进程收到之后，同步延时 dwRecInterval 秒。而后根据收到的消息，对内存内容进行校验，如果内容不全为 0x22，则证明有问题，此时，应该停止发消息（方法是将 dwSendInterval 和 dwRecInterval 置为 0xffffffff）。如果全为 0x22，则将内存释放。dwSendInterval、dwMemSize 和 dwRecInterval 作为全局变量，可以在 shell 中直接修改。

内存分配模块的测试模型如下所求：



内存分配测试模型

对于 UB 保护的测试，只要在置内存值为 0x22 的时候，将写操作的长度超出申请的内存大小，从而触发异常处理流程。

2.4 内存保护子模块

内存保护包括两个部分，UB 的保护和页表切换保护。

内存保护子模块的主要作用是在于问题定位。内存保护引入虚拟内存机制，通过页表属性的设置来决定内存的访问权限。引入内存保护的会引起程序运行效率下降，特别是在页表频繁切换的时候，因此，内存保护实现的完备程度受运行效率的限制。

2.4.1 概述

对所有静态配置的 UB，按插入虚拟页的方式进行保护。UB 在内存中的排布如图内存池集保护格局所示。加入对 UB 的保护之后，对 PPC 体系，如果页数增加过多，理论上有可能会降低效率。对于 X86 体系，由于每次查找页表所需要的

步骤相同，因此，不可能降低效率。另外，因为页的最小尺寸目前定为 4096 字节，因此，加入 UB 保护，对于不是页整数位的内存块，会浪费内存。

在 3G 统一平台中，为了实现尽可能完全的保护，内存被划分成两个空间，一是系统空间，一是进程空间。这两个空间的唯一区别就在于对内存的访问权限上，而不在指令权限上。每个空间由一套页表来维护。OSS 初始化完成后，对于内存的访问只需要一套页表完成。

系统空间拥有对内存资源最大的控制权，除了对代码段不可写之外，对进程栈和私有数据区不可写之外，对所有的内存区都可读写。进程空间只可以对本进程的栈和私有数据区可读写，对其它进程的栈和私有数据区不可写。

进程运行前，将系统空间切换到进程空间，让系统在受 MMU 保护的模式下运行。

划分两块空间的目的是为了提高运行效率，但会增加内存，增加的内存被用来存放页表。如果 CPU 采用 PPC，当实际物理内存为 256M 时，页表占用的内存空间推荐为 2M。因此，以多出 2M 的内存来换取运行效率，代价不大。如果想追求更高的效率，可以考虑采用更多套页表。目前的设计只采用两套页表，并且在页表切换完成后删除原先的系统页表。

对于全局变量的保护，采用下列方法。

将全局变量组织成一个结构（由各个进程自己完成）。在进程上电的时候调用支撑层提供的全局变量注册函数。全局变量注册函数的入口参数是结构指针的指针和结构的长度。注册时，根据长度分配页尺寸整数倍的内存，并在 PCB 表中登记本进程有使用这一全局变量的权限。其它使用这些全局变量的进程也调用该注册函数，如果已经注册过，则不需要再次为其分配内存。

2.4.2 内存保护范围

对不同的 CPU，VxWorks 的内存排布图不尽相同，但大致的结构一样。目前 3G 统一平台采用的 CPU 主要有三种，一种是 PowerPC，一种是 Intel x86，还有一种是 Arm。

PowerPC 的内存排布如图 2.8 所示：

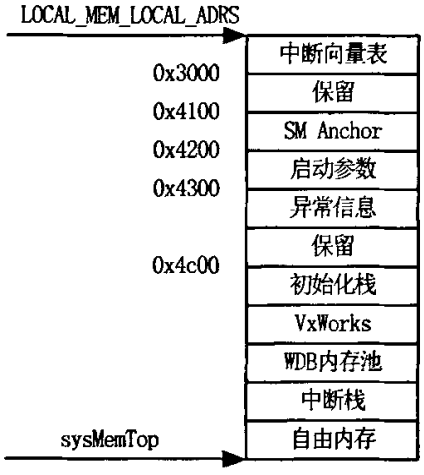


图 2.8 VxWorks 的内存排布图（PowerPC）

内存保护以页为单位，因此，所有需要保护的内存大小必须都是页的倍数。另外，分配的内存的起始地址必须按页对齐。统一平台不支持小于页尺寸大小的内存保护。

从图中可以看出，中断向量表从页边界开始，并且大小是页的倍数，可以进行保护。接下来的内存直到初始化栈结束，没有必要作保护，而且也不可能作保护。

VxWorks 开始的 text 段必须做保护，这样做的前提是 text 段必须从页边界开始，而且大小也必须是页的倍数。

WDB 内存池不作保护。

中断栈作保护。

自由内存区作可选择的保护。实际上，内存保护的主要工作是在自由内存部分。因为，进程的栈和私有数据区都在自由内存区中分配。保护的策略和方法主要采用页表切换的方法实现。

2.4.3 全局变量保护

全局变量保护需要应用来配合完成，主要目的是为应用提供一种定位问题的手段。全局变量注册函数只能在进程上电的时候调用。

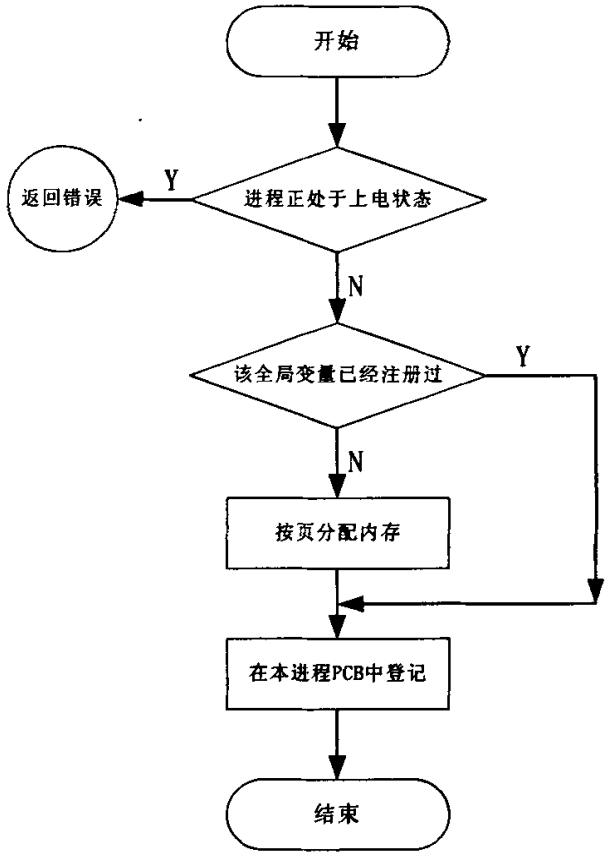


图 2.9 全局内存保护注册流程

为了支持全局内存保护，在 PCB 中加入两个字段：

```
BYTE *pucProtectGloablData; /*  指向全局保护内存 */
WORD32 dwProtectGloablDataLen; /* 保护区长度 */
```

2.4.4 可测试性设计

2.4.4.1 测试范围

保护的测试重点在于进程私有数据区被非法访问时，能不能触发异常。

2.4.4.2 测试方法

1．创建两个进程，分别为进程 A 和进程 B。打开内存保护开关，在进程 A 中

直接访问进程 B 中的私有数据区，则会触发异常。

2．递归调用同一函数，通过不断加大嵌套次数，在可预测的次数内，可以触发异常。

3．动态保护进程栈和私有数据区，在进程运行中只允许操作本进程的资源。出现非法访问栈或私有数据区时，触发异常。

4．在进程栈和私有数据区之间插入保护页，并将保护页属性设为只读，出现访问越界时触发异常。在设置内存访问属性后，注意回避一种情况，X86 和 ARM 上栈越界到只读内存，会死机。这与 vxWorks 在不同 CPU 的异常实现有关，而目前采用的 vxWorks 操作系统不支持这种异常的处理。

第三章 系统 UB 动态管理模块设计

3.1 模块描述

系统 UB 动态管理模块是做为原内存管理模块的补充设计而存在的，在 OSS 系统中的位置如图 3.1 所示：

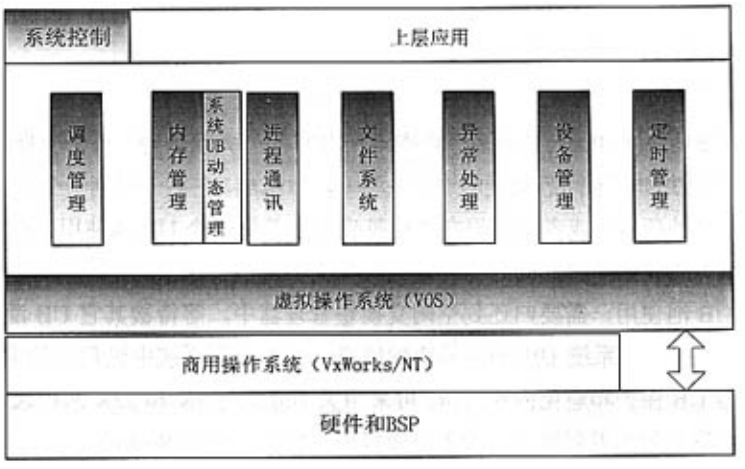


图 3.1 内存管理在 OSS 中的位置

现存的内存管理模块具有设计简单、实时性强、效率高等优点，但使用静态配置 UB 的方法很难避免配置上不合理所带来问题（UB 缺失或浪费）。内存管理的辅助功能(向上申请及系统对内存扩展)在一定程度上缓解了 UB 配置问题，但又会带其它一些问题。比如向上申请会引起实时性变弱；系统堆内存扩展将 VxWorks 本身内存管理的碎片问题暴露出来。

为解决上述问题，在保证现有 UB 管理优点的基础上，并考虑尽量合理有效的利用内存资源，将“交换块（S Block）”的概念引入到 UB 管理中来。引入“交换块”的概念后，实际上是在目前 UB 管理的基础上增加了一个层次，向内存池申请 UB 时，先要获取一个可用的交换块，进而获取已可用 UB。交换块可以被多个内存池交替使用，根据 UB 尺寸的大小，一个交换块被分解成若干 UB。交换块 UB 动态管理机制可以解决 UB 池间内存不能共享等问题。

对于 UB 保护，考虑到原来设计方案在打开 UB 保护功能时会浪费很多内存，甚至导致系统由于内存不够而不能正常启动，在设计时考虑支持动态打开 UB 保护功能，并支持对个别内存池进行保护，当然在内存资源够用时可以对全部 UB 池进行保护。

3.2 模块设计

系统 UB 动态管理模块包括 3 个子模块。空闲 S Block 管理子模块、系统 UB 使用管理子模块、系统 UB 保护子模块。

3.2.1 空闲 S Block 管理子模块

3.2.1.1 交换块的相关概念

交换块(S Block)是为多个内存池所共享的一大块内存，每个交换块由 1 个交换块控制块和若干同等数量的 UB 头和 UB 体组成。在某一时刻一个交换块或者为一个 UB 池所有，或者在空闲交换块管理器中等待一个 UB 池使用。在同一时刻，任意两个 UB 池不可能同时拥有对同一个交换块的使用权。一个交换块如果不再被一个 UB 池使用，需要归还到空闲交换块管理器中，等待被其它 UB 池使用。

根据目前系统 UB 使用的实际情况，并考虑到系统中页尺寸的大小为 4K，便于做 UB 保护和避免内存浪费，可采用大小分别为 4K 和 32K 两种尺寸的 S Block。各内存池根据其管理 UB 的大小选择使用何种尺寸的 S Block。

虽然当前 UB 尺寸、个数的配置不太合理，但也存在一定意义上的可参考性。可以根据目前系统 UB 的配置情况折算出所需相应大小 S Block 的初始个数，初始化时这些 S Block 不归任何 UB 池所有，而是统一放在空闲交换块管理器中。当空闲交换块中的初始配置的 SBlock 都被用完时，空闲块管理器负责向系统堆内存申请内存并生成新的可用 S Block。考虑到当前 UB 配置中提供了保留 UB 的功能，需要提供一定数目的保留 SBlock 供定时器模块申请 UB 时使用。按照上述所说，S Block 可以分为静态配置的，动态申请的和保留的三种类型。

S Block 类型：

```
SBLOCK_TYPE_DYNAMIC      /* 动态申请 */
SBLOCK_TYPE_CONFIG          /* 静态配置 */
SBLOCK_TYPE_RESERVE      /* 保留类型 */
```

3.2.1.2 空闲 S Block 管理

空闲 S Block 是由空闲 S Block 管理器管理的，空闲 S Block 管理器通过两个队列分别管理为大小 4K 和 32K 的 S Block。初始化时，根据当前系统 UB 的配置情

况折算出所需 SBlock 的尺寸及数量，并从系统堆内存中申请相应尺寸及数量的内存用做交换块（S Block），然后初始化这些 S Block 为静态配置类型并将其添加到相应的队列中。初始化过程如图 3.2 所示：

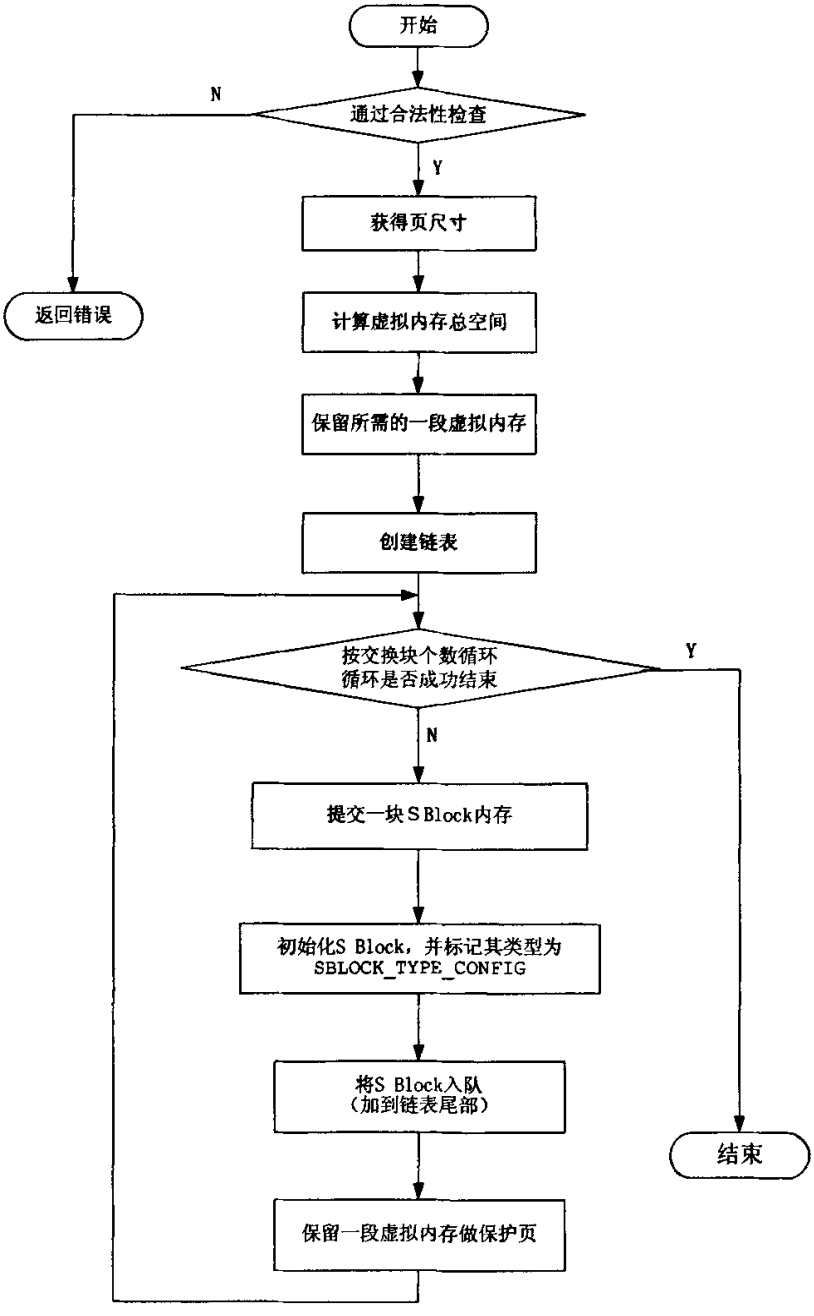


图 3.2 空闲 S Block 初始化过程

流程描述如下：

1. 进行合法性检查，主要检查空间交换块配置表数据是否正确。
2. 获得页尺寸大小。
3. 计算虚拟内存总空间，总空间的计算是根据交换块的尺寸和数目来确定的，每个交换块之间间隔一段保护页。
4. 调用 VOS 提供的虚拟内存申请函数，保留一段虚拟内存（包括交换块及交换块间的保护页），这段内存初始状态为不可访问。
5. 创建并初始化一个链表用来维护空闲交换块。
6. 按交换块的个数进行循环。
7. 提交一个交换块内存，并使这段内存可读写。
8. 初始化空闲交换块，主要初始化块中用来维护 UB 使用的链表结构以及用来指向当前可用的 UB 头的指针，同时标记交换块的类型为 SBLOCK_TYPE_CONFIG。
9. 将空闲交换块加入队列中。
10. 在交换块的后面保留一段虚拟内存作为保护页，对于 4K 和 32K 的交换块保护页的大小分别为 4K 和 32K。

当 UB 池向空闲 S Block 管理器申请一个可用的 S Block 时，空闲交换块管理器从相应队列获取 S Block 给申请者使用；当 UB 池向空闲 S Block 管理其归还一个 S Block 时，交换块管理器将该 S Block 添加到相应队列中。

当 UB 池向空闲 S Block 管理器申请 S Block 时，如果对应的队列已经为空，空闲 S Block 管理器向系统堆内存申请一定尺寸的内存生成新的 S Block，并做相关的初始化同时标记次 S Block 为动态申请类型。

对于动态申请类型的 S Block，当 UB 池将其向空闲 S Block 管理归还时，需要根据事先选择的策略决定是否向系统堆内存归还，这里所讲的策略可能有以下两种，可以通过变量控制究竟使用何种策略：

1. 对于 4K 的 S Block，静态配置不够时向堆内存申请，S Block 归还时不释放内存而是加入空闲 S Block 队列；对于 32K 的 S Block，静态配置不够时不向堆内存申请扩展 S Block。
2. 对于 4K 的 S Block，策略同上；对于 32K 的 S Block，静态配置不够时向堆内存申请扩展 S Block，当此 S Block 归还时释放其所在的内存。
3. 对于 4K 的 S Block 和 32K 的 S Block 一样处理，即空闲块用完时向堆内存申请扩展 S Block，归还时不释放。

图 3.3,3.4 分别描述了向 S Block 管理器申请和归还的流程：

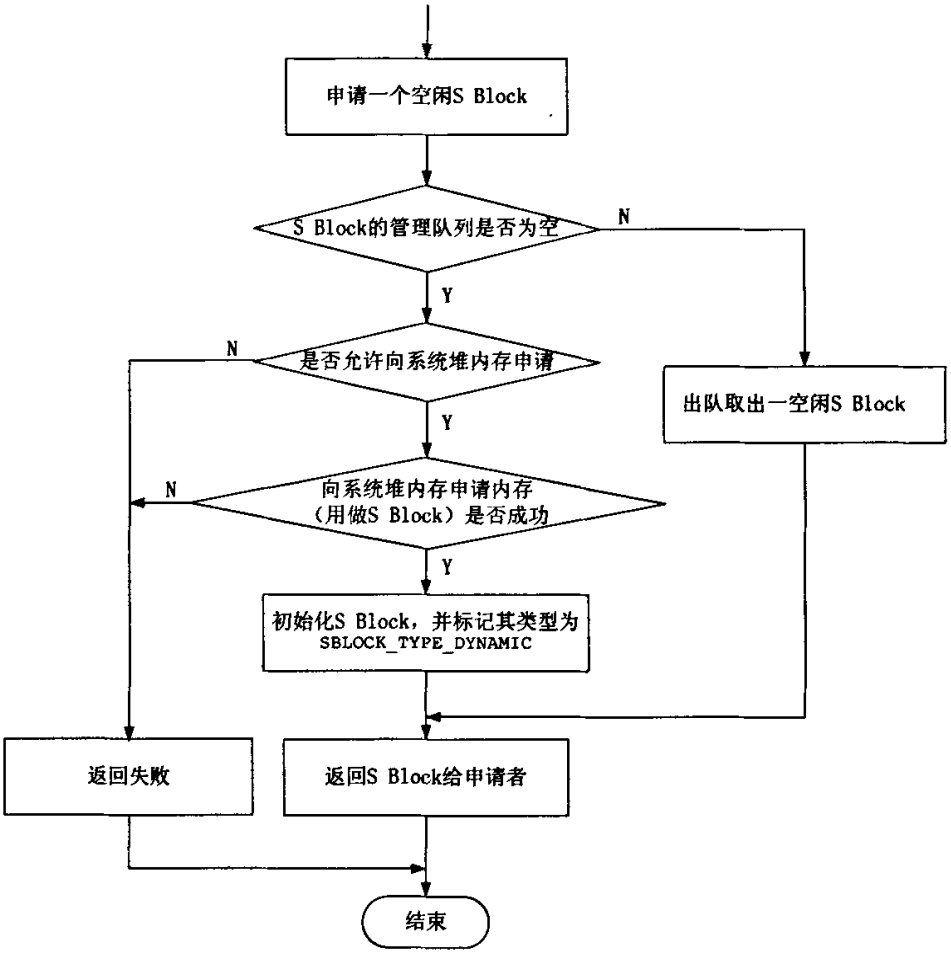


图 3.3 申请空闲 S Block 流程

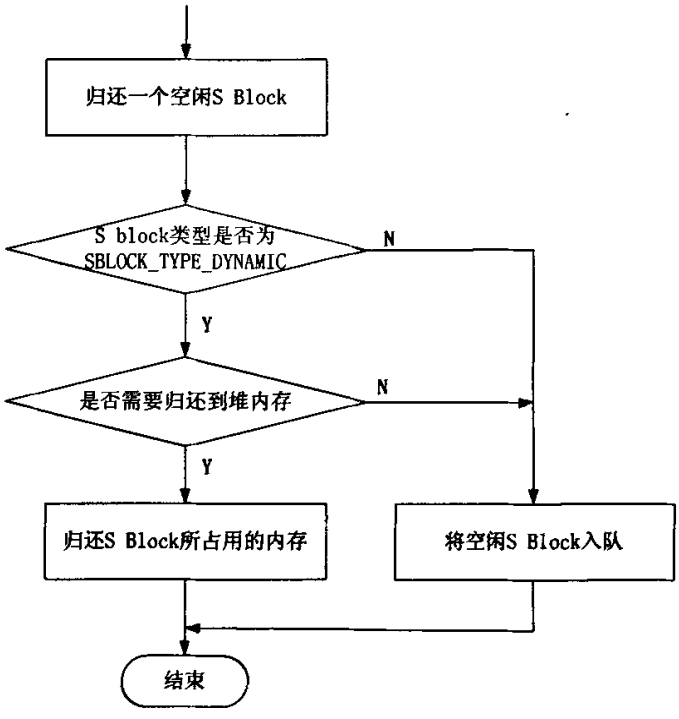


图 3.4 归还空闲 S Block 流程

3.2.2 系统 UB 使用管理子模块

系统 UB 使用管理子模块主要基于交换块动态管理技术，描述了动态管理的层次结构、模块初始化流程、UB 申请/释放流程以及 UB 使用管理的辅助功能 UB 使用告警和 UB 使用统计。

3.2.2.1 交换块动态管理的层次结构

(1) UB 池的分层结构

基于交换块的 UB 动态管理方法从大的层次结构上来看，可分为 UB 池层、交换块（S Block）层和 UB 头/体层三个层次，层次结构示意如图 3-5 所示。对每一个层的具体描述如下：

1. UB 池

相同尺寸的 UB 属于同一个 UB 池，每个 UB 池有一个 UB 池控制块和若干交换块（S Block）组成。UB 池控制块用于管理 UB 池中的交换块，其中交换块的个数不固定可以根据需要增加或减少，即 UB 池中的 UB 的总数是动态

变化的。

为了合理利用交换块，避免浪费过多的内存空间，按照管理的UB尺寸大小将UB池分成10个，每个UB池管理的UB尺寸分别为64、128、256、512、960、1984、3968、8128、16320、32640字节。

2. 交换块（S Block）

向前面所描述的，将交换块分成4K和32K两种固定尺寸，每个交换块由一个交换块控制块和若干同等数量的UB头和UB体组成，交换块控制块用于管理交换块内部的UB头和UB体。由于交换块的尺寸固定，当根据不同尺寸的UB划分交换块时，所得到的UB个数不同。各UB池根据其管理UB尺寸大小的不同选择使用4K还是32K的交换块。

3. UB头（UB head）

UB头与UB体一一对应，作为UB体的控制块，保存UB体的使用信息。

4. UB体（UB）

UB体为提供给申请者实际使用的连续内存单元。

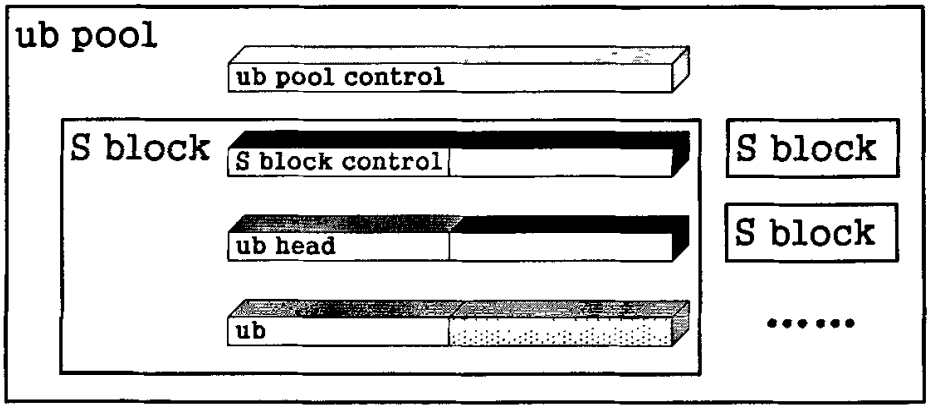


图3.5 交换块动态管理的分层结构

(2) UB池控制块结构

前面已经说过UB池控制块是用来管理交换块的，UB池控制块通过其维护的数据结构知道哪些交换块是可用的（交换块内有可使用的UB），哪些交换块是不可用的（交换块内无可用UB），哪些交换块是保留交换块。如图3.6所示，UB池控制块维护了四个链表：可用交换块链表、用满交换块链表、保留交换块链表和未保护交换块链表。下面对UB池控制块数据结构各项做一下说明：

1. 可用交换块链表：用来管理UB池中可用S Block的双向链表，当链表中的某一个S Block的UB都用尽时，该S Block将被转移到用满交换块链表中；当链表中的某个S Block的所有UB都不再被使用时，该S Block将被

归还到空闲交换块管理器中。考虑到 UB 申请的效率以及防止发生在申请空闲块和归还空闲块间震荡，归还交换块时可不将空闲交换块从可用交换块链表中摘除。

2. 用满交换块链表：用来管理 UB 池中不可用 S Block 的双向链表，当链表中的某一个 S Block 的某个 UB 被归还时，该 S Block 变为可用，并根据 S Block 的类型将被转移到可用交换块链表或保留交换块链表中。
3. 保留交换块链表：用来管理 UB 池中保留 S Block 的双向链表，由于目前只有定时器模块用到了大小为 64 字节的保留 UB，可以在 UB 池初始化时，为管理 64 字节的 UB 池分配两个 S Block 添加到保留交换块链表中。与可用交换块管理不同的是保留 S Block 不用归还到空闲交换块管理器中。
4. 未保护交换块链表：用来管理 UB 池中不被保护的 S Block 的双向链表，该链表只有在支持 UB 保护的情况下才有用，在后面 UB 保护子模块中将进一步介绍。
5. UB 大小：表示一个 UB 池所支持申请的 UB 尺寸大小。
6. 交换块的 UB 个数：该 UB 池管理的 S Block 中的 UB 个数，由于每个 S Block 中能够划分的 UB 个数在同一 UB 池中相同，因此该个数可存放在控制块中。

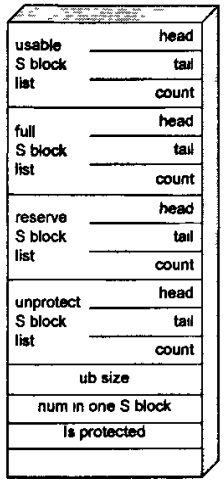


图 3.6 UB 池控制块结构示意图

(3) 交换块控制结构

正如前面所描述过的，交换块控制块时用来管理交换块内部的 UB 头和 UB 体的，其结构如图 3-7 所示，结构组成描述如下：

1. 节点：交换块在 UB 池中是以节点的形式存在于相关的链表中的，交

换块是通过节点一个个的连接被管理起来的。

2. 空闲 UB 控制链表：用来管理交换块中空闲 UB 的一个双向链表，实际链表中记录了空闲 UB 头的位置，即 UB 头作为链表中的一个节点。当有申请者向交换块申请 UB 时，从空闲 UB 控制链表的头部取出对应的 UB，并转移到已用 UB 控制链表的尾部。
3. 已用 UB 控制链表：用来管理正在被使用 UB 的一个双向链表，当链表中的 UB 被申请者归还时，将 UB 节点从该链表中摘除并添加到空闲 UB 控制链表的尾部。
4. UB 个数
5. UB 池控制块指针：用来保存该交换块所在位置（在哪个 UB 池中）的地址指针。
6. 下一个可用 UB 头指针：当空闲 UB 控制链表为空并且交换块中还有可用 UB 时，该指针指向了当前可用的 UB 头。

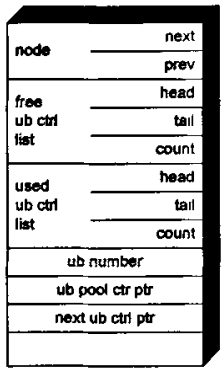


图 3.7 交换块控制块结构示意图

(4) UB 头/体数据结构

UB 头和 UB 体都存在于某个交换块的内部，根据 UB 体大小的不同交换块被分割的 UB 头/体的个数也会不同。在物理位置上可以采用“头体合一、尾部提交的” 的方法，如图 3-8 所示，UB 头和 UB 体在物理上是连续存放的。考虑到 UB 保护，采用尾部提交方式，即 UB 申请者实际使用的内存为 UB 体后面的一段，这样 UB 体的头部需要占用 4 个字节用于存放 UB 头指针。

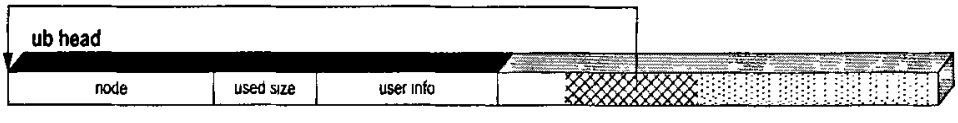


图 3.8 头体合一、尾部提交的存储方式

UB 头作为记录 UB 体使用情况的结构，由以下几部分组成：

1. 节点：UB 头作为交换块控制块某个链表的一个节点被管理前来的。
2. 实际使用尺寸：用来记录申请者实际使用了多大的内存空间。

3. 用户使用信息：用来记录 UB 申请者相关信息。

(5) UB 池中各实体间关系

UB 池控制块、交换块、UB 头和 UB 体的基本数据结构和相互关系如图 3.9 所示。图中黄色部分的表示 UB 池控制块，蓝色的块结构表示一个可用的交换块，红色的结构块表示用满的交换块，绿色的结构块表示可用 UB 头./体，褐色的结构块表示被占用 UB 头/体。从图 3.9 中可看出 UB 池交换块是分别通过一个链表管理可用的交换块和用满交换块的；两个曲线所包围的部分均表示一个完整的交换块，每个交换块通过两个链表分别管理空闲 UB 和被占用 UB。

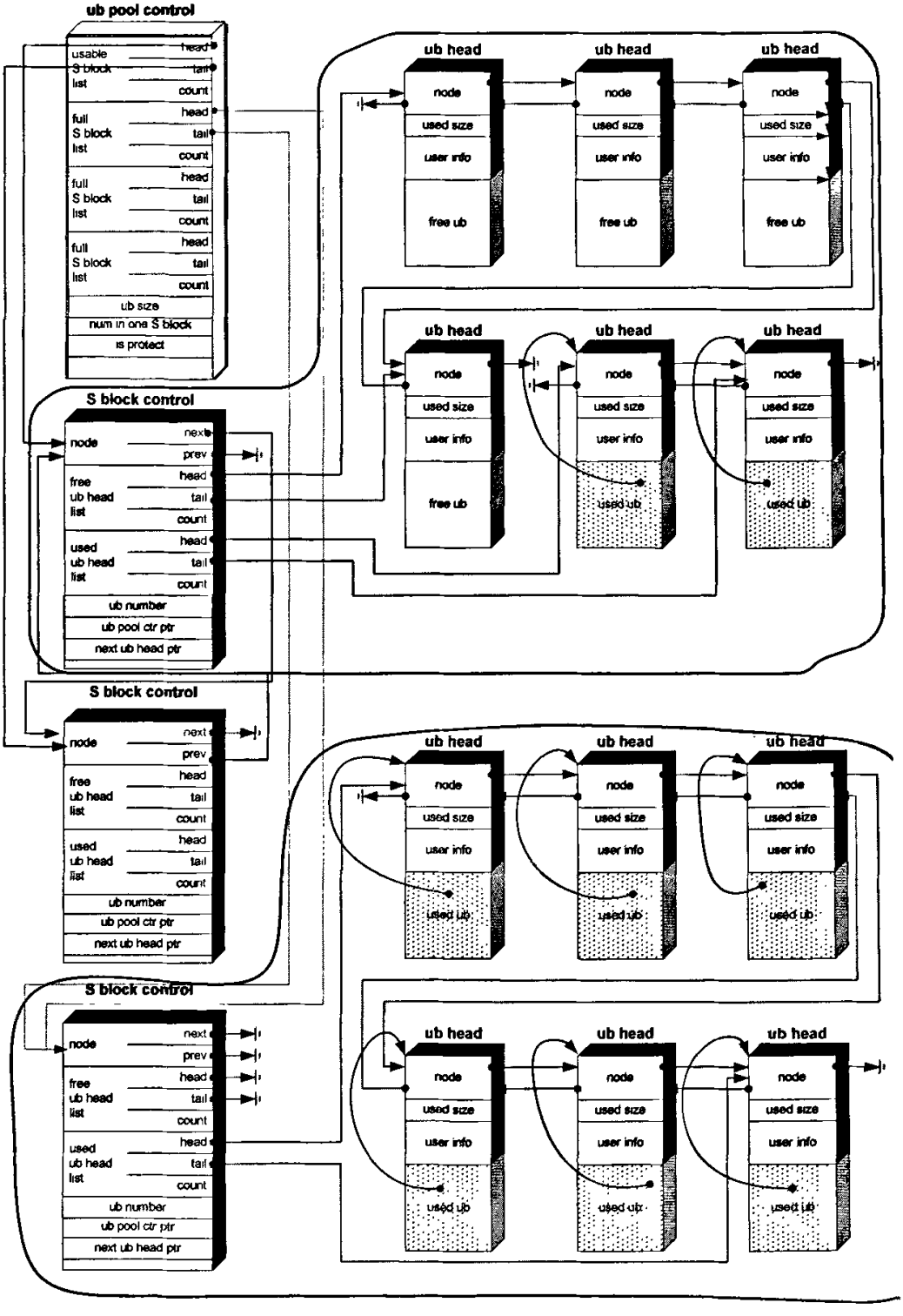


图 3.9 UB 池各实体关系图

3.2.3 模块初始化流程

该初始化流程负责整个 UB 动态管理模块的初始化工作，并向内存模块提供接口函数 R_DynaUBInit，即 UB 动态管理模块的初始化是做为整个内存模块初始化的一部分。该初始化流程主要包括空闲交换块初始化和系统内存池集初始化两部分，空闲交换块初始化在空闲交换块管理子模块已经介绍，这里针对内存池集的初始化过程进行一下描述，初始化流程如图 3.10 所示。

流程描述：

1. 合法性检查，主要对 UB 池配置表进行有效性检查，包括 UB 的大小是否为 64 的倍数等。
2. 为 UB 池创建在申请和释放 UB 时使用的互斥信号量。
3. 初始化 UB 池控制块中的各交换块控制链表。
4. 根据 UB 池配置表需要的初始交换块的个数向空闲交换块管理器申请空闲交换块。
5. 初始化空闲交换块并添加到可用交换块链表中。
6. 根据 UB 池配置表中保留交换块的个数向空闲交换块管理器申请空闲交换块。
7. 初始化空闲交换块并标记为 SBLOCK_TYPE_RESERVE 类型，同时添加到保留交换块链表中。

上述 UB 池集初始化是根据 UB 池配置表进行的，配置表的格式如图 3.11 所示，表中 UB 池的初始 UB 个数和保留 UB 个数可以为零，即所谓的零配置；若 UB 池的 UB 个数不为零，则有个初始化交换块并添加到相应交换块控制块链表中的过程。交换块初始化主要是初始化交换块控制块中各控制链表和当前可用 UB 控制头地址等；为了减小耗时（UB 申请流程可能也需要初始化交换块），这里不能一步将全部空闲 UB 加入到空闲 UB 链表中，而是将空闲链表的组织操作分摊到每次 UB 申请/释放中去。交换块初始化只包括如下过程：

1. 初始化交换块控制块中已使用 UB 链表（used ub ctrl list）。
2. 初始化交换块控制块中空闲 UB 链表（free ub ctrl list）。
3. 初始化 next ub head ptr 字段，指向当前可用 UB 头。

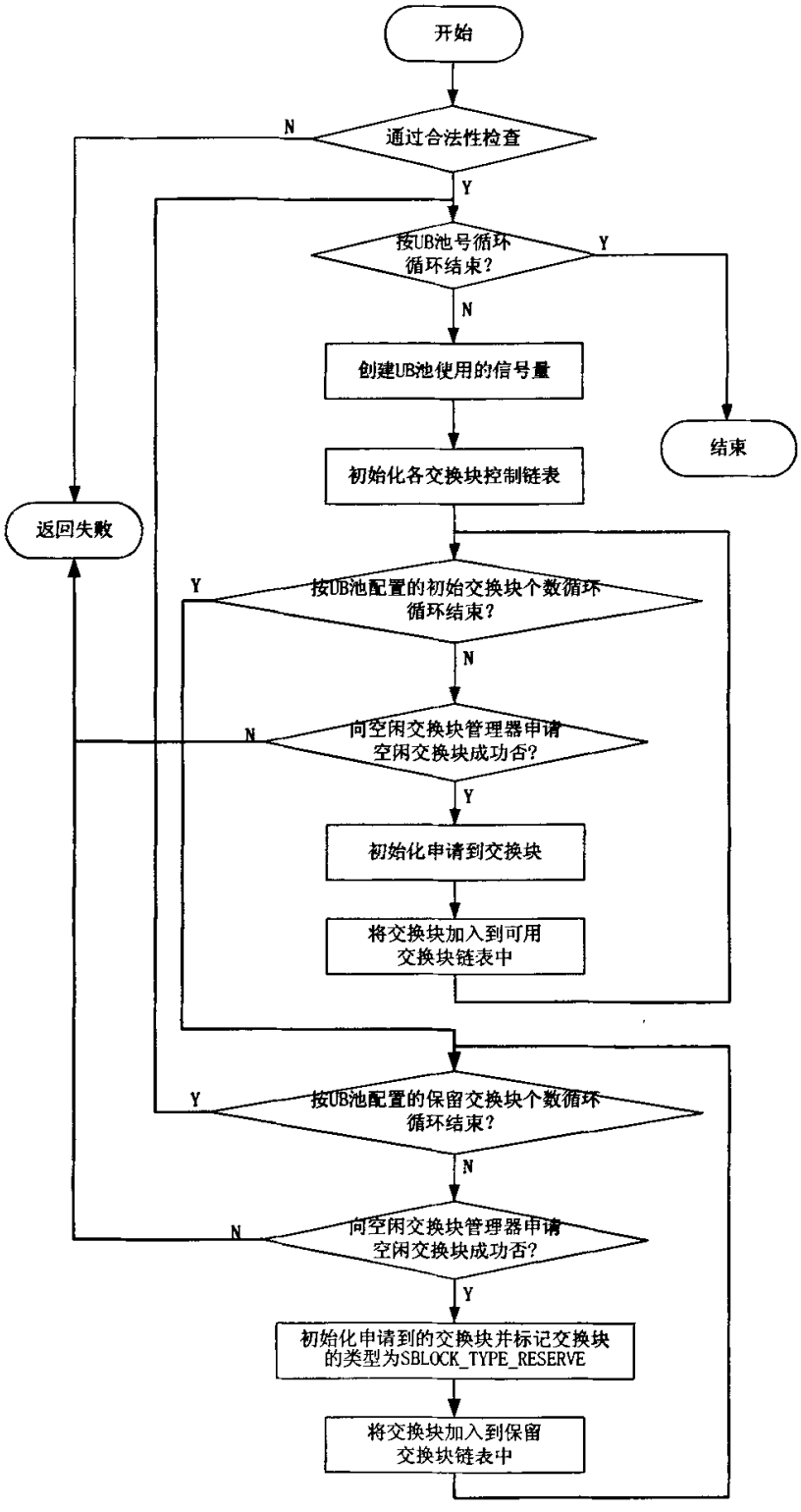


图 3.10 UB 池集初始化流程

Size	Config S Block	Reserve S Block
64	0	2
64*2	0	0
64*4	0	0
64*8	0	0
64*16	0	0
64*32	0	0
64*64	0	0
64*128	0	0
64*256	0	0

图 3.11 UB 池配置表格式

3.2.4 UB 申请流程

系统 UB 的申请是根据申请 UB 的大小找到相应的 UB 池控制块，进而寻找一个可用的 S Block，最后找到一空闲 UB 体返回给申请者，UB 申请流程如图 3.12 所示。

流程描述如下：

1. 根据申请尺寸从内存池集中寻找合适的 UB 池。
2. 在 UB 池中找到一个可用的 S Block（若 UB 池中没有向空闲交换块管理器申请并添加到 UB 池可用 S Block 链表中）。
3. 进入信号量保护。
4. 从 S Block 中申请空闲 UB。
5. 记录 UB 使用者相关信息。
6. 若 S Block 中的 UB 被用完，将被转移到用满 S Block 链表中。
7. 释放信号量。
8. 返回 UB 指针给使用者。

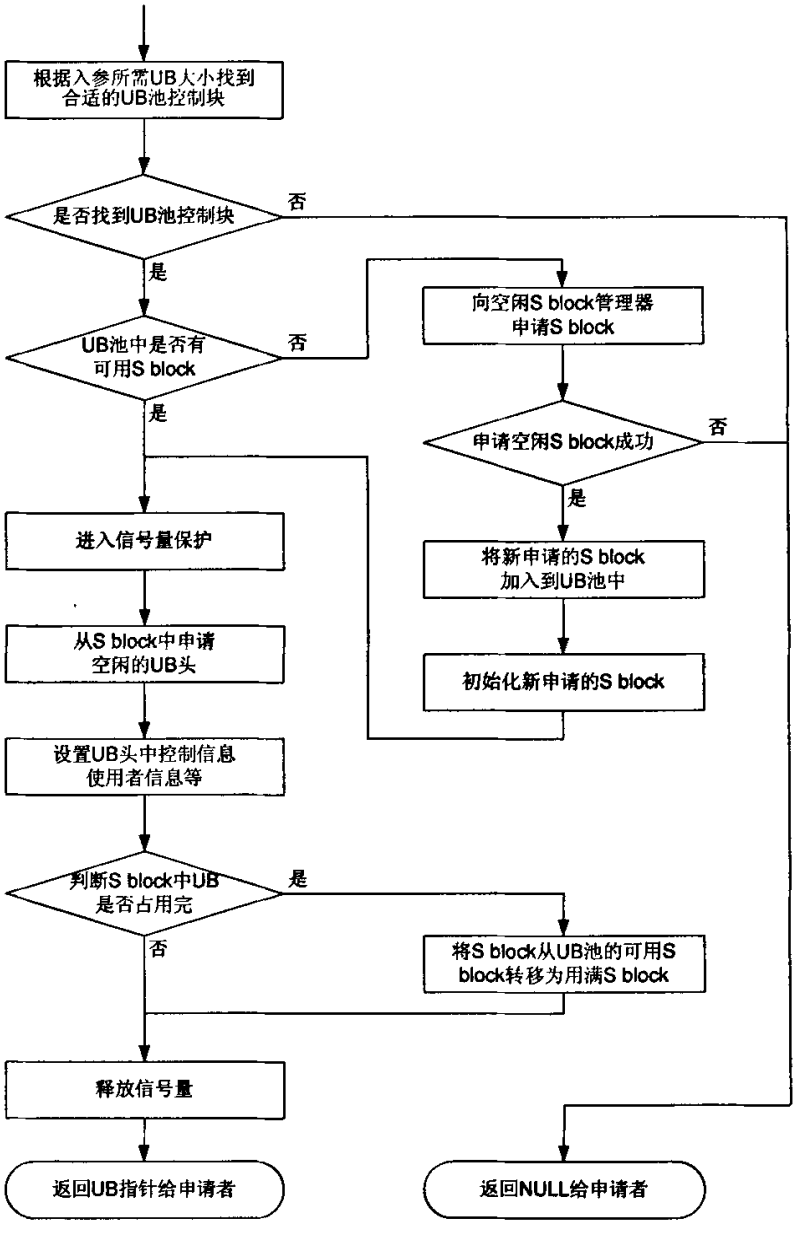


图 3.12 UB 申请流程

3.2.5 UB 释放流程

系统 UB 的释放是根据传入的 UB 体指针找到相应的 UB 控制头，并把 UB 头从其所在 S Block 的已使用链表中转移到空闲链表中，在整个流程中需要考虑到资源互斥问题，UB 释放流程如图 3.13 所示。

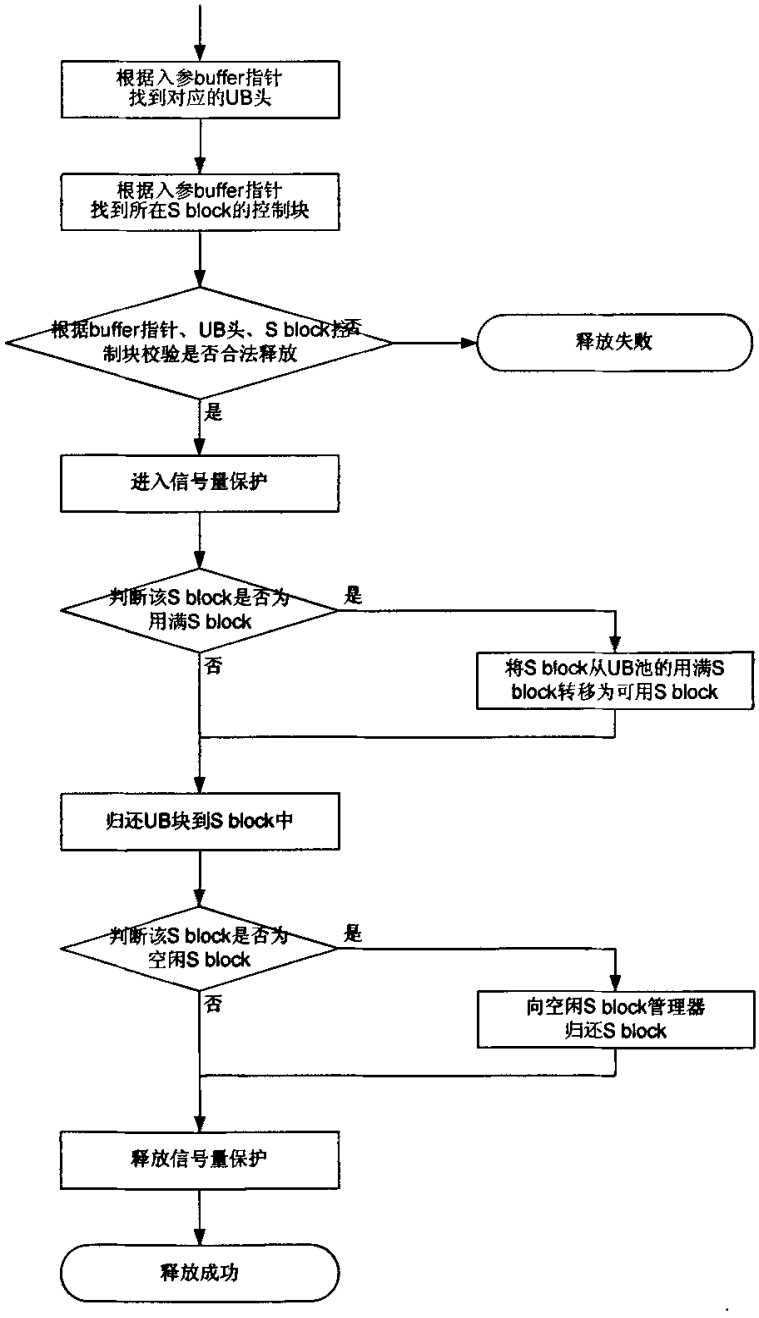


图 3.13 UB 释放流程

流程描述如下：

1. 根据传入的 UB 体指针进行偏移，获取其对应 UB 头的地址。
2. 进一步获取 UB 头/体所在 S Block 的控制块。
3. 进入信号量保护。
4. 若 UB 头/体所在 S Block 处于 UB 池控制块的用满链表中，将其转移到可

用链表中。

5. 将 UB 头从使用 UB 控制链表中转移到空闲 UB 控制链表中。
6. 若此时 S Block 变为空闲将其归还到空闲 S Block 管理器中。
7. 释放信号量保护。

3.2.6 系统 UB 使用告警功能

3.2.6.1 UB 告警分类

告警主要指告警上报和告警恢复，UB 使用告警按 S Block 的使用情况分为以下 3 种：

1. S Block 使用过多告警：为 S Block 的使用设定一个告警门限（例如 60%），当 S Block 的使用率超过告警门限时，发送 S Block 使用过多的告警；当 S Block 使用率降到门限以下，发送告警恢复。
2. S Block 用完告警通知：当向空闲交换块管理器申请空闲 S Block 失败时，发送告警通知到后台。
3. UB 池占用 S Block 过多告警：为 UB 池使用 S Block 设置一个告警门限（例如 50%），当某一 UB 池使用的 S Block 占全部 S Block 的比例超过门限值，发送 UB 池占用 S Block 过多告警；当此 UB 池使用 S Block 小于门限值，发送告警恢复。

3.2.6.2 UB 告警码定义

1. S Block 使用过多告警

事件号：EV_ALARM_REPORT_EX（告警），EV_ALARM_RESTORE_EX（恢复）

项目	内容
告警码中文名	内存交换块使用过多
告警上报者 **格式（模块-进程）** **如：OSS-进程名**	OSS
告警上报单板（可能产生该告警的单板）	所有单板

告警码标识符		ALARM_SBLOCK_USE_OVER_THRESHOLD
告警码详细描述	中文	内存交换块使用过多
	英文	Too many swap memory blocks are used
告警原因详细描述（触发原因，是否频繁发生）	中文	内存交换块使用数超过门限
	英文	The number of used swap memory blocks is over threshold
该告警是否被其他告警屏蔽（比如 E1 滑码告警在 E1 电缆没有接入信号时，是没有意义的）	中文	否
告警级别，四个告警级别： 致命（一级）：ALARM_LEVEL_FATAL 严重（二级）：ALARM_LEVEL_SERIOUS 一般（三级）：ALARM_LEVEL_COMMOM 轻微（四级）：ALARM_LEVEL_SLIGHT		ALARM_LEVEL_SLIGHT
附加信息记录详解		T_SBlockOverThreshold tSBlockRate;
告警建议采取的措施（消除方法）	中文	无
	英文	NULL
网元类型（Msc、Vlr、Sgsn、Ggsn、Hlr、BSC、所有网元）		所有网元
该告警可能会引起的其它问题（对系统的影响）	中文	无
	英文	no

其中：

```
typedef struct tagSBlockOverThreshold
{
    WORD32      dwMaxSBlockNum;        /* 最大可用交换块数 */
```

```
    WORD32      dwCurSBlockNum;      /* 当前使用交换块数 */
    BYTE        ucSBlockThreshold;   /* S Blcok 使用告警门限 */
}T_SBlockOverThreshold;
```

2. S Block 用完告警通知

事件号：EV_ALARM_INFORM_EX

项目		内容
告警码中文名		内存交换块缺失
告警上报者 格式（模块-进程） 如：OSS-进程名		OSS
告警上报单板（可能产生该告警的单板）		所有单板
告警码标识符		INFORM_OSS_SBLOCK_LACK
告警码详细描述	中文	空闲内存交换块缺失
	英文	Free swap memory block is lacking
告警原因详细描述（触发原因，是否频繁发生）	中文	空闲内存交换块申请失败
	英文	The application of free swap memory blocks fails
该告警是否被其他告警屏蔽（比如 E1 滑码告警在 E1 电缆没有接入信号时，是没有意义的）	中文	否
告警级别，四个告警级别： 致命（一级）：ALARM_LEVEL_FATAL 严重（二级）：ALARM_LEVEL_SERIOUS 一般（三级）：ALARM_LEVEL_COMMOM 轻微（四级）：ALARM_LEVEL_SLIGHT		

附加信息记录详解		T_SBlockLackErr tSBlockLackErr;
告警建议采取的措施（消除方法）	中文	无
	英文	NULL
网元类型(Msc、Vlr、Sgsn、Ggsn、Hlr、BSC、所有网元)		所有网元
该告警可能会引起的其它问题（对系统的影响）	中文	无
	英文	no

其中：

```
typedef struct tagSBlockLackErr
{
    WORD32      dwSize              /* 申请内存大小 */
    WORD32      dwCurSBlockNum;     /* 当前使用交换块数 */
    WORD32      dwTid;              /* 申请者任务 ID */
    WORD32      dwPno;              /* 申请者进程号 */
    WORD32      dwMsgId;            /* 当前消息号 */
}T_SBlockLackErr;
```

3. UB 池占用 S Block 过多告警

事件号：EV_ALARM_REPORT_EX（告警），EV_ALARM_RESTORE_EX（恢复）

项目		内容
告警码中文名		UB 池占用内存交换过多
告警上报者 **格式（模块-进程）** **如：OSS-进程名**		OSS
告警上报单板（可能产生该告警的单板）		所有单板
告警码标识符		ALARM_UBPOOL_USE_SBLOCK_THRESHOLD
告警码详细描述	中文	某 UB 池占用内存交换块过多
	英文	Too many swap memory blocks are used by one UB pool
告警原因详细描述	中文	某一 UB 池占用内存交换块数超过门限

（触发原因，是否频繁发生）	英文	The number of swap memory blocks those used by one UB pool is over threshold
该告警是否被其他告警屏蔽（比如 E1 滑码告警在 E1 电缆没有接入信号时，是没有意义的）	中文	否
告警级别，四个告警级别： 致命（一级）：ALARM_LEVEL_FATAL 严重（二级）：ALARM_LEVEL_SERIOUS 一般（三级）：ALARM_LEVEL_COMMOM 轻微（四级）：ALARM_LEVEL_SLIGHT		ALARM_LEVEL_SLIGHT
附加信息记录详解		T_UBPoolUseSBlockThreshold tUBPoolUseRate;
告警建议采取的措施（消除方法）	中文	无
	英文	NULL
网元类型（Msc、Vlr、Sgsn、Ggsn、Hlr、BSC、所有网元）		所有网元
该告警可能会引起的其它问题（对系统的影响）	中文	无
	英文	no

其中：

```
typedef struct tagUBPoolUseSBlockThreshold
{
    WORD32      dwMaxSBlockNum;      /* 该类交换块最大可用数 */
    WORD32      dwCurSBlockNum;      /* UB 池当前使用交换块数 */
    WORD32      dwSize               /* UB 池中 UB 大小 */
    BYTE        ucSBlockThreshold;   /* UB 池使用 S Blcok 告警门限 */
}T_ UBPoolUseSBlockThreshold
```

3.2.6.3 UB 告警的实现

在实际进行 UB 使用告警时需要考虑到以下几点：

1. 为防止相同告警突发性，对连续的相同告警至上报一个。
2. 为防止连续跳变告警的突发性，设定一个合理的告警时长，在这个告警时长内只上报一次告警和告警恢复。
3. 考虑到 UB 告警和告警恢复同样使用 UB，存在 UB 恢复时同时存在 UB 告警，要避免互相嵌套（虽然这种可能性比较小）。

第一种和第三种 UB 使用告警类型都属于告警，因此有告警和相应的告警恢复，UB 使用告警和告警恢复分别是在 UB 申请和释放的过程中实现的；第二种 UB 使用告警属于通知，因此没有告警恢复。3 种 UB 使用告警或通知在流程上比较相似，以 S Block 使用过多告警为例，其告警和恢复流程分别如图 3.14 和图 3.15 所示：

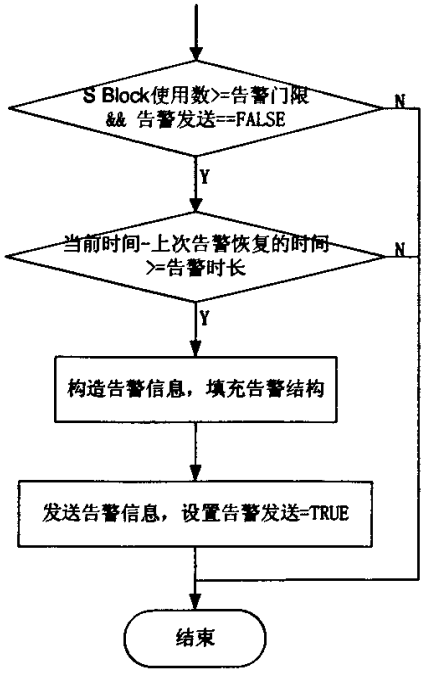


图 3.14 S Block 使用过多告警流程

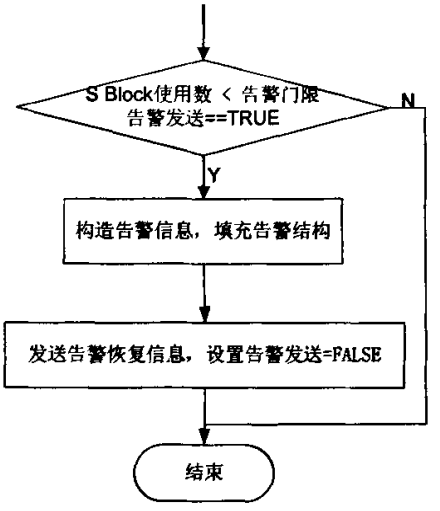


图 3.15 S Block 使用过多告警恢复流程

3.2.7 UB 使用统计功能

在 S block 中有占用 UB 链表，可用于方便的进行遍历获取 UB 使用信息。遍历各 UB 池可用 S block 链表、用满 S block 链表、可用保留 S block 链表、不支持 UB 保护 S block 链表，再遍历各 S block 中占用 UB 链表，可提取其中各种信息，并进行统计。

由于系统 UB 体中有足够的使用信息，因此对于系统 UB 其 UB 头中“user info”字段可为空。

UB 使用情况统可以按 UB 的位置（被哪个任务或进程占用）和用途（被占用的 UB 是用来做什么的），进行横向及纵向的的统计，可对一类大小的 UB 进行统计也可对所有占用的 UB 进行统计。具体分布若采用动态变化设计上过于复杂，且需要占用大量统计信息内存空间，考虑只在需要这些信息时进行遍历获取。可以在通过后台定时查询、调试函数查询、发送告警时进行统计 UB 使用情况。

3.3 系统 UB 保护子模块

UB 保护的目的和方法参见第二章内存保护子模块，这里主要介绍基于交换块方式的系统 UB 动态管理是如何实现 UB 保护的。

3.3.1 UB 保护下的内存排布

目前系统虚拟页的尺寸为 4096 字节，利用虚拟内存页表属性的设置来实现 UB

保护时，至少对一页大小的内存进行保护。

对于 4K 大小的交换块，刚好占用一页的虚拟空间，为支持 UB 保护，一个交换块内只能有一个 UB，交换块之间加入一页大小的空洞（未映射物理内存的虚拟内存），通过 UB 申请尾部提交的方式能够有效的防止 UB 向后越界访问。

对于 32K 大小的交换块，为充分利用内存资源，可以实现在一个交换块支持多个 UB，每个 UB 的后面保留 4K 的不可访问内存作为保护页，但是由于交换块内的 UB 尺寸是不固定的，在空闲交换块的申请/释放过程中需要动态的设置内存的访问属性，这样会导致性能下降，实时性的不到满足。为此采用另一种方式，一个 32K 的交换块只支持一个 UB，交换块之间增加 32K 的内存空洞来实现对 UB 向后越界访问的保护。

内存保护模式下内存的排布如图 3.16 所示：

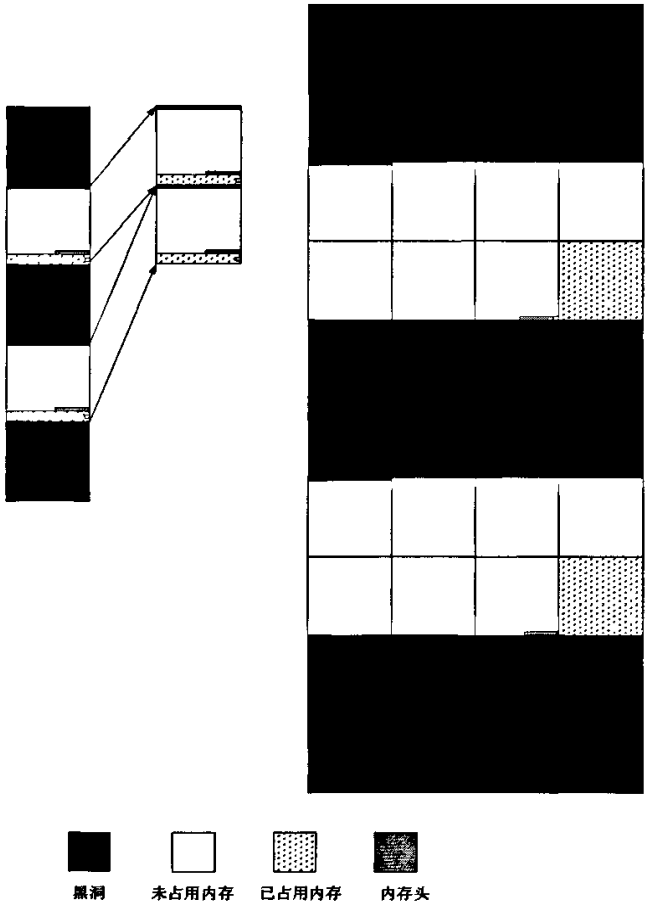


图 3.16 UB 保护下的内存排布图

3.3.2 动态 UB 保护的支持

由于交换块之间的空洞是在初始化过程中组织好的，因此可以支持系统运行过程中动态开启 UB 保护功能，由于系统内存资源有限，并且打开 UB 保护功能时内

存浪费太多，需要支持对单个UB池的UB保护功能的动态开启和关闭，当然也可同时对多个甚至全部UB池的UB进行保护。

考虑到保留UB全部为定时器模块使用，为简化UB动态保护功能的设计，对保留UB不做保护。

UB保护子模块需要提供两个接口函数分别实现动态打开UB保护和关闭UB保护功能。打开UB保护功能需要执行的操作如下：

1. 对相应UB池进行信号量保护；
2. 将相应UB池控制块设置为“保护中”标志；
3. 将UB池中可用交换块链表和用满交换块链表中的全部交换块转移到“unprotect S Block list”链表中，如果UB池中的交换块只能支持一个UB，则不需要做以上操作，因为这种情况下事实上已经支持UB保护了；
4. 释放相应UB池的信号量保护。

打开UB保护状态下申请和释放UB是需要注意以下两点：

1. 申请UB时，若可用交换块链表为空，需要从空闲交换块管理器申请新的交换块时，要根据UB池控制块的“是否支持UB保护”字段设置交换块中能够支持的UB数。如果UB池支持UB保护，设置交换块能支持的UB数为1，并直接将交换块加入到用满交换块链表；否则设置成实际能够支持的UB数，并将交换块加入到可用交换块链表中。

2. 释放UB时，若UB所在的交换块变为空闲状态，归还空闲交换块到空闲交换块管理器；如果该交换块处于保护状态下（支持的UB数为1），可同时将交换块添加到可用交换块链表中，以免空闲交换块管理器频繁申请空闲交换块，否则不需要上述操作。若UB所在的交换块变为可用状态，说明此交换块之不支持UB保护的，继续保存在未保护交换块链表中。

撤销UB保护需要执行以下操作：

1. 对相应UB池进行信号量保护；
2. 将相应UB池控制块设置为“非保护”标志；
3. 将UB池未保护交换块链表中的交换块根据其实UB的使用情况，分别追加到可用和用满交换块链表中；
4. 释放相应UB池的信号量保护。

第四章 测试描述

4.1 测试项目描述

4.1.1 初始化

4.1.1.1 测试内容

内存初始化的入口函数为 Mem_SysInit，依次完成如下操作：

1、分配核心数据区。

2、获取页长。

3、获取页长位移。

4、初始化内存池集控制块。

5、初始化系统内存池集。

6、初始化用户内存池集。

上述 6 步均要测试，主要手段是通过打印命令来检测初始化结果。另外，对于合法性检查，制造各种不合法情况（通过修改 UB 的配置表），初始化过程必须检测出这种情况。合法性检查包括：

- 第一个 UB 尺寸必须大于 MEM_UB_SPAN。
- 内存池内存尺寸必须是 MEM_UB_SPAN 的倍数，目前 MEM_UB_SPAN 为 64。
- UB 尺寸必须按序增长，即数组下标越大，对应的尺寸越大。
- UB 的保留个数必须小于总的 UB 个数。

页长受操作系统、CPU 类型、内存保护开关的影响。如果内存保护开关不打开，保护页大小为 64 字节，如果打开，则为 4096 字节。

页位移是根据页长获得的，主要的目的是为了提高效率。

内存划分了两套 UB 池集，两者的初始化流程完全相同。一套 UB 池集有一个数据结构对其进行描述，包括互斥信号量、内存池个数、最大的有效 UB 尺寸、内存池控制块首指针。

内存初始化包括合法性检查、最大有效 UB 尺寸的获取、创建互斥信号量，创建队列、本内存池集需要的总的页数（包括虚拟页）、UB 头指针入队。

如果加入内存保护，其初始化流程略有不同。如果 UB 保护开关不打开，则采用的是加物理页保护的方式，如果 UB 保护开关打开，则采用虚拟页的保护方式。

这两种方式要分别测试。

另外，对于进程栈和数据区的分配，采用了先保留，再提交的方式，这一部分也要测试。测试的目的主要在于能否为所有进程分配进程栈和数据区。

4.1.1.2 测试方法

后面的整合测试可以测试出总的结果。但对于初始化部分，主要的测试手段是打印信息。下面逐项列出：

分配核心数据区：打印出分配前的可用物理内存，分配完毕之后，打印出成功与否的信息、分配了多少内存，还剩余多少内存。

获取页长：系统启动之后，打印出页长值。并根据内存保护是否打开、操作系统种类（NT/VxWorks）、CPU 种类，来验证页长值是否同期望的一样。

合法性检查：主要是通过修改 UB 配置表，制造各种不合法情况，以验证合法性检查的有效性。

4.1.2 申请、释放内存

4.1.2.1 自测内容

这一部分的测试包括三个方面：

- 基本功能测试。
- 特殊情况测试。
- 性能测试。

基本功能测试在于测试在正常情况下，申请者能否申请到可用内存，并且，获得的内存靠后分配。靠后分配的测试可以借助于内存保护。假设申请一段长度为 100 个字节的内存，如果内存保护打开，那么，当写的长度超过 100 个字节的时候，会触发异常。如果没有触发异常，证明靠后分配错误。

特殊情况测试。我们界定如下情况为特殊情况：

- 内存池满。
- 多个不同优先级的任务同时申请内存。
- ARM 下返回的内存地址必须按照四字节对齐。

前两种情况都要通过编写测试代码来验证。后面一种情况通过打印返回值来验证。

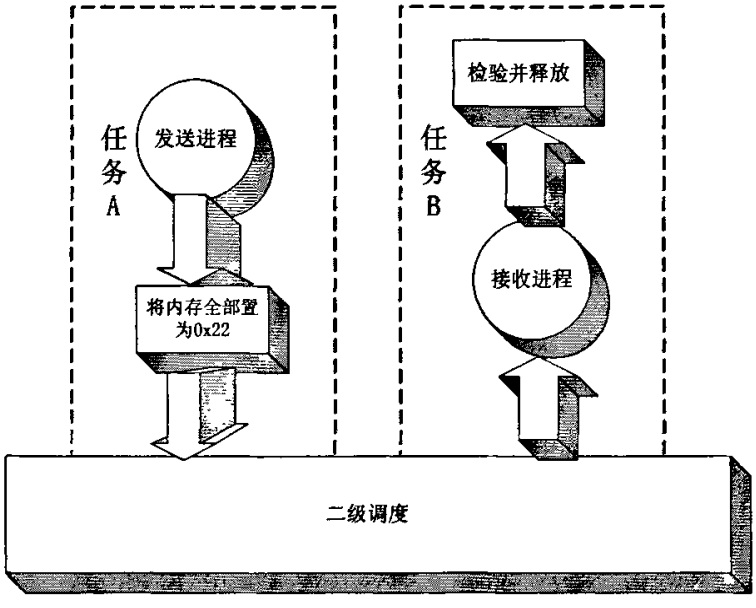
性能测试。性能测试在于测试内存申请所耗费的时间，从而给出性能优化的参

考值。性能测试也必须通过编写测试代码。

4.1.2.2 测试方法

基本功能测试：创建两个进程，假设进程名分别为 Sender 和 Receiver，分属不同的调度任务，任务优先级不同（可以测试内存池集的信号量保护是否正确）。Sender 进程以时间间隔 dwSendInterval 申请大小为 dwMemSize 的内存，并且内存内容全部置为 0x22，而后将内存指针、内存大小发送给 Receiver 进程。接收进程收到之后，同步延时 dwRecInterval 秒。而后根据收到的消息，对内存内容进行校验，如果内容不全为 0x22，则证明有问题，此时，应该停止发消息（方法是将 dwSendInterval 和 dwRecInterval 置为 0xffffffff）。如果全为 0x22，则将内存释放。dwSendInterval、dwMemSize 和 dwRecInterval 作为全局变量，可以在 shell 中直接修改。

内存分配模块的测试模型如图 4.1 所示：



内存分配测试模型

图 4.1 基本功能测试

特殊情况测试参见整合测试的测试方法。对于内存池满，将发送任务的优先级调高，等待一段时间后，会造成 UB 池满的情况。在这样的状态下，通过打印命令，可以观测 UB 申请是否正常。

4.1.3 整合测试

4.1.3.1 测试方法

创建多个不同优先级的发送任务和接收，每个接收任务各有一个信箱。每个发送任务以随机的时间间隔，随机的申请个数和尺寸申请 UB。每次申请完一个 UB 之后，随机选取一个任务信箱，将 UB 头指针发送出去，发送完毕，按 UB 尺寸进行计数。接收任务收到消息之后，释放 UB，并按 UB 尺寸进行计数。每个任务都有一张表，用以计量发送和接收情况。通过调整发送任务和接收任务的优先级，可以模拟出各种情况。并创建一个最高优先级的任务，将发送情况和接收情况打印出来，用以验证 UB 申请流程的正确性。

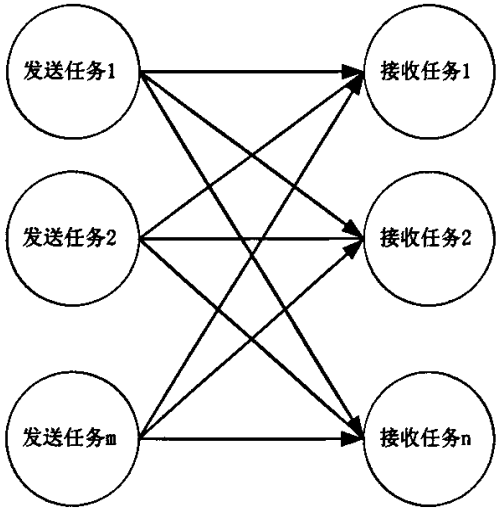


图 4.2 内存整合测试模型

4.1.4 UB 池保护

4.1.4.1 测试内容

对于每一个 UB，都加入了保护页。保护的方式如图 4.3 所示：

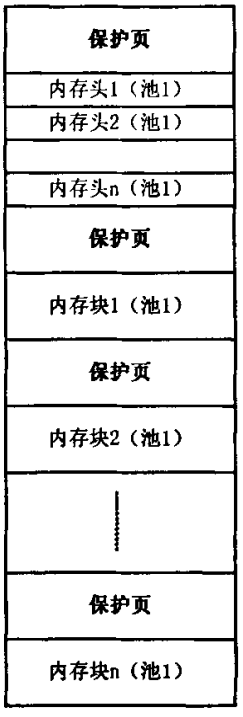


图 4.3 内存池集的保护

如果内存保护功能打开，当用户申请一块内存时，如果写出界，则会触发异常。自测的内容就是对这一部分功能进行测试。申请大小不同的内存，对每一块内存都进行写出界一个字节的操作。如果内存保护正确，则会按预期的设计报告异常。

4.1.4.2 测试方法

主要是通过编写测试代码。向测试者提供一个测试函数。输入的参数包括：

- 申请内存大小
- 使用哪一套内存池集

该测试函数实现如下功能：

1、根据入参，从合合适的地方申请内存。

2、写内存，并按内存大小+1 的长度进行写内存操作。

4.1.5 页表切换保护

4.1.5.1 自测内容

页表切换主要是对进程的私有栈和数据区进行保护。在进入进程空间的时候，可以对本进程数据和私有栈进行读写操作。如果离开进程空间，则所有的进程数据区和私有栈都不能进行写操作。

正常情况下，进程切换的原因由以下几种：

1、消息驱动进程调度。

2、同步调用返回。

3、同步延时返回。

4、远程过程调用返回。

上述几种情况都可以归结成如下两种情况：

对于普通消息，引发进程调度时，会由系统空间进入进程空间。调度完成之后，则由进程空间进入系统空间。

对于同步消息，引发进程调度时，也会由系统空间进入进程空间。在同步消息处理完毕之后，会将被阻塞的进程加入就绪队列，而后即出进程空间。当下一次处理刚刚解除阻塞态的进程时，会从任务空间同步返回到进程空间，而后恢复曾经被同步阻塞的进程继续执行。

4.1.5.2 测试方法

在空间切换处，加入测试代码。

1、进入进程空间前，将程序断住，而后写进程数据区或和栈区。

2、进入进程空间后，将程序断住，而后写进程数据区或和栈区。

3、退出进程空间后，将程序断住，而后写进程数据区或和栈区。

4.2 测试环境描述

内存测试不需要机架环境，但需要不同类型的单板。

4.2.1 硬件环境

- UIM 单板

- MPX86
- MNIC

这三种硬件单板，分别代表 PPC、X86 和 ARM。

4.2.2 软件环境

- Windows NT/2000 操作系统。
- Tornado 集成开发环境——PPC、X86 和 Arm
- VC 集成开发环境
-

实验结果：经验证，文中所述功能均实现，其中内存申请方案所需时间比较如下：

原申请方案

申请块	测试 1	测试 2	测试 3	测试 4	测试 5
64	686us	685us	677us	678us	677us
128	702us	699us	688us	688us	688us
256	732us	730us	723us	706us	706us
512	758us	764us	719us	726us	719us
960	800us	800us	733us	732us	731us
1984	989us	1000us	862us	862us	862us
3968	1074us	1072us	811us	811us	812us
8128	1143us	1107us	841us	834us	860us

采用静态交换块

UB	测试 1	测试 2	测试 3	测试 4	测试 5
64	712us	705us	707us	704us	704us
128	722us	715us	714us	713us	712us
256	749us	743us	773us	756us	743us
512	788us	781us	779us	780us	780us
960	840us	846us	837us	837us	836us
1984	917us	906us	905us	906us	904us
3968	1084us	1066us	1067us	1069us	1068us
8128	929us	915us	914us	914us	915us

采用动态交换块

申请块	测试 1	测试 2	测试 3	测试 4	测试 5
64	632us	636us	630us	629us	631us
128	685us	661us	661us	663us	660us
256	740us	717us	711us	716us	717us
512	847us	817us	815us	815us	816us
960	988us	945us	961us	948us	944us
1984	1319us	1227us	1225us	1243us	1225us
3968	1773us	1651us	1651us	1655us	1650us
8128	1210us	1048us	1049us	1046us	1049us

采用动态交换块时，需要调用 Malloc 申请内存，因此比较耗时。实际应用中，动态交换块使用的几率比较小，因此对系统影响很小。

信号量操作，新老方案都只有一级。

采用静态交换块和共享交换块，UB 的申请效率和老方案相当

新方案的 UB 池概念与老方案存在区别；现在实际上中间增加了一级交换块池，每个 交换块中包含小 UB 池。

结束语

论文工作总结

本论文是作者攻读硕士研究生期间主要研究工作的总结。论文描述了基于3G软件开发平台的内存管理方法的设计与实现，该方法的实现现在已经成功应用于3G基站的支撑软件底层模块内。

本文首先简要介绍了内存管理的应用及现状，由此引出了在应用中存在的瓶颈问题——如何保证动态申请内存和释放内存的实时性，将内存碎片限制在可控的范围之内，指出了内存管理的重要性。

在分析内存分配和保护基本原理的基础上，对国内外主要的内存管理的方法进行了对比，引入了动态管理，交换块的概念。

内存管理完成两个主要功能。一是内存分配管理，二是内存保护。对于长期运行、实时性要求高的系统，必须自己对内存进行管理。

静态配置UB的方法相比有一定优势，具有设计简单、实时性强、效率高等优点，在一定程度上缓解了UB配置问题，但又会带其它一些问题，比如向上申请会引起实时性变弱；系统堆内存扩展将VxWorks本身内存管理的碎片问题暴露出来。为解决上述问题，在保证现有UB管理优点的基础上，并考虑尽量合理有效的利用内存资源，将“交换块（S Block)”的概念引入到UB管理中来。引入“交换块”的概念后，实际上是在目前UB管理的基础上增加了一个层次，向内存池申请UB时，先要获取一个可用的交换块，进而获取已可用UB。交换块可以被多个内存池交替使用，根据UB尺寸的大小，一个交换块被分解成若干UB。交换块UB动态管理机制可以解决UB池间内存不能共享等问题。

对于UB保护，考虑到原来设计方案在打开UB保护功能时会浪费很多内存，甚至导致系统由于内存不够而不能正常启动，在设计时考虑支持动态打开UB保护功能，并支持对个别内存池进行保护，当然在内存资源够用时可以对全部UB池进行保护。

最后将所设计的下载到板子上验证，全面的测试项验证了其功能。

今后工作展望

本文所论述的内存管理方法，虽然在各个方面都表现出了其优良的性能，但还有许多待解决的技术难题。就本文来说，主要是研究如何保证动态申请内存和释放内存的实时性，将内存碎片限制在可控的范围之内，所考虑的都不是很复杂的条件。在本文的基础上还有许多事情要做。

（1）内存管理技术要应用到下一代移动通信系统中，要考虑许多实际应用中的问题，因此下一步研究可以综合考虑多种因素来进一步合理设计系统参数。

（2）内存保护方面，对所有静态配置的UB，按插入虚拟页的方式进行保护。

加入对 UB 的保护之后，对 PPC 体系，如果页数增加过多，理论上有可能会降低效率。另外，因为页的最小尺寸目前定为 4096 字节，因此，加入 UB 保护，对于不是页整数位的内存块，会浪费内存。

内存保护子模块的主要作用是在于问题定位。内存保护引入虚拟内存机制，通过页表属性的设置来决定内存的访问权限。引入内存保护的会引起程序运行效率下降，特别是打开页表切换保护下，页表频繁切换的时候。因此，内存保护实现的完备程度受运行效率的限制。

致谢

通过两年多的研究生生活学习，我学到了很多东西，不仅仅是学术上的，更多的是对待困难的态度上。王伟老师不断的教诲和鞭策使我终身受益。我非常感谢王老师这几年来孜孜不倦的教诲，正是他教会了我看待事物的方法。更重要的是他让我明白，一颗不断进取的心，对于生活和工作是多么的重要。王老师渊博的知识、严谨的治学态度和孜孜不倦的科研精神也将是我一生学习的榜样。

感谢硕士期间教导和帮助过我的各位老师，他们传授给我的知识和做人的道理，我一辈子都受用不尽。感谢实习期间教导和帮助我的各位同事，感谢他们永不厌倦的帮助，严于律己热心助人的态度将在以后的工作生活中永远激励我前进。

感谢师兄师姐们，感谢你们在我读研期间对我指导和帮助。感谢曾一起学习、生活、实习和工作过的各位同学，我们之间建立起了深厚的友情。他们有李伟硕士、齐伟硕士、许远昂硕士、王欣硕士、邹继业硕士、李海涛硕士、许楠硕士。

感谢我的家人，感谢他们深深无私的爱，他们的爱是我永恒的依靠！没有他们的无私奉献和牺牲，就没有我的现在和将来。焉得谖草，言树之背，养育之恩，无以回报，你们永远健康快乐是我最大的心愿。

最后，衷心感谢参加论文评审和答辩的各位专家教授！

参考文献

[1] 王立柱.C/C++与数据结构.北京清华大学出版社，2002

[2] Bruce Eckel.Thinking in C++.Alan Apt，2000

[3] Michael Barr.C/C++嵌入式系统编程.于志宏译.北京中国电力出版社，2001

[4] 吕京建.面向21世纪的嵌入式系统综述.http://www.bol-system.com.2002

[5] 沈连丰，宋铁成，叶芝惠.嵌入式系统级开发应用..北京电子工业出版社，2005

[6] 田泽.嵌入式系统开发与应用教程. 北京：北京航空航天大学出版社，2005

[7] 罗蕾.嵌入式实时操作系统及应用开发. 北京：北京航空航天大学出版社，2005

[8] 中兴通讯深圳研究所产品开发部ZXH01系列产品开发资料，2005，2006，2007，内部资料。

[9] Molina H G, Ullman J D,Widom J. Database system inplementation[M] EngleWood Cliffs：Prentice-Hall，2000

[10] 张素琴，吕映芝，蒋维杜，戴桂兰 编译原理 清华大学出版社 2005，2

[11] Deeamtech，王勇等译，Programming for Embedded Systems。电子工业出版社。2002

[12] Molina H G, Salem K, Main memory database system An overview [J]。JEEE TransKnow ledeg and Data Engineering，1992

[13] 许海燕，付炎。嵌入式系统技术与应用。机械工业出版社。2004

[14] 刘尉悦，张万生等，VxWorks操作系统及实时多任务程序设计。2001

[15] 孔祥营 柏桂枝。嵌入式实时操作系统VxWorks及其开发环境Tornado。中国电力出版社，2002

[16] Kirk Zurell。嵌入式系统的C程序设计。艾克武等译。机械工业出版社，2001

[17] Y.Kim and S.H.Son。Supporting Predictability in Real-Time Database Systems.IEEE Real-Time Technology and Application Symposium,Boston,MA ,June 1996:38~48

[18] 陈鑫。嵌入式软件技术的现状与发展动向。2001

[19] 梁华。嵌入式系统：数字化产品的核心。2001

[20] Kaushik Ghosh。A Survey of Real-time Operating System,2002

[21] Jean J. Labrosse 著。邵贝贝译。源代码公开的实时嵌入式操作系统。北京中国电力出版社。2002.6

[22]Jean J. Labrosse。Embedded System Building Blocks,Second Edition Published

by R&D Books, CMP Media,Inc. 1999.12

[23] Khawar M. Zuberi. Real-Time Operating System Services For Networked Embedded System: Ph.D. Thesis. 1998

[24] 汤子赢 哲凤屏 汤小丹。计算机操作系统。西安电子科技大学出版社。

[25] 赫伯特.希尔特。C 语言大全。电子工业出版社。

[26] 舒良才，刘云生 实时内存数据库的数据管理 计算机世界 1999 年第 40 期

[27] 刘云生，吴绍春，李国徽等 一种实时内存数据库组织与管理方法。

[28] 杨波，张效义，面向嵌入式应用的通用型操作系统，微型机与应用，Vol.23 2004

[29] Tornado 2.0 Online Manuals—>VxWorks programmer's guide

[30] Tornado 2.0 Online Manuals—>VxWorks Reference Manual

[31] http://www.windriver.com

研究成果

参与项目：

2006 年 12 月～2007 年 03 月 ：

3G 统一平台 CPTool 抓包工具研发，CPTOOL 是用于 3GBTS 侧的调试工具，用于版本运行过程中研发故障定位。主要实现对 Rdup，Clp，485，HirsHdlc，媒体流，DSMAB 上 E1 检测流，DSMAB/DSMC 上转发代理 Agent 包，BPM_BAP 上 Disco 包，DSMC 上 Arp 报文，Bfd 报文的截获，可抓包的单板有 CCM，DSMAB/DSMC，CHM（HDR5500，1X5000，CHM2_6800，CHM3），RMM，TRX 等，通过设定过滤条件抓包可以诊断前台 BTS 单板的通信状况。

2007 年 03 月～2007 年 12 月 ：

3G技术时间模块维护级升级。时间技术是CDMA关键技术之一。在系统中主要体现在16chip中断，pp2s中断，时区，夏令时，闰秒，16chip时间，系统软时钟，单板时间，精确时钟，TOD,自制TOD,辅助时钟等等。

2007 年 05 月～2007 年 12 月 ：

高温工装项目部分测试项编码及协助测试。

附录

术语，缩略语：

UB	User Buffer	分配给内存申请者的内存块
S Block	Swap Block	交换块
OSS	Operating System Subsystem	操作系统子系统
VOS	Virtual Operating System	虚拟操作系统
PCB	Process Control Block	进程控制块

UB 体：实际使用的一块内存单元。

UB 头：描述 UB 体使用情况的数据结构。

UB 池：相同尺寸的 UB 块属于同一个 UB 池，每个内存池由 1 个内存池控制块和若干交换块组成。

UB 池控制块：一个 UB 池用来管理交换块使用的数据结构，一个 UB 池控制块管理若干个交换块，其中的交换块个数不固定。

交换块(SBlock)：为多个 UB 池所共享的一大块内存，每个交换块由 1 个交换块控制块和若干同等数量的 UB 头和 UB 体组成。

交换块控制块：一个交换块用来管理 UB 申请及释放的数据结构，一个交换块管理若干个 UB 头及 UB 体。