|  |
| --- |
|  |
| Codezero源码分析 |
| ——lib目录 |

|  |
| --- |
|  |

|  |  |
| --- | --- |
| 当前版本： | 1.0 |
| 日期： | 2014-07-07 |
| 作者： | 徐伟健 |

版本历史

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 版本状态 | | 作者 | | 参与者 | | 起止日期 | | 备注 |
| 1.0 | 徐伟健 | |  | | 2014-07-07 ~ 2014-07-18 | | 初稿 | |
|  |  | |  | |  | |  | |
|  |  | |  | |  | |  | |

1. 概述

本目录（src/lib）主要包含用于内核使用的基本库函数

1. 总体功能描述

主要包含以下内容：

位操作（bit.c）

ID 分配（idpool.c）注：用于内存空间 ID 以及线程 ID 的分配

内存缓存区操作（memcache.c）

互斥／信号量（mutex.c）

格式化输出（printk.c）

单字符输出（putc.c）

字符串处理（string.c）

线程等待与唤醒（wait.c）

注意，实际上还有大量功能处于这些源代码引用的头文件中，在以下代码分析中部分涉及这些头文件。

1. SConscript文件

SConscript 文件比较简单，代码如下：

# Inherit global environment

Import('env') #导入 SConstruct 中的 env

# The set of source files associated with this SConscript file.

src\_local = ['printk.c', 'putc.c', 'string.c', 'bit.c',

'wait.c', 'mutex.c', 'idpool.c', 'memcache.c']

obj = env.Object(src\_local) #编译成目标文件

Return('obj') #返回

仍然使用默认项目根目录下的SConstruct文件中的env作为构造环境

本SConscript将该目录下的所有源代码都编译成目标文件，并将目标文件集合以 obj 返回

1. bit.c
   1. 功能描述（对于复杂操作建议画出流程图）

主要包含基本的位操作：

\_\_clz 数给定数前面有多少连续二进制0

find\_and\_set\_first\_free\_bit 找给定内存区域第一个不为0的位

check\_and\_clear\_bit 检测给定内存区域某位是否为1并清除

check\_and\_set\_bit 检测给定内存区域某位是否为0并置1

setbit 设置某些位为1

clrbit 清除某些位（清除为置0）

tstbit 检测某些位是否为1

tstclr 检测某些位是否为1，并清除那些位

具体函数参数：

unsigned int \_\_clz(unsigned int bitvector)

返回 bitvector 中最开始连续二进制0的个数

int find\_and\_set\_first\_free\_bit(u32 \*word, unsigned int lastbit)

返回 word 指针开始（记为相对位置0）到第 lastbit 个二进制位为止（记为相对位置 lastbit - 1）中顺序出现的第一个不为0的相对位置，若没有找到（全为0）则返回 -1

int check\_and\_clear\_bit(u32 \*word, int bit)

检查（word 开始的内存中）第 bit 位是否为 1。若是则置 0 返回 0；否则返回 -1

int check\_and\_set\_bit(u32 \*word, int bit)

检查（word 开始的内存中）第 bit 位是否为 0。若是则置 1 返回 0；否则返回 -1

static inline void setbit(unsigned int \*w, unsigned int flags)

将 w 指针指向变量中 flags 对应为 1 的位，设为 1

static inline void clrbit(unsigned int \*w, unsigned int flags)

将 w 指针指向变量中 flags 对应为 1 的位，设为 0

static inline int tstbit(unsigned int \*w, unsigned int flags)

将 w 指针指向变量中 flags 对应为 1 的位，检查是否为 1 位，并返回所有为 1 位

static inline int tstclr(unsigned int \*w, unsigned int flags)

将 w 指针指向变量中 flags 对应为 1 的位，检查是否为 1 位，清除这些为 1 位，并返回所有为 1 位

* 1. 数据结构

无

* 1. 代码注释

由于代码比较简单，这里不作完整代码注释，仅对其中常使用的两个宏作注解（在include/l4/lib/bit.h中）：

#define BITWISE\_GETWORD(x) ((x) >> WORD\_BITS\_LOG2) /\* Divide by 32 \*/

//GETWORD：x 位中最多有多少个完整的 word

#define BITWISE\_GETBIT(x) (1 << ((x) % WORD\_BITS))

//GETBIT：设置一个 word 中的第 x 位为 1（如果 x >= 32 则取 x mod 32）

* 1. 疑问或改进建议

无

* 1. 体系结构和平台相关的代码

以下宏可能与体系结构有关（ARM 下的 WORD 定义）

BITWISE\_GETWORD

BITWISE\_GETBIT

1. idpool.c
   1. 功能描述（对于复杂操作建议画出流程图）

ID 分配相关函数，用于构建 ID 池来做内存空间 ID 以及线程 ID 的分配：

id\_pool\_new\_init 初始化 id 池

id\_new 在 id 池中寻找新 id

id\_del 在 id 池中删除给定 id

id\_get 在 id 池中获取一个指定的空白 id

具体函数参数：

struct id\_pool \*id\_pool\_new\_init(int mapsize, void \*buffer)

将 freebuf 指向的块初始化为 totalbits 个二进制位（32 位对齐）的 id\_pool

int id\_new(struct id\_pool \*pool)

在 pool 中寻找第一个新空白 id。若有则返回其相对位置（作为 id），若没有则直接 BUG

int id\_del(struct id\_pool \*pool, int id)

在 pool 中检查 id 是否占用（bitmap 第 id 位是否为 1）。若占用则清空并返回 0，否则直接 BUG

int id\_get(struct id\_pool \*pool, int id)

检查并获取一个特定位的空白 id。若 pool 中 id 项空白，则设置占用并返回该 id，否则返回 -1

* 1. 数据结构

//id 池（< 1 \* 4KB 页）

struct id\_pool {

//访问自旋锁

struct spinlock lock;

//nwords 作为控制分配空白 id 的 bitmap 所使用 word 数目

//系统调用时应当注意 <= SYSTEM\_IDS\_MAX

int nwords;

//bitmap 使用 word 数组构建

u32 bitmap[SYSTEM\_IDS\_MAX];

};

//可变 id 池

struct id\_pool\_variable {

struct spinlock lock;

int nwords;

u32 bitmap[];

};

* 1. 代码注释

/\*

\* Used for thread and space ids.

\*

\* Copyright (C) 2007, 2008 Bahadir Balban

\*/

#include <l4/lib/printk.h>

#include <l4/lib/idpool.h>

#include INC\_GLUE(memory.h)

//将 freebuf 指向的块初始化为 totalbits 个二进制位（32 位对齐）的 id\_pool

//注：这里最好 totalbits 作 32 位对齐

struct id\_pool \*id\_pool\_new\_init(int totalbits, void \*freebuf)

{

//注意，这里的 nwords 作为控制分配空白 id 总数（以 word 计）

int nwords = BITWISE\_GETWORD(totalbits);

struct id\_pool \*new = freebuf;

//初始化 lock

spin\_lock\_init(&new->lock);

//初始化 nwords

new->nwords = nwords;

//注意，这里并没有初始化 bitmap！

//所以需要初始化 id\_pool 时，先要初始化 freebuf！

return new;

}

//在 pool 中寻找第一个新空白 id。若有（bitmap 中为 0 位）

//则返回其相对位置（作为 id），若没有则直接 BUG。

int id\_new(struct id\_pool \*pool)

{

int id;

//锁 pool

spin\_lock(&pool->lock);

//找空白块（感觉这个效率很一般 －－，但是最简单）

//nwords \* 32 bits 是限制

id = find\_and\_set\_first\_free\_bit(pool->bitmap,

pool->nwords \* WORD\_BITS);

//解锁

spin\_unlock(&pool->lock);

//无空白块直接报 BUG

//TODO: BUG 是否可跳出？还是直接 kernel fatal error？

BUG\_ON(id < 0);

return id;

}

//在 pool 中检查 id 是否占用（bitmap 第 id 位是否为 1）。若占用则清空并返回 0，否则直接 BUG

int id\_del(struct id\_pool \*pool, int id)

{

int ret;

//加锁

spin\_lock(&pool->lock);

//检查并清空一个有效 id

ret = check\_and\_clear\_bit(pool->bitmap, id);

//解锁

spin\_unlock(&pool->lock);

//TODO: 这里为什么和下面使用不一样的处理方式？

BUG\_ON(ret < 0);

return ret;

}

/\* Return a specific id, if available \*/

//检查并获取一个特定位的空白 id。若 pool 中 id 项空白，则设置占用并返回该 id，否则返回 -1

int id\_get(struct id\_pool \*pool, int id)

{

int ret;

//加锁

spin\_lock(&pool->lock);

//检查并获取

ret = check\_and\_set\_bit(pool->bitmap, id);

//解锁

spin\_unlock(&pool->lock);

if (ret < 0)

//出错

return ret;

else

//正常

return id;

}

* 1. 疑问或改进建议

在include/l4/lib/idpool.h中：

/\* One page size minus the structure fields \*/

//如果要求是在 4KB 大小的一页内的话

//最多 id 数目为何是 1023 \* 32 而不是 1022 \* 32？因为还要加 nwords 和 spinlock 总共两个 32 bits 量，这样总共就超过了 1024 \* 32 bits = 4 KB 了啊

//FIXME: CONFIG\_MAX\_SYSTEM\_IDS 应该是 1022 \* 32？

#define CONFIG\_MAX\_SYSTEM\_IDS (1023\*32)

* 1. 体系结构和平台相关的代码

无

1. memcache.c
   1. 功能描述（对于复杂操作建议画出流程图）

内存缓冲区的建立，主要包括以下功能：

mem\_cache\_bufsize 计算 memcache 结构总共需要空间大小。

mem\_cache\_zalloc 从 cache 里寻找并分配一块空白块，并初始化为0

mem\_cache\_alloc 从 cache 里寻找并分配一块空白块。

mem\_cache\_free 清除给定 memcache 里某一占用的块

mem\_cache\_is\_full 判断 memcache 是否满

mem\_cache\_is\_empty 判断 memcache 是否空

mem\_cache\_is\_last\_free 判断 memcache 是否只剩 1 个

mem\_cache\_total\_empty 返回 memcache 中空白块的数目

具体函数参数：

int mem\_cache\_bufsize(void \*start, int struct\_size, int nstructs, int aligned);

计算 mem\_cache 结构总共需要空间大小。给定开始地址 start，单个空白块大小 struct\_size，总共空白块数 nstructs，是否要求对齐 aligned；返回总共需要的内存大小（字节数）

void \*mem\_cache\_zalloc(struct mem\_cache \*cache);

从 cache 里寻找并分配一块空白块。如果有空白块，用 0 初始化，返回空白块地址；否则返回 0

void \*mem\_cache\_alloc(struct mem\_cache \*cache);

从 cache 里寻找并分配一块空白块。如果有空白块，返回空白块地址；否则返回 0

int mem\_cache\_free(struct mem\_cache \*cache, void \*addr);

清除给定 memcache 里某一占用的块，将 cache 里 addr 地址对应的块释放掉。若释放成功，返回 0；否则返回 -1

struct mem\_cache \*mem\_cache\_init(void \*start, int cache\_size,

int struct\_size, unsigned int alignment);

初始化 memcache。给定开始地址 start，单个空白块大小 struct\_size，总共空白块数 nstructs，是否要求对齐 aligned，按照图示结构初始化 cache->bitmap cache->start cache->end，最后返回已初始化的 mem\_cache 结构体 cache。

static inline int mem\_cache\_is\_full(struct mem\_cache \*cache)

判断 memcache 是否满，若满返回 1，否则返回 0（系统默认逻辑值）

static inline int mem\_cache\_is\_empty(struct mem\_cache \*cache)

判断 memcache 是否空，若空返回 1，否则返回 0（系统默认逻辑值）

static inline int mem\_cache\_is\_last\_free(struct mem\_cache \*cache)

判断 memcache 是否只剩 1 个，若只剩 1 个返回 1，否则返回 0（系统默认逻辑值）

static inline int mem\_cache\_total\_empty(struct mem\_cache \*cache)

返回 memcache 中空白块的数目

* 1. 数据结构

图示结构（struct mem\_cache）：

// mem\_cache 内存结构（没有设置 aligned）

// +---------------------+ <- cache

// | mem\_cache structure |

// +---------------------+ <- cache->bitmap

// | bitmap |

// +---------------------+ <- cache->start

// | | \

// | memory blocks | struct\_size \* cache->total

// | | /

// +---------------------+ <- cache->end

//不可变大小的 mem\_cache bitmap！

struct mem\_cache {

struct link list;//链表，用于构建 memcache 链表

struct mutex mutex;//互斥锁

int total;//总块数

int free;//空白块数

unsigned int start;//连续空白块的起始地址

unsigned int end;//连续空白块的结束地址

unsigned int struct\_size;//单个空白块大小

unsigned int \*bitmap;//bitmap

};

* 1. 代码注释

头文件：

/\*

\* Bitmap-based link-listable fixed-size memory cache.

\*

\* Copyright (C) 2007 Bahadir Balban

\*/

#ifndef \_\_MEMCACHE\_H\_\_

#define \_\_MEMCACHE\_H\_\_

#include <l4/config.h>

#include <l4/macros.h>

#include <l4/types.h>

#include <l4/lib/list.h>

#include <l4/lib/mutex.h>

/\*

\* Very basic cache structure. All it does is, keep an internal bitmap of

\* items of struct\_size. (Note bitmap is fairly efficient and simple for a

\* fixed-size memory cache) Keeps track of free/occupied items within its

\* start/end boundaries. Does not grow/shrink but you can link-list it.

\*/

//不可变大小的 mem\_cache bitmap！

struct mem\_cache {

struct link list;//链表，用于构建 memcache 链表

struct mutex mutex;//互斥锁

int total;//总块数

int free;//空白块数

unsigned int start;//连续空白块的起始地址

unsigned int end;//连续空白块的结束地址

unsigned int struct\_size;//单个空白块大小

unsigned int \*bitmap;//bitmap

};

int mem\_cache\_bufsize(void \*start, int struct\_size, int nstructs, int aligned);

void \*mem\_cache\_zalloc(struct mem\_cache \*cache);

void \*mem\_cache\_alloc(struct mem\_cache \*cache);

int mem\_cache\_free(struct mem\_cache \*cache, void \*addr);

struct mem\_cache \*mem\_cache\_init(void \*start, int cache\_size,

int struct\_size, unsigned int alignment);

//一系列 memcache 状态判断内联函数

//判断 memcache 是否满，若满返回 1，否则返回 0（系统默认逻辑值）

static inline int mem\_cache\_is\_full(struct mem\_cache \*cache)

{

return cache->free == 0;

}

//判断 memcache 是否空，若空返回 1，否则返回 0（系统默认逻辑值）

static inline int mem\_cache\_is\_empty(struct mem\_cache \*cache)

{

return cache->free == cache->total;

}

//判断 memcache 是否只剩 1 个，若只剩 1 个返回 1，否则返回 0（系统默认逻辑值）

static inline int mem\_cache\_is\_last\_free(struct mem\_cache \*cache)

{

return cache->free == 1;

}

//返回 memcache 中空白块的数目

static inline int mem\_cache\_total\_empty(struct mem\_cache \*cache)

{

return cache->free;

}

#endif /\* \_\_MEMCACHE\_H\_\_ \*/

具体代码：

/\*

\* Bitmap-based linked-listable fixed-size memory cache.

\*

\* Copyright (C) 2007 Bahadir Balban

\*/

#include <l4/lib/memcache.h>

#include <l4/lib/string.h>

#include <l4/lib/printk.h>

#include INC\_GLUE(memory.h)

#include <l4/lib/bit.h>

#include <l4/api/errno.h>

/\* Allocate, clear and return element \*/

//从 cache 里寻找并分配一块空白块。如果有空白块，用 0 初始化，返回空白块地址；否则返回 0

void \*mem\_cache\_zalloc(struct mem\_cache \*cache)

{

void \*elem = mem\_cache\_alloc(cache);

//初始化为 0

memset(elem, 0, cache->struct\_size);

return elem;

}

/\* Allocate another element from given @cache. Returns 0 when full. \*/

//从 cache 里寻找并分配一块空白块。如果有空白块，返回空白块地址；否则返回 0

void \*mem\_cache\_alloc(struct mem\_cache \*cache)

{

int bit;

int err;

if (cache->free > 0) {

//加锁

if ((err = mutex\_lock(&cache->mutex)) < 0)

//TODO: Rev err 使用方式

//返回错误

return PTR\_ERR(err); /\* Interruptible mutex \*/

cache->free--;

if ((bit = find\_and\_set\_first\_free\_bit(cache->bitmap,

cache->total)) < 0) {

//这里不完全信任 cache->free，而是还做检查看是否由空白块？还有一个原因，如果在 lock 前两个线程都判定 cache->free > 0

//此时，在分别获取 lock（前后分开）后，可能会发生冲突，所以需要二次检验

//TODO: 可是为什么不先获取互斥锁再考虑 cache->free？是因为效率问题？

//TODO: 在其他程序里也分析一下这种情形

printk("Error: Anomaly in cache occupied state.\n"

"Bitmap full although cache->free > 0\n");

BUG();

}

//释放锁

mutex\_unlock(&cache->mutex);

//返回块指针

return (void \*)(cache->start + (cache->struct\_size \* bit));

} else {

/\* Cache full \*/

return 0;

}

}

/\* Free element at @addr in @cache. Return negative on error. \*/

//清除给定 memcache 里某一占用的块

//将 cache 里 addr 地址对应的块释放掉。若释放成功，返回 0；否则返回 -1

//TODO: Rev mutex/spinlock 锁不锁读？

int mem\_cache\_free(struct mem\_cache \*cache, void \*addr)

{

unsigned int struct\_addr = (unsigned int)addr;

unsigned int bit;

int err = 0;

/\* Check boundary \*/

if (struct\_addr < cache->start || struct\_addr > cache->end)

return -1; /\* Address doesn't belong to cache \*/

//已自动取整

bit = ((struct\_addr - cache->start) / cache->struct\_size);

/\*

\* Check alignment:

\* Find out if there was a lost remainder in last division.

\* There shouldn't have been, because addresses are allocated at

\* struct\_size offsets from cache->start.

\*/

//检测是否整（有更高效算法？）

if (((bit \* cache->struct\_size) + cache->start) != struct\_addr) {

printk("Error: This address is not aligned on a predefined "

"structure address in this cache.\n");

err = -1;

return err;

}

//加锁

if ((err = mutex\_lock(&cache->mutex)) < 0)

return err; /\* Interruptible mutex \*/

//若全正常，由此，返回 0

/\* Check free/occupied state \*/

//检查是否是有效块并清空

if (check\_and\_clear\_bit(cache->bitmap, bit) < 0) {

//若已经空白则跳出

printk("Error: Anomaly in cache occupied state:\n"

"Trying to free already free structure.\n");

err = -1;

goto out;

}

//空白块数量增加

cache->free++;

//空白块数比总块数多

//TODO: 这些诸多约束条件是否需要统一的方式处理？

if (cache->free > cache->total) {

printk("Error: Anomaly in cache occupied state:\n"

"More free elements than total.\n");

err = -1;

goto out;

}

out:

//解锁

mutex\_unlock(&cache->mutex);

return err;

}

/\*

\* Given a buffer start address, structure size, number of

\* structs and alignment requirements, determines how much

\* memory is needed from that starting address

\*/

//计算 mem\_cache 结构总共需要空间大小

//给定开始地址 start，单个空白块大小 struct\_size，总共空白块数 nstructs，是否要求对齐 aligned

//返回总共需要的内存大小（字节数）

/////////////////////////////////////////

//struct mem\_cache {

// struct link list;//链表，用于构建 memcache 链表

// struct mutex mutex;//互斥锁

// int total;//总块数

// int free;//空白块数

// unsigned int start;//连续空白区域的起始地址

// unsigned int end;//连续空白区域的结束地址

// unsigned int struct\_size;//单个空白块大小

// unsigned int \*bitmap;//bitmap

//};

/////////////////////////////////////////

int mem\_cache\_bufsize(void \*start, int struct\_size, int nstructs, int aligned)

{

unsigned long start\_address = (unsigned long)start;

int total\_bytes, bwords, bitmap\_size;

/\* Word alignment requirement \*/

//上取整

start\_address = align\_up(start\_address, sizeof(int));

/\* Total bytes to contain structures \*/

//块占用

//TODO: 这里是不是真实数据块？

total\_bytes = struct\_size \* nstructs;

/\* Total words to contain bitmap \*/

//bitmap 占用 word 数（部分）

bwords = nstructs >> 5;

/\* An extra word if not a multiple of one word's bits \*/

//相当于上取整

if (nstructs & 0x1F)

bwords++;

/\* Total bitmap bytes \*/

//bitmap 占用大小

bitmap\_size = bwords \* sizeof(int);

/\* Current would-be start address \*/

//TODO: 还是叫 start\_address 么？

//主要占用部分：memcache 结构体 ＋ bitmap ＋ 空白块 （＋边角余料未对齐）

start\_address += bitmap\_size + total\_bytes + sizeof(struct mem\_cache);

/\* Check alignment requirement \*/

//对齐（是指都按 struct\_size 对齐）

if (aligned)

start\_address = align\_up(start\_address, struct\_size);

return start\_address - (unsigned long)start;

}

// memcache 结构（没有设置 aligned）

// +---------------------+ <- cache

// | mem\_cache structure |

// +---------------------+ <- cache->bitmap

// | bitmap |

// +---------------------+ <- cache->start

// | | \

// | memory blocks | struct\_size \* cache->total

// | | /

// +---------------------+ <- cache->end

//初始化 memcache

//给定开始地址 start，单个空白块大小 struct\_size，总共空白块数 nstructs，是否要求对齐 aligned

//按照图示结构初始化 cache->bitmap cache->start cache->end

//最后返回已初始化的 mem\_cache 结构体 cache

struct mem\_cache \*mem\_cache\_init(void \*bufstart, int cache\_size,

int struct\_size, unsigned int aligned)

{

void \*start;

struct mem\_cache \*cache;

unsigned int area\_start;

unsigned int \*bitmap;

int bwords, total, bsize;

/\* Align to nearest word boundary \*/

//word 对齐

start = (void \*)align\_up(bufstart, sizeof(int));

//对齐后 memcache 的起始地址和大小

cache\_size -= (int)start - (int)bufstart;

cache = start;

if ((struct\_size < 0) || (cache\_size < 0) ||

((unsigned long)start == ~(0))) {

printk("Invalid parameters.\n");

return 0;

}

/\*

\* The cache definition itself is at the beginning.

\* Skipping it to get to start of free memory. i.e. the cache.

\*/

//加上 memcache 结构的大小

area\_start = (unsigned long)start + sizeof(struct mem\_cache);

//剩下的是 bitmap ＋ 空白块

cache\_size -= sizeof(struct mem\_cache);

if (cache\_size < struct\_size) {

printk("Cache too small for given struct\_size\n");

return 0;

}

//获取 bitmap 占用空间

/\* Get how much bitmap words occupy \*/

total = cache\_size / struct\_size;

//以 word 计

bwords = total >> 5; /\* Divide by 32 \*/

//上取整

if (total & 0x1F) { /\* Remainder? \*/

bwords++; /\* Add one more word for remainder \*/

}

//占用字节数

bsize = bwords \* 4;

/\* Reduce bitmap bytes from cache size \*/

//剩下的是空白块

cache\_size -= bsize;

/\* Recalculate total - it may or may not have changed \*/

//只会比 bitmap 可以索引的部分数量要少而不会多

total = cache\_size / struct\_size;

/\* This should always catch too small caches \*/

//TODO: 为什么要两次处理？

if (total <= 0) {

printk("Cache too small for given struct\_size\n");

return 0;

}

if (cache\_size <= 0) {

printk("Cache too small for given struct\_size\n");

return 0;

}

//bitmap 开始地址

bitmap = (unsigned int \*)area\_start;

//TODO: 这里为什么要加 bwords？为什么不是 bsize？

area\_start = (unsigned int)(bitmap + bwords);

//要求按 struct\_size 对齐

if (aligned) {

//TODO: 可以这样定义么？

unsigned int addr = area\_start;

unsigned int addr\_aligned = align\_up(area\_start, struct\_size);

unsigned int diff = addr\_aligned - addr;

//为什么这里设置 BUG\_ON?

BUG\_ON(diff >= struct\_size);

//再减去向上取整时的差值

cache\_size -= diff;

/\* Recalculate total \*/

//重新计算空白块数

total = cache\_size / struct\_size;

area\_start = addr\_aligned;

}

//初始化 memcache 数据结构

/////////////////////////////////////////////////

link\_init(&cache->list);

cache->start = area\_start;

cache->end = area\_start + cache\_size;

cache->total = total;

cache->free = cache->total;

cache->struct\_size = struct\_size;

cache->bitmap = bitmap;

//初始化 mutex

mutex\_init(&cache->mutex);

//bitmap 初始化为 0

memset(cache->bitmap, 0, bwords\*SZ\_WORD);

/////////////////////////////////////////////////

return cache;

}

* 1. 疑问或改进建议

无

* 1. 体系结构和平台相关的代码

无

1. mutex.c
   1. 功能描述（对于复杂操作建议画出流程图）

互斥量（二进制信号量）的相关操作，具体功能如下：

mutex\_trylock 非阻塞加锁

mutex\_lock 阻塞加锁

mutex\_unlock 同步解锁

mutex\_unlock\_async 异步解锁

具体函数参数如下：

int mutex\_trylock(struct mutex \*mutex);

尝试加锁互斥量 mutex，即非阻塞式，锁住 mutex 成功返回 1，失败返回 0

int mutex\_lock(struct mutex \*mutex);

阻塞方式加锁，成功加锁 mutex 后返回 0，加锁失败则线程进入 sleep 队列并等待，若出现 TASK\_INTERRUPTED 的唤醒则直接返回错误 -EINTR

void mutex\_unlock(struct mutex \*mutex);

同步解锁 mutex

void mutex\_unlock\_async(struct mutex \*mutex);

异步解锁 mutex

* 1. 数据结构

互斥量构造

struct mutex {

//等待队列（访问之还需先获得自旋锁）

struct waitqueue\_head wqh;

//锁（互斥锁的主体）

unsigned int lock;

};

注意，在代码中锁 mutex 前都需要将等待队列先锁住，避免多个线程同时修改而引发问题

附注：waitqueue 链表头构造

struct waitqueue\_head {

//睡眠任务数量

int sleepers;

//自旋锁

struct spinlock slock;

//任务列表的链表

struct link task\_list;

};

* 1. 代码注释

/\*

\* Mutex/Semaphore implementations.

\*

\* Copyright (c) 2007 Bahadir Balban

\*/

#include <l4/lib/mutex.h>

#include <l4/generic/scheduler.h>

#include <l4/generic/tcb.h>

#include <l4/api/errno.h>

/\*

\* Semaphore usage:

\*

\* Producer locks/produces/unlocks data.

\* Producer does semaphore up.

\* --

\* Consumer does semaphore down.

\* Consumer locks/consumes/unlocks data.

\*/

//信号量处理

//以下代码之后分析

#if 0

/\* Update it \*/

/\*

\* Semaphore \*up\* for multiple producers. If any consumer is waiting, wake them

\* up, otherwise, sleep. Effectively producers and consumers use the same

\* waitqueue and there's only one kind in the queue at any one time.

\*/

void sem\_up(struct mutex \*mutex)

{

int cnt;

spin\_lock(&mutex->slock);

if ((cnt = mutex\_inc(&mutex->lock)) <= 0) {

struct waitqueue \*wq;

struct ktcb \*sleeper;

/\* Each producer wakes one consumer in queue. \*/

mutex->sleepers--;

BUG\_ON(list\_empty(&mutex->wq.task\_list));

list\_foreach\_struct(wq, &mutex->wq.task\_list, task\_list) {

list\_remove\_init(&wq->task\_list);

spin\_unlock(&mutex->slock);

sleeper = wq->task;

printk("(%d) Waking up consumer (%d)\n", current->tid,

sleeper->tid);

sched\_resume\_task(sleeper);

return; /\* Don't iterate, wake only one task. \*/

}

} else if (cnt > 0) {

DECLARE\_WAITQUEUE(wq, current);

link\_init(&wq.task\_list);

list\_insert\_tail(&wq.task\_list, &mutex->wq.task\_list);

mutex->sleepers++;

sched\_prepare\_sleep();

printk("(%d) produced, now sleeping...\n", current->tid);

spin\_unlock(&mutex->slock);

schedule();

}

}

/\*

\* Semaphore \*down\* for multiple consumers. If any producer is sleeping, wake them

\* up, otherwise, sleep. Effectively producers and consumers use the same

\* waitqueue and there's only one kind in the queue at any one time.

\*/

void sem\_down(struct mutex \*mutex)

{

int cnt;

spin\_lock(&mutex->slock);

if ((cnt = mutex\_dec(&mutex->lock)) >= 0) {

struct waitqueue \*wq;

struct ktcb \*sleeper;

/\* Each consumer wakes one producer in queue. \*/

mutex->sleepers--;

BUG\_ON(list\_empty(&mutex->wq.task\_list));

list\_foreach\_struct(wq, &mutex->wq.task\_list, task\_list) {

list\_remove\_init(&wq->task\_list);

spin\_unlock(&mutex->slock);

sleeper = wq->task;

printk("(%d) Waking up producer (%d)\n", current->tid,

sleeper->tid);

sched\_resume\_task(sleeper);

return; /\* Don't iterate, wake only one task. \*/

}

} else if (cnt < 0) {

DECLARE\_WAITQUEUE(wq, current);

link\_init(&wq.task\_list);

list\_insert\_tail(&wq.task\_list, &mutex->wq.task\_list);

mutex->sleepers++;

sched\_prepare\_sleep();

printk("(%d) Waiting to consume, now sleeping...\n", current->tid);

spin\_unlock(&mutex->slock);

schedule();

}

}

#endif

/\* Non-blocking attempt to lock mutex \*/

// 非阻塞方式加锁

// 锁住 mutex 成功返回 1，失败返回 0

int mutex\_trylock(struct mutex \*mutex)

{

int success;

//TODO: Rev 这里为什么要加自旋锁？是为了阻止 wait/wake 相关的函数？

spin\_lock(&mutex->wqh.slock);

/////////////////////////////////////

if ((success = \_\_mutex\_lock(&mutex->lock)))

//TODO: Rev 锁的数量的用途

current->nlocks++;

/////////////////////////////////////

spin\_unlock(&mutex->wqh.slock);

return success;

}

// 阻塞方式加锁

// 成功加锁 mutex 后返回 0，加锁失败则线程进入 sleep 队列并等待

// 若出现 TASK\_INTERRUPTED 的唤醒则直接返回错误 -EINTR

int mutex\_lock(struct mutex \*mutex)

{

/\* NOTE:

\* Everytime we're woken up we retry acquiring the mutex. It is

\* undeterministic as to how many retries will result in success.

\* We may need to add priority-based locking.

\*/

//未来加入优先级是为了保证一定在有限时间内可以获得锁

for (;;) {

//自旋锁锁住访问 mutex （短时、临时）

spin\_lock(&mutex->wqh.slock);

///////////////////////////////////////////////

//检查互斥量 lock

if (!\_\_mutex\_lock(&mutex->lock)) { /\* Could not lock, sleep. \*/

//快速创建／初始化 waitqueue：wq

CREATE\_WAITQUEUE\_ON\_STACK(wq, current);

//一套和 wait 类似的 sleep 过程

task\_set\_wqh(current, &mutex->wqh, &wq);

list\_insert\_tail(&wq.task\_list, &mutex->wqh.task\_list);

mutex->wqh.sleepers++;

// 睡眠

sched\_prepare\_sleep();

///////////////////////////////////////////

// 解锁自旋锁

spin\_unlock(&mutex->wqh.slock);

// printk("(%d) sleeping...\n", current->tid);

// 调度

schedule();

/\* Did we wake up normally or get interrupted \*/

if (current->flags & TASK\_INTERRUPTED) {

current->flags &= ~TASK\_INTERRUPTED;

// 中断方式进入则直接跳出

// 注意这里“中断”的含义

return -EINTR;

}

} else {

//若成功

current->nlocks++;

break;

}

}

///////////////////////////////////////////////////

// 这里是成功获取互斥量之后的自旋锁解锁

spin\_unlock(&mutex->wqh.slock);

return 0;

}

// 公共（同步／异步）的 mutex 解锁

// 解锁 mutex，其中 sync 为 0 时为异步解锁，非 0 时同步解锁

static inline void mutex\_unlock\_common(struct mutex \*mutex, int sync)

{

//TODO: 这步检测的作用？

struct ktcb \*c = current; if (c);

spin\_lock(&mutex->wqh.slock);

///////////////////////////////////////////////////

\_\_mutex\_unlock(&mutex->lock);

current->nlocks--;

//BUG trace

//TODO: Rev OO 中不变量

BUG\_ON(current->nlocks < 0);

BUG\_ON(mutex->wqh.sleepers < 0);

if (mutex->wqh.sleepers > 0) {

//TODO: 对比 wake 相关

struct waitqueue \*wq = link\_to\_struct(mutex->wqh.task\_list.next,

struct waitqueue,

task\_list);

struct ktcb \*sleeper = wq->task;

task\_unset\_wqh(sleeper);

BUG\_ON(list\_empty(&mutex->wqh.task\_list));

list\_remove\_init(&wq->task\_list);

mutex->wqh.sleepers--;

///////////////////////////////////////////////

spin\_unlock(&mutex->wqh.slock);

/\*

\* TODO:

\* Here someone could grab the mutex, this is fine

\* but it may potentially starve the sleeper causing

\* non-determinism. We may consider priorities here.

\*/

//和之前一样，可能会有饥饿状态，未来添加优先级

if (sync)

sched\_resume\_sync(sleeper);

else

sched\_resume\_async(sleeper);

/\* Don't iterate, wake only one task. \*/

//与 wake 的不同，为了避免竞争锁的消耗？

return;

}

spin\_unlock(&mutex->wqh.slock);

}

//TODO: 区分同步异步与阻塞非阻塞之间的区别

// 同步解锁（直接调度）

void mutex\_unlock(struct mutex \*mutex)

{

mutex\_unlock\_common(mutex, 1);

}

//异步解锁（等待调度）

void mutex\_unlock\_async(struct mutex \*mutex)

{

mutex\_unlock\_common(mutex, 0);

}

* 1. 疑问或改进建议

TASK\_INTERRUPTED 这个进程状态还没能好好分析

以及

struct ktcb \*c = current; if (c);

这步检查的作用也不是特别清楚

* 1. 体系结构和平台相关的代码

\_\_mutex\_lock

这个函数的具体定义在 arch 里，根据 ARM 版本号不同而采取了不同的机制

1. printk.c
   1. 功能描述（对于复杂操作建议画出流程图）

格式化输出相关函数，具体功能如下：

printk 输出指定格式字符串

注意这里的printk只实现了传统printf的一个功能子集，并添加了部分特殊功能，具体参数见后面分析。

实际上这份代码文件里有一些辅助函数，不过大都是为 printk 服务，主要功能还是由 printk 体现。

具体函数参数：

int printk(char \*format, ...)

和传统的printf类似，printk的参数由格式字符串指针format以及后面的可变参数表构成，其中格式字符串满足以下选项

% 特殊选项开始（而双写符号即 %% 则表示将其转义成普通“%”输出）

其格式满足%[prefix][type] 其中prefix表示前缀的附加选项，type表示数据类型

其中 type 有以下几种（注意，其中d和u的使用时，prefix中只有宽度参数有效）

c 单个字符

m 64位变量的十六进制值

d 整数

u 无符号整数

x 十六进制数

s 字符串

p 精度为word十六进制长度输出十六进制数

（即精度为16位十六进制数，以下统称为标准长度，常用于内存地址、32位数据等的显示）

而 prefix 有以下几种，可以看做是[-][width][w][.[precision][w]]

其中 ‘-’ 表示右对齐

width 是一个十进制数字，表示输出宽度，有前导0（即数字以0开头）时设置使用‘0’填充空余占位字符，没有前导0时使用空格‘ ’填充空余占位字符

w(第一个) 表示设定宽度为标准长度

‘.’ 表示之后还设定精度

precision 是一个十进制数字，表示输出精度

w(第二个) 表示设定精度为标准长度

注：精度在这里十六进制数的输出中定义为数据的最小输出长度，若不足则添加前导‘0’，相当于右对齐加前导0加指定宽度的效果。

示例代码（使用修正后的函数）：

printk("test1 %d\n",255);

printk("test2 %p\n",255);

printk("test3 #%6.5x#\n",255);

printk("test4 #%-6.5x#\n",255);

printk("test5 #%6w.5x#\n",255);

printk("test6 #%6.5wx#\n",255);

printk("test7 #%06.5x#\n",255);

printk("test8 #%6x#\n",255);

printk("test9 #%-6x#",255);

输出结果：

test1 255

test2 000000ff

test3 # 000ff#

test4 #000ff #

test5 # 000ff#

test6 #000000ff#

test7 #0000ff#

test8 # ff#

test9 #ff #

* 1. 代码注释

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Copyright (C) 2002-2004 Karlsruhe University

\*

\* File path: generic/printk.cc

\* Description: Implementation of printf

\*

\* Redistribution and use in source and binary forms, with or without

\* modification, are permitted provided that the following conditions

\* are met:

\* 1. Redistributions of source code must retain the above copyright

\* notice, this list of conditions and the following disclaimer.

\* 2. Redistributions in binary form must reproduce the above copyright

\* notice, this list of conditions and the following disclaimer in the

\* documentation and/or other materials provided with the distribution.

\*

\* THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND

\* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE

\* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE

\* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE

\* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL

\* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS

\* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)

\* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT

\* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY

\* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF

\* SUCH DAMAGE.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdarg.h> /\* for va\_list, ... comes with gcc \*/

#include <l4/lib/printk.h>

#include <l4/lib/mutex.h>

/\* FIXME: LICENSE LICENCE \*/

typedef unsigned int word\_t;

/\* 输出单字符 \*/

extern void putc(const char c);

extern int print\_tid (word\_t val, word\_t width, word\_t precision, int adjleft);

/\* convert nibble to lowercase hex char \*/

// 整数转十六进制字符

#define hexchars(x) (((x) < 10) ? ('0' + (x)) : ('a' + ((x) - 10)))

/\*\*

\* Print hexadecimal value

\*

\* @param val value to print

\* @param width width in characters

\* @param precision minimum number of digits to appear

\* @param adjleft left adjust the value

\* @param nullpad pad with leading zeros (when right padding)

\*

\* Prints a hexadecimal value with leading zeroes of given width

\* using putc(), or if adjleft argument is given, print

\* hexadecimal value with space padding to the right.

\*

\* @returns the number of characters printed (should be same as width).

\*/

/\* TODO: 学习 @ 在注释中的使用方法 \*/

/\* TODO: 复习面向对象课上学习的注释方法 \*/

/\*

\* 输出一个 64 位数的十六进制数

\*

\* @param val 要输出的 64 位数

\* @param width 指定的输出宽度

\* @param precision 输出精度，在十六进制输出里指最小输出宽度

\* @param adjleft 指定左对齐还是右对齐，非零指左对齐，零指右对齐

\* @param nullpad 指定输出宽度空余位是否填 '0'（仅仅用于左对齐时），非零指填 '0'，零指空格 ' '

\*

\* 使用 putc() 输出一个给定宽度、精度、空余位填法、对其方向的 64 位数的十六进制格式字符串

\*

\* @returns 返回真实输出的字符个数

\*/

int print\_hex64(u64 val, int width, int precision, int adjleft, int nullpad)

{

// n 是输出字符总数，即 putc 调用次数

int i, n = 0;

//原始数据十六进制下有效宽度

int nwidth = 0;

//高 32 位、低 32 位数据

u32 high, low;

//获取高 32 位、低 32 位

high = val >> 32;

low = (u32)val;

// Find width of hexnumber

if (high) {

// 高位存在

// 为何有 ((unsigned) nwidth < 2 \* sizeof (u32)) 这个判断标准？有用否？偶尔有用！\

因为一个字节占用两个十六进制位，所以 high 或 low 最多应该不超过 2 \* sizeof(u32) 个十六进制位

while ((high >> (4 \* nwidth)) && ((unsigned) nwidth < 2 \* sizeof (u32)))

nwidth++;

//TODO: 这里为什么要加 32？是为了说明存在高位？这样从 nwidth 第五位的有无就可以判断是否存在高位（high），\

且由于高位十六进制位不超过 2 \* sizeof(u32) = 8 从而不会造成影响？

nwidth += 32;

} else {

// 只有低位

while ((low >> (4 \* nwidth)) && ((unsigned) nwidth < 2 \* sizeof (u32)))

nwidth++;

}

// 十六进制宽度为 0 时，自动变为 1，从而输出 0

if (nwidth == 0)

nwidth = 1;

// May need to increase number of printed digits

// 由于这里是十六进制数，这里 precision 指最小输出宽度，从后面处理来看，多的位数就会补 '0' 了

if (precision > nwidth)

nwidth = precision;

// May need to increase number of printed characters

// 至少是 nwidth 宽

// TODO: 这里代码是不是有问题？应该是 (width == 0 || width < nwidth) 吧？

if (width == 0 && width < nwidth)

width = nwidth;

// Print number with padding

if (high)

{

// 输出高位

if (!adjleft)

// 右对齐，可以自定义左边 padding

// TODO: 这里可是含有 32 个十六进制位的高位啊，怎么都感觉是写错了！因为这里可是 width - nwidth 直接减啊 \

所有低位也没有 32 位啊，之前应该是加上 2 \* sizeof(u32) = 8 才对吧

for (i = width - nwidth; i > 0; i--, n++)

// 输出 padding，可以自定义

putc (nullpad ? '0' : ' ');

// 输出高位

for (i = 4 \* (nwidth - 33); i >= 0; i -= 4, n++)

putc (hexchars ((high >> i) & 0xF));

if (adjleft)

// 左对齐，右边默认 padding 为空格

// TODO: 这里不对吧，为什么是在这里加 padding？不应当等低位输出完了之后再加 padding 么？

for (i = width - nwidth; i > 0; i--, n++)

putc (' ');

// 分别减去之前加上的 32

width -= 32;

nwidth -= 32;

// 注意，这里一定记得 nullpad = 1，使低位的 padding 是 0，从而构成完整的数字

nullpad = 1;

}

if (! adjleft)

// 右对齐，可以自定义左边 padding

for (i = width - nwidth; i > 0; i--, n++)

putc (nullpad ? '0' : ' ');

// 输出低位

for (i = 4 \* (nwidth - 1); i >= 0; i -= 4, n++)

putc (hexchars ((low >> i) & 0xF));

if (adjleft)

// 左对齐，右边默认 padding 为空格

for (i = width - nwidth; i > 0; i--, n++)

putc (' ');

//返回输出字符总数

return n;

}

int print\_hex\_3arg(const word\_t val, int width, int precision)

{

long i, n = 0;

long nwidth = 0;

int adjleft = 0;

int nullpad = 0;

// Find width of hexnumber

while ((val >> (4 \* nwidth)) && (word\_t) nwidth < 2 \* sizeof (word\_t))

nwidth++;

if (nwidth == 0)

nwidth = 1;

// May need to increase number of printed digits

if (precision > nwidth)

nwidth = precision;

// May need to increase number of printed characters

if (width == 0 && width < nwidth)

width = nwidth;

// Print number with padding

// 右对齐时加入左边 padding，可以自定义

if (! adjleft)

for (i = width - nwidth; i > 0; i--, n++)

//nullpad 只要非空就设置 '0' 作为 padding

putc (nullpad ? '0' : ' ');

for (i = 4 \* (nwidth - 1); i >= 0; i -= 4, n++)

//输出一个十六进制值

putc (hexchars ((val >> i) & 0xF));

//左对齐时加入右边 padding，仅使用空格

if (adjleft)

for (i = width - nwidth; i > 0; i--, n++)

putc (' ');

return n;

}

int print\_hex\_5arg(const word\_t val, int width,

int precision, int adjleft, int nullpad)

{

long i, n = 0;

long nwidth = 0;

// Find width of hexnumber

while ((val >> (4 \* nwidth)) && (word\_t) nwidth < 2 \* sizeof (word\_t))

nwidth++;

if (nwidth == 0)

nwidth = 1;

// May need to increase number of printed digits

if (precision > nwidth)

nwidth = precision;

// May need to increase number of printed characters

if (width == 0 && width < nwidth)

width = nwidth;

// Print number with padding

if (! adjleft)

for (i = width - nwidth; i > 0; i--, n++)

putc (nullpad ? '0' : ' ');

for (i = 4 \* (nwidth - 1); i >= 0; i -= 4, n++)

putc (hexchars ((val >> i) & 0xF));

if (adjleft)

for (i = width - nwidth; i > 0; i--, n++)

putc (' ');

return n;

}

/\*\*

\* Print a string

\*

\* @param s zero-terminated string to print

\* @param width minimum width of printed string

\*

\* Prints the zero-terminated string using putc(). The printed

\* string will be right padded with space to so that it will be

\* at least WIDTH characters wide.

\*

\* @returns the number of charaters printed.

\*/

int print\_string\_3arg(const char \* s, const int width, const int precision)

{

int n = 0;

for (;;)

{

if (\*s == 0)

break;

putc(\*s++);

n++;

if (precision && n >= precision)

break;

}

while (n < width) { putc(' '); n++; }

return n;

}

int print\_string\_1arg(const char \* s)

{

int n = 0;

int width = 0;

int precision = 0;

for (;;) {

if (\*s == 0)

break;

putc(\*s++);

n++;

if (precision && n >= precision)

break;

}

while (n < width) {

putc(' ');

n++;

}

return n;

}

/\*\*

\* Print hexadecimal value with a separator

\*

\* @param val value to print

\* @param bits number of lower-most bits before which to

\* place the separator

\* @param sep the separator to print

\*

\* @returns the number of charaters printed.

\*/

int print\_hex\_sep(const word\_t val, const int bits, const char \*sep)

{

int n = 0;

n = print\_hex\_3arg(val >> bits, 0, 0);

n += print\_string\_1arg(sep);

n += print\_hex\_3arg(val & ((1 << bits) - 1), 0, 0);

return n;

}

/\*\*

\* Print decimal value

\*

\* @param val value to print

\* @param width width of field

\* @param pad character used for padding value up to width

\*

\* Prints a value as a decimal in the given WIDTH with leading

\* whitespaces.

\*

\* @returns the number of characters printed (may be more than WIDTH)

\*/

int print\_dec(const word\_t val, int width)

{

word\_t divisor;

int digits;

/\* estimate number of spaces and digits \*/

for (divisor = 1, digits = 1; val/divisor >= 10; divisor \*= 10, digits++);

/\* print spaces \*/

for ( ; digits < width; digits++ )

putc(' ');

/\* print digits \*/

do {

putc(((val/divisor) % 10) + '0');

} while (divisor /= 10);

/\* report number of digits printed \*/

return digits;

}

/\*\*

\* Does the real printk work

\*

\* @param format\_p pointer to format string

\* @param args list of arguments, variable length

\*

\* Prints the given arguments as specified by the format string.

\* Implements a subset of the well-known printf plus some L4-specifics.

\*

\* @returns the number of characters printed

\*/

/\*

\* 格式化输出函数实体

\*

\* @param format\_p 格式字符串指针

\* @param args 可变参数表

\*

\* 输出给定格式字符串样式的字符

\* 这里实现了传统 printf 的一个功能子集，并附加了一些 L4 内核的特定功能

\*

\* @returns 返回输出的字符串个数

\*/

int do\_printk(char\* format\_p, va\_list args)

{

const char\* format = format\_p;

// 输出字符串总数

int n = 0;

int i = 0;

// 默认宽度 8

int width = 8;

// 默认精度 0

int precision = 0;

// 默认右对齐，用 ' ' 填充

int adjleft = 0, nullpad = 0;

// 这些参数到后面会重新初始化

// TODO: 检查这些参数与后面参数是否有冲突

// arg 宏是用来简单获取指定类型的当前可变参数的

#define arg(x) va\_arg(args, x)

/\* sanity check \*/

// 若格式字符串为空，直接结束

if (format == '\0')

{

return 0;

}

// 当格式字符串非空

while (\*format)

{

switch (\*(format))

{

// 出现 '%' 时

case '%':

// 初始化

// 这一步非常重要

width = precision = 0;

adjleft = nullpad = 0;

reentry:

switch (\*(++format))

{

/\* modifiers \*/

// 一些前缀选项

// ".数字" 表示精度 precision

case '.':

for (format++; \*format >= '0' && \*format <= '9'; format++)

precision = precision \* 10 + (\*format) - '0';

// ".w" 表示精度为 word 的十六进制长度

if (\*format == 'w')

{

// Set precision to printsize of a hex word

precision = sizeof (word\_t) \* 2;

format++;

}

// 恢复到上次读取的最后一个字符，方便 switch 中自加

format--;

goto reentry;

// 当前导 '0' 即在 width == 0 时出现 0，表示输出宽度空余位设置为零

case '0':

nullpad = (width == 0);

// 注意这种用法 ...

case '1'...'9':

//这里实际上也包含了对数字中出现 '0' 的处理

width = width\*10 + (\*format)-'0';

goto reentry;

// 'w' 表示宽度为 word 的十六进制长度

case 'w':

// Set width to printsize of a hex word

width = sizeof (word\_t) \* 2;

goto reentry;

// '-' 表示左对齐，这里写错了！

case '-':

// 以下注释掉的是原来错误的代码

// adjleft = 0;

adjleft = 1;

goto reentry;

// 'l' 本来表示长型，这里忽略

case 'l':

goto reentry;

break;

// 一些主体选项

// 'c' 表示单个字符，这里直接输出

case 'c':

// 注意，这里使用的是 int 跟对齐有关

// 详情参看变长参数表使用相关说明

putc(arg(int));

// 输出字符数自加

n++;

break;

// 'm' 表示输出给定 64 位数值

case 'm': /\* microseconds \*/

{

n += print\_hex64(arg(u64), width, precision,

adjleft, nullpad);

break;

}

// 'd' 表示输出整数，默认为长整型

case 'd':

{

long val = arg(long);

// 负数直接输出负号

if (val < 0)

{

// 这里没有作对齐的判断，且之后调用 print\_dec 也没有传相关参数，似乎忽略了很多

putc('-');

val = -val;

}

// 输出整数，这里传出的是正整数

n += print\_dec(val, width);

break;

}

// 'u' 表示输出无符号整数

case 'u':

n += print\_dec(arg(long), width);

break;

// 'p' 表示精度为 word 的十六进制长度（与下面 'x' 是有一定程度的公用部分的）

case 'p':

precision = sizeof (word\_t) \* 2;

// 'x' 表示输出十六进制数

case 'x':

n += print\_hex\_5arg(arg(long), width, precision, adjleft, nullpad);

break;

// 's' 表示输出字符串

case 's':

{

char\* s = arg(char\*);

//注意，这里当字符串指针为空时会显式输出 "(null)" 可用于容错

if (s)

n += print\_string\_3arg(s, width, precision);

else

n += print\_string\_3arg("(null)", width, precision);

}

break;

// 't' 暂时未使用，可以用于显示线程 ID

// TODO: print\_tid() 的作用

case 't':

case 'T':

// Do nothing for now.

//n += print\_tid (arg (word\_t), width, precision, adjleft);

break;

// '%' 即构成 "%%" 或其他特殊形式，此时取消转义

case '%':

putc('%');

n++;

// 直接跳出 '%' 处理过程！

format++;

continue;

default:

// 输出参数有问题

n += print\_string\_1arg("?");

break;

};

// 似乎是统计有效 '%' 个数，即有效的可变参数个数

i++;

break;

default:

// 非 % 循环内，默认直接输出

putc(\*format);

n++;

break;

}

format++;

}

// 返回输出的字符串个数

return n;

}

// 自旋锁

DECLARE\_SPINLOCK(printk\_lock);

/\*\*

\* Flexible print function

\*

\* @param format string containing formatting and parameter type

\* information

\* @param ... variable list of parameters

\*

\* @returns the number of characters printed

\*/

/\*

\* 格式化输出函数

\*

\* @param format 格式化字符串指针

\* @param ... 可变参数表

\*

\* @returns 返回输出的字符串个数

\*/

int printk(char \*format, ...)

{

// 可变参数表项

va\_list args;

int i;

unsigned long irqstate;

// 获取第一个可变参数

va\_start(args, format);

// 锁住

// 注意，这里连中断也锁住了，为了输出的连贯性

// TODO: 复习是否锁中断会锁调度

spin\_lock\_irq(&printk\_lock, &irqstate);

// 解析与输出

i = do\_printk(format, args);

// 解锁

spin\_unlock\_irq(&printk\_lock, irqstate);

// 重新初始化

va\_end(args);

return i;

}

* 1. 疑问或改进建议

do\_printk() 中以下代码处有误（注释中为原来代码，下面为修正代码）

case '-':

// 以下注释掉的是原来错误的代码

// adjleft = 0;

adjleft = 1;

goto reentry;

这里应当设置 adjleft 为非0值，因为根据 POSIX 标准以及后面代码的处理，’-’应当表示左对齐。

而且感觉这份格式化输出代码有些拼凑，冗余代码有些多，可以作适当优化。

* 1. 体系结构和平台相关的代码

无

1. putc.c
   1. 功能描述（对于复杂操作建议画出流程图）

输出一个单字符，具体功能如下：

putc 输出单字符

具体函数参数：

void putc(char c) 输出单字符 c

这个函数非常简单，不过要注意的是其中包含了对换行符的处理（跨硬件平台）

* 1. 数据结构

无

* 1. 代码注释

/\*

\* Generic putc implementation that ties with platform-specific uart driver.

\*

\* Copyright (C) 2007 Bahadir Balban

\*/

#include INC\_PLAT(uart.h)

void putc(char c)

{

//换行符转换

if (c == '\n')

uart\_tx\_char(PLATFORM\_CONSOLE\_VBASE, '\r');

//串口输出

uart\_tx\_char(PLATFORM\_CONSOLE\_VBASE, c);

}

* 1. 疑问或改进建议

可以讲换行符更换作为一个内核选项单独拿出来

* 1. 体系结构和平台相关的代码

uart\_tx\_char(PLATFORM\_CONSOLE\_VBASE, '\r');

uart\_tx\_char(PLATFORM\_CONSOLE\_VBASE, c);

这两者都是根据UART端口位置来决定的

1. string.c
   1. 功能描述（对于复杂操作建议画出流程图）

一些基本的字符串操作，具体功能如下：

memset 写入一块内存区域所有字节为某值

memcpy 拷贝一块内存区域到指定位置

strcmp 比较两个字符串大小

strncpy 拷贝字符串前部分字节

具体函数参数如下：

void \*memset(void \*p, int c, int size)

将指针 p 所指前 size 字节写为 c

void \*memcpy(void \*d, void \*s, int size)

将指针 s 所指前 size 字节内容拷贝到指针 d

int strcmp(const char \*s1, const char \*s2)

比较字符串大小，s1 < s2 返回负值，s1 = s2 返回 0，s1 > s2 返回正值

char \*strncpy(char \*to, const char \*from, int count)

从指针 from 拷贝 count 字节到指针 to，其中若 from 到结尾，则剩下均拷贝 0，返回指针 to

* 1. 数据结构

无

* 1. 代码注释

/\*

\* Copyright (C) 2008-2010 B Labs Ltd.

\*/

//汇编优化代码

void \*\_memset(void \*p, int c, int size);

void \*\_memcpy(void \*d, void \*s, int size);

//将 p 所指前 size 字节内容写为 c？

void \*memset(void \*p, int c, int size)

{

return \_memset(p, c, size);

}

//将 s 所指前 size 字节内容拷贝到 d

void \*memcpy(void \*d, void \*s, int size)

{

return \_memcpy(d, s, size);

}

//比较字符串大小，s1 < s2 返回负值，s1 = s2 返回 0，s1 > s2 返回正值

int strcmp(const char \*s1, const char \*s2)

{

unsigned int i = 0;

int d;

while(1) {

d = (unsigned char)s1[i] - (unsigned char)s2[i];

//注意，这里只需要写一个 s1[i] == '\0'（或 s2[i] == '\0'）即可

//因为要么两个字符串同时到 '\0' 要么在两个字符串结尾前一定不同！

if (d != 0 || s1[i] == '\0')

return d;

i++;

}

}

/\*

\* Copies string pointed by @from to string pointed by @to.

\*

\* If count is greater than the length of string in @from,

\* pads rest of the locations with null.

\*/

//从 from 拷贝 count 字节到 to，其中若 from 到结尾，则剩下均拷贝 0

char \*strncpy(char \*to, const char \*from, int count)

{

char \*temp = to;

while (count) {

\*temp = \*from;

/\*

\* Stop updating from if null

\* terminator is reached.

\*/

//相当于，from 结尾后还有 count 时，对应 to 全置 0

if (\*from)

from++;

temp++;

count--;

}

return to;

}

* 1. 疑问或改进建议

无

* 1. 体系结构和平台相关的代码

//汇编优化代码

void \*\_memset(void \*p, int c, int size);

void \*\_memcpy(void \*d, void \*s, int size);

1. wait.c
   1. 功能描述（对于复杂操作建议画出流程图）

线程等待与唤醒相关函数，具体功能如下：

（注：KTCB 内核线程控制块）

task\_set\_wqh 设置 KTCB 中等待队列相关参数

task\_unset\_wqh 清除 KTCB 中等待队列相关参数

//以下是等待方法1

wait\_on\_prepared\_wait 线程已做好等待准备时的调度

wait\_on\_prepare 线程等待准备（需结合 wait\_on\_prepared\_wait 一起使用）

//以下是等待方法2

wait\_on 直接线程等待

//注意，目前两种等待方法的区别主要在于方法2少了关抢占、开抢占的过程，具体原因还有待分析

//唤醒

wake\_up\_all 唤醒所有线程

wake\_up 唤醒一个线程

wake\_up\_task 唤醒一个给定线程

* 1. 数据结构

//waitqueue 链表头

struct waitqueue\_head {

//睡眠任务数量

int sleepers;

//自旋锁

struct spinlock slock;

//任务列表的链表

struct link task\_list;

};

//waitqueue 链表（项）

struct waitqueue {

//注意，link 都放在最前

struct link task\_list;

//内核线程控制块

struct ktcb \*task;

};

* 1. 代码注释

/\*

\* Implementation of wakeup/wait for processes.

\*

\* Copyright (C) 2007, 2008 Bahadir Balban

\*/

#include <l4/generic/scheduler.h>

#include <l4/lib/wait.h>

#include <l4/lib/spinlock.h>

#include <l4/api/errno.h>

//这里似乎没有考虑线程池，而仅仅是提供单内核线程（可抢占或不可抢占）

//使用等待队列挂起用户线程（当前运行在内核空间）

//等待唤醒

//可能有不同的 waitqueue！

/\*

\* This sets any wait details of a task so that any arbitrary

\* wakers can know where the task is sleeping.

\*/

//在 ktcb 中设置 waitqueue 相关的内容

//TODO: Rev UTCB & KTCB

void task\_set\_wqh(struct ktcb \*task, struct waitqueue\_head \*wqh,

struct waitqueue \*wq)

{

//IRQ 状态存储变量

unsigned long irqflags;

//锁 wait

//TODO: 什么时候中断什么时候只关抢占？

spin\_lock\_irq(&task->waitlock, &irqflags);

//设置所在 waitqueue 头以及自己的项

task->waiting\_on = wqh;

task->wq = wq;

//解锁 wait

spin\_unlock\_irq(&task->waitlock, irqflags);

}

/\*

\* This clears all wait details of a task. Used as the

\* task is removed from its queue and is about to wake up.

\*/

//取消 ktcb 中对应 waitqueue 相关内容（用于从 waitqueue 中移除对应线程后）

void task\_unset\_wqh(struct ktcb \*task)

{

unsigned long irqflags;

spin\_lock\_irq(&task->waitlock, &irqflags);

//置零

task->waiting\_on = 0;

task->wq = 0;

spin\_unlock\_irq(&task->waitlock, irqflags);

}

/\*

\* Initiate wait on current task that

\* has already been placed in a waitqueue

\*

\* NOTE: This enables preemption and wait\_on\_prepare()

\* should be called first.

\*/

//线程已经处于 wait 状态时的调度

int wait\_on\_prepared\_wait(void)

{

/\* Now safe to sleep by preemption \*/

//开抢占（抢占优先于调度）

preempt\_enable();

/\* Sleep voluntarily to initiate wait \*/

//开调度

schedule();

//这里等到可以调度这个线程

//TODO: 有哪些状态啊可以调度？下面的 TASK\_INTERRUPTED 是如何引入？

/\* Did we wake up normally or get interrupted \*/

if (current->flags & TASK\_INTERRUPTED) {

//去除 TASK\_INTERRUPTED

current->flags &= ~TASK\_INTERRUPTED;

return -EINTR;

}

/\* No errors \*/

return 0;

//TODO: 然后如何处理？考虑对该函数的调用

}

/\*

\* Do all preparations to sleep but return without sleeping.

\* This is useful if the task needs to get in the waitqueue before

\* it releases a lock.

\*

\* NOTE: This disables preemption and it should be enabled by a

\* call to wait\_on\_prepared\_wait() - the other function of the pair.

\*/

//需要结合 wait\_on\_prepared\_wait() 一起使用

//wait 的准备工作，即将线程置于 wait 状态

int wait\_on\_prepare(struct waitqueue\_head \*wqh, struct waitqueue \*wq)

{

unsigned long irqflags;

/\* Disable to protect from sleeping by preemption \*/

preempt\_disable();

//这里是要锁中断的 spin lock

//TODO: Rev 是否锁 IRQ 在代码上的区别

spin\_lock\_irq(&wqh->slock, &irqflags);

///////////////////////////////////////////

wqh->sleepers++;

//插入队列

list\_insert\_tail(&wq->task\_list, &wqh->task\_list);

//设置 ktcb

task\_set\_wqh(current, wqh, wq);

//TODO: Rev wait 和一般 sleep 的区别，以及为什么要引入 waitqueue

//设置 ktcb 睡眠状态

sched\_prepare\_sleep();

//printk("(%d) waiting on wqh at: 0x%p\n",

// current->tid, wqh);

///////////////////////////////////////////

spin\_unlock\_irq(&wqh->slock, irqflags);

//此时仍然处于关闭抢占的状态

return 0;

}

/\* Sleep without any condition \*/

//直接 wait

//和前面一种方式的区别是，在整个过程中不涉及关抢占、开抢占

//TODO: 为何要设置两种方式？

int wait\_on(struct waitqueue\_head \*wqh)

{

unsigned long irqsave;

CREATE\_WAITQUEUE\_ON\_STACK(wq, current);

spin\_lock\_irq(&wqh->slock, &irqsave);

///////////////////////////////////////////////

wqh->sleepers++;

list\_insert\_tail(&wq.task\_list, &wqh->task\_list);

task\_set\_wqh(current, wqh, &wq);

sched\_prepare\_sleep();

//printk("(%d) waiting on wqh at: 0x%p\n",

// current->tid, wqh);

///////////////////////////////////////////////

spin\_unlock\_irq(&wqh->slock, irqsave);

//直接调度

schedule();

//等待唤醒

/\* Did we wake up normally or get interrupted \*/

if (current->flags & TASK\_INTERRUPTED) {

current->flags &= ~TASK\_INTERRUPTED;

return -EINTR;

}

return 0;

}

/\* Wake up all in the queue \*/

//唤醒队列中所有线程

void wake\_up\_all(struct waitqueue\_head \*wqh, unsigned int flags)

{

unsigned long irqsave;

spin\_lock\_irq(&wqh->slock, &irqsave);

////////////////////////////////////////////////////

//异常情况

BUG\_ON(wqh->sleepers < 0);

//wqh->sleepers =(==) 0 时跳

while (wqh->sleepers > 0) {

//TODO: Rev C99

struct waitqueue \*wq = link\_to\_struct(wqh->task\_list.next,

struct waitqueue,

task\_list);

struct ktcb \*sleeper = wq->task;

task\_unset\_wqh(sleeper);

//异常情况

BUG\_ON(list\_empty(&wqh->task\_list));

//TODO: 如果只有一个了怎么办？是否会把那一个删掉？Rev List 的处理方式

list\_remove\_init(&wq->task\_list);

wqh->sleepers--;

//TODO: 这里来决定唤醒方式：关闭中断下的唤醒？

if (flags & WAKEUP\_INTERRUPT)

sleeper->flags |= TASK\_INTERRUPTED;

// printk("(%d) Waking up (%d)\n", current->tid, sleeper->tid);

///////////////////////////////////////////////

spin\_unlock\_irq(&wqh->slock, irqsave);

if (flags & WAKEUP\_SYNC)

//同步唤醒（立即调度）

sched\_resume\_sync(sleeper);

else

//异步唤醒（等待最后统一调度）

sched\_resume\_async(sleeper);

//重新关中断

///////////////////////////////////////////////

spin\_lock\_irq(&wqh->slock, &irqsave);

}

spin\_unlock\_irq(&wqh->slock, irqsave);

///////////////////////////////////////////////////

}

/\* Wake up single waiter \*/

//唤醒一个线程

void wake\_up(struct waitqueue\_head \*wqh, unsigned int flags)

{

unsigned long irqflags;

BUG\_ON(wqh->sleepers < 0);

spin\_lock\_irq(&wqh->slock, &irqflags);

//和前面几乎一样

if (wqh->sleepers > 0) {

struct waitqueue \*wq = link\_to\_struct(wqh->task\_list.next,

struct waitqueue,

task\_list);

struct ktcb \*sleeper = wq->task;

BUG\_ON(list\_empty(&wqh->task\_list));

list\_remove\_init(&wq->task\_list);

wqh->sleepers--;

task\_unset\_wqh(sleeper);

if (flags & WAKEUP\_INTERRUPT)

sleeper->flags |= TASK\_INTERRUPTED;

// printk("(%d) Waking up (%d)\n", current->tid, sleeper->tid);

spin\_unlock\_irq(&wqh->slock, irqflags);

if (flags & WAKEUP\_SYNC)

sched\_resume\_sync(sleeper);

else

sched\_resume\_async(sleeper);

return;

}

spin\_unlock\_irq(&wqh->slock, irqflags);

}

/\*

\* Wakes up a task. If task is not waiting, or has been woken up

\* as we were peeking on it, returns -1. @sync makes us immediately

\* yield or else leave it to scheduler's discretion.

\*/

//唤醒指定线程

int wake\_up\_task(struct ktcb \*task, unsigned int flags)

{

unsigned long irqflags[2];

struct waitqueue\_head \*wqh;

struct waitqueue \*wq;

spin\_lock\_irq(&task->waitlock, &irqflags[0]);

//////////////////////////////////////////////

if (!task->waiting\_on) {

//////////////////////////////////////////

spin\_unlock\_irq(&task->waitlock, irqflags[0]);

return -1;

}

//从 ktcb 中取参数

wqh = task->waiting\_on;

wq = task->wq;

/\*

\* We have found the waitqueue head.

\*

\* That needs to be locked first to conform with

\* lock order and avoid deadlocks. Release task's

\* waitlock and take the wqh's one.

\*/

//task->waitlock 用于检测 wait 状态

///////////////////////////////////////////////

spin\_unlock\_irq(&task->waitlock, irqflags[0]);

/\*

\* Task can be woken up by someone else here.

\*/

//TODO: 为什么要做锁分离的设计？为何这样可以避免 deadlock？Rev 其他操作系统

//切换到 wqh 锁

spin\_lock\_irq(&wqh->slock, &irqflags[0]);

///////////////////////////////////////////////////

/\*

\* Now lets check if the task is still

\* waiting and in the same queue. Not irq version

\* as we called that once already (so irqs are disabled)

\*/

//再一次加 waitlock 锁

spin\_lock\_irq(&task->waitlock, &irqflags[1]);

///////////////////////////////////////////////////

//检查是不是还在 wait

if (task->waiting\_on != wqh) {

/\* No, task has been woken by someone else \*/

///////////////////////////////////////////////

spin\_unlock\_irq(&wqh->slock, irqflags[0]);

///////////////////////////////////////////////

spin\_unlock\_irq(&task->waitlock, irqflags[1]);

return -1;

}

/\* Now we can remove the task from its waitqueue \*/

list\_remove\_init(&wq->task\_list);

wqh->sleepers--;

task->waiting\_on = 0;

task->wq = 0;

//INTERRUPTED

if (flags & WAKEUP\_INTERRUPT)

task->flags |= TASK\_INTERRUPTED;

///////////////////////////////////////////////////

spin\_unlock\_irq(&wqh->slock, irqflags[0]);

///////////////////////////////////////////////////

spin\_unlock\_irq(&task->waitlock, irqflags[1]);

/\*

\* Task is removed from its waitqueue. Now we can

\* safely resume it without locks as this is the only

\* code path that can resume the task.

\*/

if (flags & WAKEUP\_SYNC)

//SYNC

sched\_resume\_sync(task);

else

//ASYNC

sched\_resume\_async(task);

return 0;

}

* 1. 疑问或改进建议

如原来代码注释所说，这一套对于等待队列并未使用优先级，有可能出现饥饿状况，所以可以考虑添加优先级。

不过，在wake\_up\_task出现了锁分离的情况，详见代码注释，尚未弄清楚设计原因；并且两种 wait 方式对抢占的处理目的还是没有弄太清楚。

* 1. 体系结构和平台相关的代码

无

1. 总结

该目录（lib）下内容不算复杂，不过存在的问题（或错误）以及疑惑之处还是有很多（详见每一文件的“疑问或改进建议”）。总的来说没有比较复杂的调用关系，少量函数引用了平台相关代码，主要是输出函数相关（putc）以及原子性锁操作相关（mutex）。复杂的数据结构也很少，主要在内存缓存（memcache）的设计上有较复杂结构上的考量（包括对齐、位图等）。

此次分析还是比较粗糙，注释不是很规范，且代码注释中给出了比较多的代码。在逐步规范注释后会对该文档作进一步修订。