haihui0705

博客访问： 231670
博文数量： 79
博客积分： 3458
博客等级： 中校
技术积分： 921
用 户 组： 普通用户
注册时间： 2010-05-25 17:09

# 内存池技术

原创 分类： LINUX 2010-12-26 17:05:08

## 内存池技术

内存池技术用来解决因不停的对系统堆malloc/free，而造成的系统调用开销和内存碎片问题。
内存池的原理就是预先malloc一块大block，程序需要内存时从这个block里面取，用完再归还到此block里面。如果block里面的内存不够，可以再次malloc一个或若干个block，供给程序调用。这样就会节省不少的系统调用时间，也不会形成内存碎片。
内存池的重要数据结构是：指向内存块block的头指针block_head，组成内存块block的内存节点node，还有一个指向空闲节点链表的头指针free_head。
为内存池mem_pool分配内存块后，内存块插入到内存块的链表里，内存块里的所有节点node就插入到空闲节点链表里，使用内存时只要移动空闲节点链表的头指针free_head，归还时按相反的

方式移动free_head。这种情况适合分配固定大小内存的情况。分配和释放过程的时间复杂度为O(1)。

boost::pool

采用的是预测模型，对每次分配的block大小都加倍，这也是std::vector内存增长采用的模型。ordered_free(void *p)：free会把释放的节点放到自由链表的开头，而ordered_free则假设自由链表是有序的，会遍历自由链表，并把返回的内存插入合适的位置。

sgi-stl

大家都说他是比较通用的策略：建立16个mem_pool，<=8字节的内存申请由0号mem_pool分配，<=16字节的内存申请由1号mem_pool分配，<=24字节的内存有2号mem_pool分配，以此类推。最后，>128字节的内存申请由普通的malloc分配。

这里实现的是单CPU下的单线程，分配固定大小内存的内存池，其他的内存池需要不同的策略。

```
/************************mem_pool.h************************
  *
  * * author:bripengandre
  * * modified by: LaoLiulaoliu
  * * TODO: thread_safe, different model
  *
  * ***********************************************************/

#ifndef _MEM_POOL_H_
#define _MEM_POOL_H_

#define BUF_SIZE 100
#define BASE_COUNT 10000
#define STEP_COUNT 1000


typedef union _mem_node
```


yshiilu

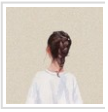
simiaoxi

深蓝苹果

xueyumic

康龙1990

## 最近访客


浪花小雨

yxl15098

风际的云

aqliuwz

wang3890

wtliang5

65188430

wangguop

junxy

## 推荐博文

- 稀奇稀奇真稀奇，asm实例建用...
- cvu在干什么？
- shell编程当中的注意事项...
- 对于io性能oracle的要求...
- oradebug 之后的工具 dbx...

```
{
        char buf[BUF_SIZE];
        union _mem_node *next;
} mem_node_t, *pmem_node_t;

/* consider of the efficiency, node_cnt can be annotated */
/* used to store block information */
typedef struct _mem_block
{
        mem_node_t *node_head;
        mem_node_t *node_tail;
        int node_cnt; /* node count */
        struct _mem_block *next;
} mem_block_t, *pmem_block_t;

/* consider of the efficiency, block_cnt can be annotated */
/* used to store the pool information */
typedef struct _mem_pool
{
        mem_block_t *block_head;
        // mem_block_t *block_tail;

        mem_node_t *free_head;
        int block_cnt; /* block count */
        int free_cnt; /* free node count; */
        int base;
        int step;
} mem_pool_t, *pmem_pool_t;
```

```
/* mem_pool will have at least base blocks, and will increase steps a time if
needed */
int mem_pool_init(int base, int step);
void mem_pool_destroy(void);
void print_mem_pool_info(void);

/* since the block size is constant, this function need no input parameter */
void *mem_alloc(void);
void mem_free(void *ptr);


#endif /* _MEM_POOL_H */
```

```
/************************mem_pool.c************************
  *
  * * author:bripengandre
  * * modified by: LaoLiulaoliu
  *
  * ***********************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "mem_pool.h"


/* static functions are the mainly expense of cpu in memory pool */
/* add new memory block to our memory pool */
static int add_mem_block(int cnt);
```

```
/* init the new block */
static int mem_block_init(int cnt, mem_block_t *block);
/* init free_list of the new block */
static int free_list_init(const mem_block_t *block);

static mem_pool_t mem_pool;

int mem_pool_init(int base, int step)
{
        if(base <= 0)
        {
                base = BASE_COUNT;
        }
        if(step <= 0)
        {
                step = STEP_COUNT;
        }

        /* initiate mem_pool */
        memset( &mem_pool, 0, sizeof(mem_pool) );
        mem_pool.base = base;
        mem_pool.step = step;

        /* add the base block(node of base count) into the memory pool */
        if( !add_mem_block(base) )
        {
                fprintf(stderr, "mem_pool_init::add_mem_block error\n");
                return 0;
        }
```

```
        return 1;
}


void mem_pool_destroy(void)
{
        mem_block_t *prev_block, *cur_block;

        prev_block = NULL;
        cur_block = mem_pool.block_head;
        while(cur_block != NULL)
        {
                prev_block = cur_block;
                cur_block = cur_block->next;
                /* mem_block_init() malloc once,so just free once of the head
pointer */
                free(prev_block->node_head);
                free(prev_block);
        }

        memset( &mem_pool, 0, sizeof(mem_pool_t) );
}


void print_mem_pool_info(void)
{
        int i;
        mem_block_t *p;

        if(mem_pool.block_head == NULL)
```

```
        {
                fprintf(stderr, "memory pool has not been created!\n");
                return;
        }
        printf("***************memory pool information
start*******************\n");
        printf("base block size: %d\n", mem_pool.base);
        printf("increasing block size: %d\n", mem_pool.step);
        printf("block count: %d\n", mem_pool.block_cnt);
        printf("current free node count: %d\n", mem_pool.free_cnt);
        printf("the first block: %#x\n", mem_pool.block_head);
        //printf("the last block: %#x\n", mem_pool.block_tail);

        printf("the first free node: %#x\n\n", mem_pool.free_head);
        for(p = mem_pool.block_head, i = 0; p != NULL; p = p->next, i++)
        {
                printf("-----------------block %d----------------------\n",
i+1);
                printf("node count: %d\n", p->node_cnt);
                printf("the first node: %#x\n", p->node_head);
                printf("-----------------------------------------------\n");
        }
        printf("***************memory pool information
end*********************\n\n");
}


void *mem_alloc(void)
{
        mem_node_t *p;
```

```
        /* no free node ready, attempt to allocate new free node */
        if(mem_pool.free_head == NULL)
        {
                if( !add_mem_block(mem_pool.step) )
                {
                        fprintf(stderr, "mem_alloc::add_mem_block error\n");
                        return NULL;
                }
        }

        /* get free node from free_list */
        p = mem_pool.free_head;
        mem_pool.free_head = p->next;

        /* decrease the free node count */
        mem_pool.free_cnt--;

        return p;
}


void mem_free(void *ptr)
{
        if(ptr == NULL)
        {
                return;
        }

        /* return the node to free_list */
```

```
        ((mem_node_t *)ptr)->next = mem_pool.free_head;
        mem_pool.free_head = ptr;

        /* increase the free node count */
        mem_pool.free_cnt++;
}


static int add_mem_block(int cnt)
{
        mem_block_t *block;

        if( (block = malloc(sizeof(mem_block_t))) == NULL )
        {
                fprintf(stderr, "mem_pool_init::malloc block error\n");
                return 0;
        }

        if( !mem_block_init(cnt, block) )
        {
                fprintf(stderr, "mem_pool_init::mem_block_init error\n");
                return 0;
        }

        /* insert the new block in the head */
        /* for the first time, block->next == NULL */
        block->next = mem_pool.block_head;
        mem_pool.block_head = block;
        // if(mem_pool.block_tail == NULL)
        // {
```

```
        //          mem_pool.block_tail = block;
        // }

        /* insert the new block into the free list */
        /* block->node_tail->next == NULL in these two situations of
add_mem_block() */
        block->node_tail->next = mem_pool.free_head;
        mem_pool.free_head = block->node_head;
        mem_pool.free_cnt += cnt;

        /* increase the block count */
        mem_pool.block_cnt++;

        return 1;
}


static int mem_block_init(int cnt, mem_block_t *block)
{
        size_t size;
        mem_node_t *p;

        if(block == NULL)
        {
                return 0;
        }

        size = cnt * sizeof(mem_node_t);
        if( (p = malloc(size)) == NULL )
        {
```

```
                fprintf(stderr, "mem_pool_init::malloc node error\n");
                return 0;
        }
        memset(p, 0, size);
        memset(block, 0, sizeof(mem_block_t));
        block->node_cnt = cnt;
        block->node_head = p;
        block->node_tail = p+cnt-1;
        free_list_init(block);

        return 1;
}


static int free_list_init(const mem_block_t *block)
{
        mem_node_t *p, *end;

        if(block == NULL)
        {
                return 0;
        }

        /* start initiating free list */
        end = block->node_tail; /* block_cnt > 0 */
        for(p = block->node_head; p < end; p++)
        {
                p->next = (p+1);
        }
        p->next = NULL; /* end->next = NULL */
```

```
        return 1;
}
```

测试程序：

```
/**********************mem_pool_debug.c*********************/
#include <stdio.h>
#include <stdlib.h>
#include "mem_pool.h"

#define ALLOC_COUNT 10

void alloc_test(char *ptr[])
{
        int i, j;

        for(i = 0; i < ALLOC_COUNT; i++)
        {
                if( (ptr[i] = mem_alloc()) == NULL )
                {
                        fprintf(stderr, "mem_alloc error\n");
                        return;
                }
                for(j = 0; j < ALLOC_COUNT; j++)
                {
                        ptr[i][j] = 'a' + j;
                }
        }
        for(i = 0; i < ALLOC_COUNT; i++)
```

```
        {
                for(j = 0; j < ALLOC_COUNT; j++)
                {
                        printf("ptr[%d][%d]=%c ", i, j, ptr[i][j]);
                }
                fputc('\n', stdout);
        }
}


int main(int argc, char *argv[])
{
        int base, step;
        char *ptr1[ALLOC_COUNT], *ptr2[ALLOC_COUNT];

        switch(argc)
        {
                case 1:
                        base = 0; /* default count */
                        step = 0; /* default count */
                        break;
                case 2:
                        base = atoi(argv[1]);
                        step = 0;
                        break;
                case 3:
                        base = atoi(argv[1]);
                        step = atoi(argv[2]);
                        break;
                default:
```

```
                fprintf(stderr, "Usage: %s [ [step]]\n", argv[0]);
                break;
        }

        if( !mem_pool_init(base, step) )
        {
                fprintf(stderr, "mem_pool_init error\n");
                return 1;
        }
        print_mem_pool_info();
        alloc_test(ptr1);
        print_mem_pool_info();

        //mem_free(ptr1[5]);

        print_mem_pool_info();

        alloc_test(ptr2);
        print_mem_pool_info();

        mem_pool_destroy();


        /* once again */
        if( !mem_pool_init(base, step) )
        {
                fprintf(stderr, "mem_pool_init error\n");
                return 1;
        }
        print_mem_pool_info();
```

```
        alloc_test(ptr1);
        print_mem_pool_info();

        mem_free(ptr1[5]);
        print_mem_pool_info();

        alloc_test(ptr2);
        print_mem_pool_info();

        mem_pool_destroy();

        return 1;
}
```

jjjj

编译： gcc mem_pool_debug.c mem_pool.c -o test_mem_pool

运行： ./test_mem_pool

结果： 当分配的BUF_SIZE 100~1000 时，比直接malloc的效率比会升高。

参考链接：

程序原作者：http://blog.csdn.net/bripengandre/archive/2008/11/02/3206018.aspx

清晰的解释：http://blog.csdn.net/xushiweizh/archive/2006/11/22/1402967.aspx

简单的自动回收器：http://blog.csdn.net/xushiweizh/archive/2006/11/19/1396573.aspx

上一篇：linux做RAID1的具体方法
下一篇：阅读C语言程序的有效方式



给主人留下些什么吧！~~

chinaunix网友 2011-01-03 16:15:36

很好的，收藏了 推荐一个博客，提供很多免费软件编程电子书下载： http://free-ebooks.appspot.com

回复 | 举报