CS61 Lab3

参与人员：CS61讨论组

Outline

- 相关背景知识
- CS61 Lab3

Outline

- 相关背景知识
- CS61 Lab3

内存管理概述（内核态）

- 内核内存分配器
  - 资源图分配算法
  - 2的幂次方空闲链表
  - McKusick-Karels分配算法
  - Mach的区域（Zone）分配算法
  - Dynix分配算法
  - Slab分配器
  - 伙伴系统

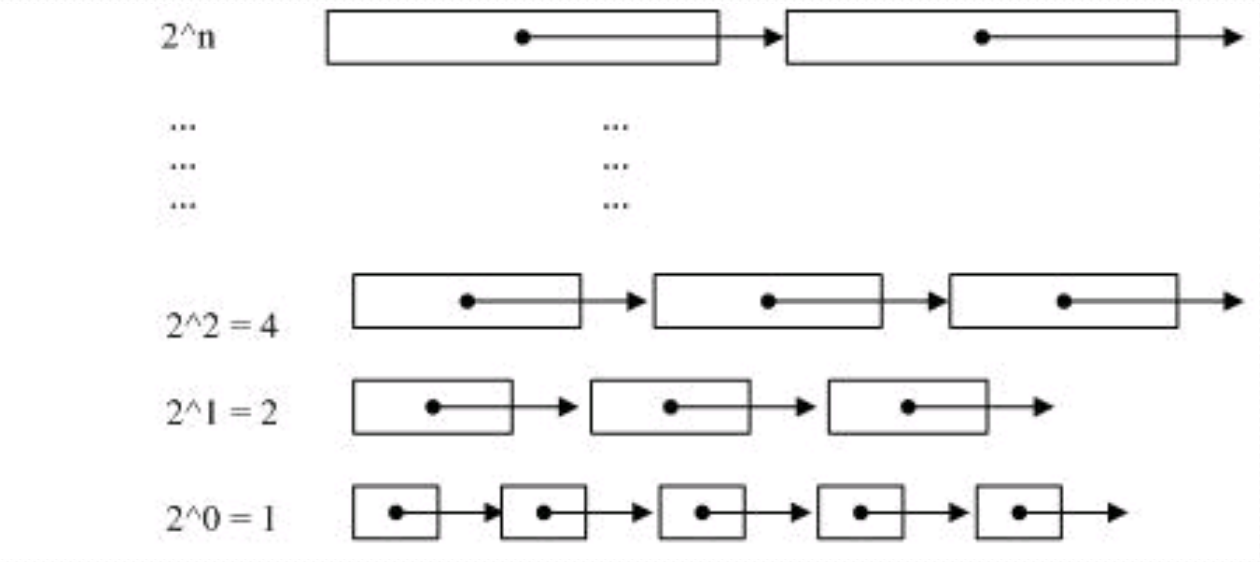
  Linux内核在伙伴系统之上采用了Slab分配算法。

Slab（slob、slub）分配器

- Slab分配器主要思想（最初由 Jeff Bonwick在SunOS引入）
  - 整齐排列。
  - 对象缓存。
  - 对象复用。
  - 一些对象在析构以前，通常已经回到了初始化的状态。另外，对象的初始化比其分配更耗时间。一个对象 A 在不被使用后，slab算法不急于调用析构器回收内存，而是等下一个同类对象B使用。这样对象 B 可以利用 A 的数据不用初始化直接投入使用。从而节省了分配、构造/析构、回收的开销。

伙伴系统

- Linux采用了伙伴系统算法来管理内存,即把内页按2^0,2^1,2^2...2^10大小进行分组.每次分配内存时,从相应大小的池中分配内存,然后再把余下的内存分配给它的下一级缓存池.



内存管理概述（用户态）

- 应用程序内存分配器
  - dlmalloc（ Doug Lea ）
    - 由Doug Lea完成，于1987年首次发布，目前最新版本为2.8.4。
  - nedmalloc（ Niall Douglas ）
    - 基于dlmalloc，支持多线程程序，分配效率很高，Git项目中使用了nedmalloc。
  - ptmalloc（ Wolfram Gloger ）
    - 基于dlmalloc，目前发布了ptmalloc 1/2/3共三个版本，glibc中的内存分配即采用ptmalloc。
  - tcmalloc（ Google ）
    - 由google发布，在perftools中提供，垃圾回收。
  - Jemalloc（ Jason Evans ）
    - OpenBSD以及其他项目（mozilla）中采用。
  - hoard（ Emery Berger ）
    - 以64KB为“超级块”进行内存分配。
- http://en.wikipedia.org/wiki/Malloc

各种分配策略比较

- 首次适应算法
  - 从空闲分区链首开始查找，直至找到一个能满足其大小要求的空闲分区为止。然后从该分区中划出一块内存分配给请求者，余下的空闲分区仍留在空闲分区链中。
- 循环首次适应算法
  - 在为进程分配内存空间时，不再每次从链首开始查找，而是从上次找到的空闲分区开始查找，直至 找到一个能满足要求的空闲分区，并从中划出一块来分给作业。
- 最佳适应算法
  - 该算法总是把既能满足要求，又是最小的空闲分区分配给作业。
- 最差适应算法
  - 该算法按大小递减的顺序形成空闲区链，分配时直接从空闲区链的第一个空闲分区中 分配（不能满足需要则不分配）。

Linux典型的进程空间映像

- struct mm_struct 结构
  - 保存了与进程地址空间的有关的全部信息。
  - 重要字段（本节我们关心的）
    - unsigned long start_brk　　堆的起始地址。
    - unsigned long brk　　堆的当前最后地址。
    - unsigned long start_stack　　用户态堆栈的起始地址。
    - unsigned long arg_start
    - unsigned long arg_end
    - unsigned long env_start
    - unsigned long env_end

brk和mmap系统调用

- sys_brk(addr)
  - 直接修改堆的大小。addr参数指定current->mm->brk的新值，返回值是线性区的新的结束地址（进程必须检查该地址和和请求的地址值addr是否一致）。
  - sbrk(incr)，不是系统调用，但类似于brk，不过其中的incr参数指定是增加还是减少的以字为单位的堆大小。
- sys_mmap(sys_mmap2)
  - Linux下大于128KB（该值可调整），内核采用mmap方式进行内存分配。可以在进程空间堆以外进行内存分配，比brk灵活，还可以映射文件，使读写文件如同读写内存一样方便。

Glibc内存分配器（一）

- 堆是通过 brk 的方式来增长或压缩的，如果在现有的堆中不能找到合适的 chunk，会通过增长堆的方式来满足分配，如果堆顶的空闲块超过一定的阀值会收缩堆，所以只要堆顶的空间没释放，堆是一直不会收缩的。

Glibc内存分配器（二）

- 第一．是通过 chunk 的头， chunk 中的头一个字是记录前一个 chunk 的大小，第二个字是记录当前 chunk 的大小和一些标志位，从第三个字开始是要使用的内存。所以通过内存地址可以找到 chunk ，通过 chunk 也可以找到内存地址。还可以找到相邻的下一个 chunk ，和相邻的前一个 chunk 。一个堆完全是由 n 个 chunk 组成 。
- 第二．是由 3 种队列记录，只用空闲 chunk 才会出现在队列中，使用的 chunk 不会出现在队列中。如果内存块是空闲的它会挂到其中一个队列中，它是通过内存复用的方式，使用空闲 chunk 的第 3 个字和第 4 个字当作它的前链和后链（变长块是第 5 个字和第 6个字）。

Glibc内存分配器（三）

- 第一种队列是 bins， bins 有 128 个队列，前 64 个队列是定长的，每隔 8 个字节大小的块分配在一个队列，后面的 64 个队列是不定长的，就是在一个范围长度的都分配在一个队列中。所有长度小于 512 字节（大约）的都分配在定长的队列中 。后面的 64 个队列是变长的队列，每个队列中的 chunk 都是从小到大排列的。
- 第二种队列是 unsort 队列（只有一个队列），（是一个缓冲）所有 free 下来的如果要进入 bins 队列中都要经过 unsort 队列。
- 第三种队列是 fastbins，大约有 10 个定长队列，（是一个高速缓冲）所有 free 下来的并且长度是小于 80 的 chunk 就会进入这种队列中。进入此队列的 chunk 在 free 的时候并不修改使用位，目的是为了避免被相邻的块合并掉。

Glibc内存分配器（四）

- malloc 的步骤
  I. 先在 fastbins 中找，如果能找到，从队列中取下后（不需要再置使用位为 1 了）立刻返回。
  II. 判断需求的块是否在小箱子（ bins 的前 64 个 bin ）范围，如果在小箱子的范围，并且刚好有需求的块，则直接返回内存地址；如果范围在大箱子（ bins 的后 64 个 bin ）里，则触发 consolidate 。（因为在大箱子找一般都要切割，所以要优先合并，避免过多碎片）
  III. 然后在 unsort 中取出一个 chunk ，如果能找到刚好和想要的 chunk 相同大小的 chunk ，立刻返回，如果不是想要 chunk 大小的 chunk ，就把他插入到 bins 对应的队列中去。直到清空，或者一次循环了 10000 次。

Glibc内存分配器（四——续）

I. 然后才在 bins 中找，找到一个最小的能符合需求的 chunk 从队列中取下，如果剩下的大小还能建一个 chunk，就把 chunk 分成两个部分，把剩下的 chunk 插入到 unsort 队列中去，把 chunk 的内存地址返回。

II. 在 topchunk（是堆顶的一个 chunk，不会放到任何一个队列里的）找，如果能切出符合要求的，把剩下的一部分当作 topchunk，然后返回内存地址。

III. 如果 fastbins 不为空，触发 consolidate 即把所有的 fanbins 清空（是把 fanbins 的使用位置 0，把相邻的块合并起来，然后挂到 unsort 队列中去），然后继续第 3 步。

IV. 还找不到话就调用 sysalloc，其实就是增长堆了。然后返回内存地址。

Glibc内存分配器（五）

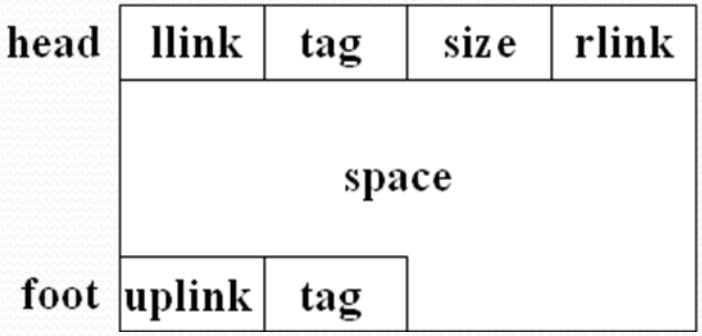
- free 的步骤
  I. 如果和 topchunk 相邻，直接和 topchunk 合并，不会放到其他的空闲队列中去。
  II. 如果释放的大小小于 80 字节，就把它挂到 fastbins 中去，使用位仍然为 1，当然更不会去合并相邻块。
  III. 如果释放块大小介于 80-128k，把 chunk 的使用位置成 0，然后试图合并相邻块，挂到 unsort 队列中去，如果合并后的大小大于 64k，也会触发 consolidate，（可能是周围比较多小块了吧），然后才试图去收缩堆。（收缩堆的条件是当前 free 的块大小加上前后能合并 chunk 的大小大于 64k，并且要堆顶的大小要达到阀值，才有可能收缩堆
  IV. 对于大于 128k 的块，直接 munmap

Outline

- 相关背景知识
- CS61 Lab3

回归正题，CS61 Lab3

- 边界标志算法
- 由***Donald Knuth***提出，目前绝大部分内存分配器都采用了边界标志算法。



分配算法

- 采用了首次适应和循环首次适应算法。
- 对首次拟合的分配算法进行一下介绍。设申请空间的大小为size_n，搜索内存块的指针为pointer_pav。首先获取当前内存块头部信息中内存块的大小size，如果内存块的大size大于或者等于申请空间的大小size_n，则将此内存块的一部分或整个分配给用户。
- 如果内存块的大小size小于申请空间的大小size_n，则将搜索内存块的指针pointer_pav指向下一个空闲的内存块。继续比较，直到找到或搜索完所有的空闲内存块为止。 为了避免产生无法使用的内存碎片，我们定义一个常量size_e。如果分配给用户后内存块剩余空间的大小小于常量size_e，则将整个内存块全部分配给用户。否则，只分配申请空间的大小。
- 另外，如果搜索指针的位置固定从某一个内存块开始，势必产生内存分布不均，一些地方密集，而一些地方稀疏。为了解决这个问题，我们将搜索指针指向每次释放后的内存块。

回收算法（一）

- 1、左右邻区都为空闲的情况；
- 2、左右邻区都为占用的情况；
- 3、左邻区为空闲，右邻区为占用的情况；
- 4、左邻区为占用，右邻区为空闲的情况。

回收算法（二）

- 第一种：左右邻区都为空闲的情况
  - 为了让三个空闲块合并成一个空闲块，可以修改左邻块的空间大小，然后，将右邻块从链表中删除，再修改合并后内存块的底部信息。
- 第二种：左右邻区都为占用的情况
  - 这种情况比较简单，只要将释放块插入到空闲块链表中即可。因为链表是无序的，所以插入位置是任意的。
- 第三种：左邻区为空闲，右邻区为占用的情况
  - 修改左邻块的空间大小，然后修改底部信息，将此释放块和左邻块合并成一个内存块。
- 第四种：左邻区为占用，右邻区为空闲的请况
  - 修改释放块的头部信息，然后修改释放块的底部信息。

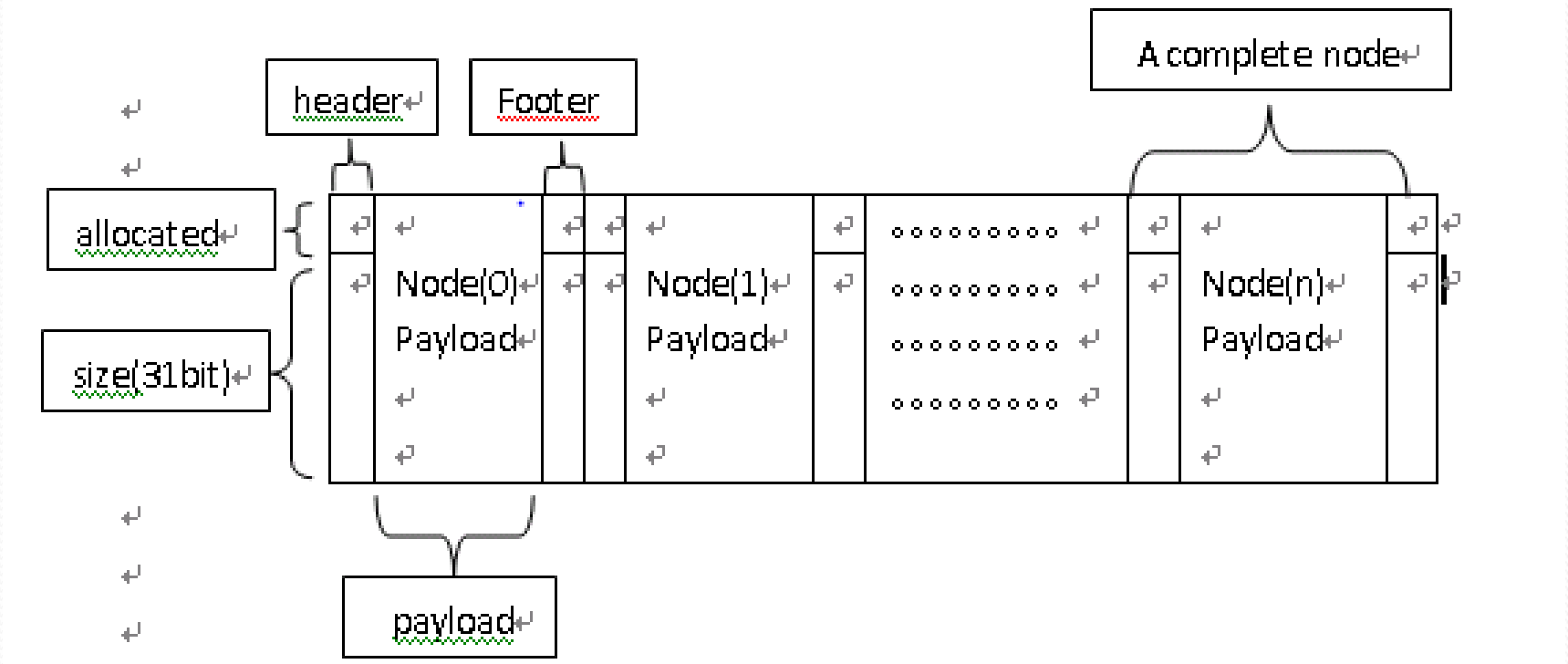
分配方案

- 分配方案采用了首次拟合法（first-fit）和和循环首次拟合法（next-fit），如下程序段所描述：

```
#ifdef NEXT_FIT
        static char  *g_pNextFit;  /*  循环首次适应算法的指针g_pNextFit  */
#endif
```

- 如果定义了宏NEXT_FIT，则采用了循环首次拟合法，否则默认为首次拟合法。

内存节点定义


A complete node
header
Footer
allocated
size(31bit)
Node(0)
Payload
Node(1)
Payload
Node(n)
Payload
payload

相关说明

- 在上面的节点定义中，每个节点有三部分组成，首部header，占用4个字节；尾部footer，也是占用4个字节，和payload部分，大小由void * mm_malloc(size_t size);中的size参数指定。
- 其中首部和尾部中的各个字段相同，首部中的size字段（31bits）为在内存块的大小，allocated字段（1bit）指示该内存块是否被占用，0表示空闲，1表示被占用。

常见宏定义

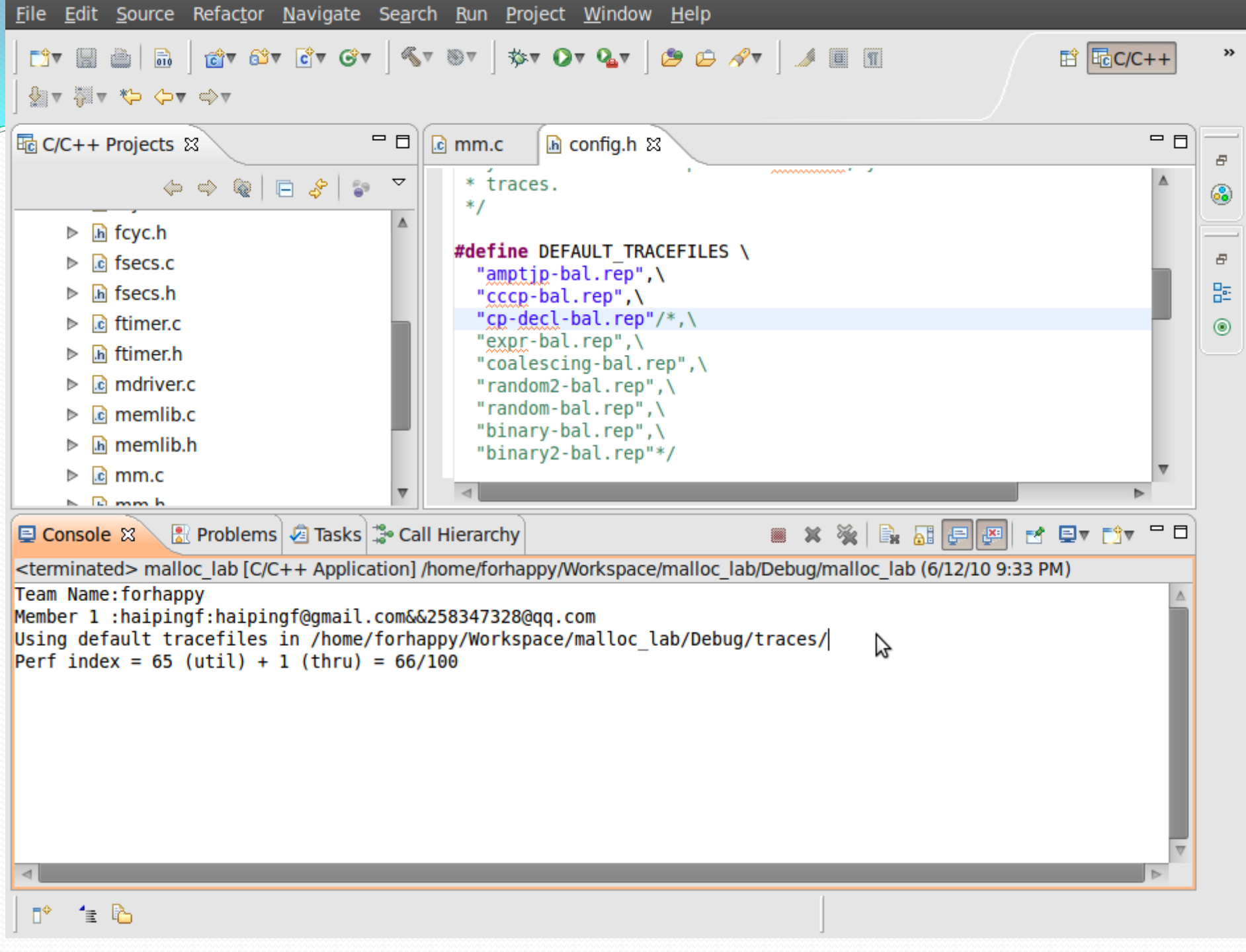
- /* 将size和allocated字段压缩成一个字*/
- #define COMPRESS(size, alloc) ((size) | (alloc))
- /* 从地址 p 所示的内存单元读取一个数据 */
- #define READ(p) (*(unsigned int *)(p))
- /* 从地址 p 所示的内存单元写入一个数据 */
- #define WRITE(p, val) (*(unsigned int *)(p) = (val))
- /* 从地址 P 指定的内存单元读取size字段，以获取当前内存块的大小*/
- #define GET_BLKSIZE(p) (READ(p) & ~(ALIGNMENT-1))
- /* 从地址 P 指定的内存单元读取allocated字段，以判断内存块是否被占用*/

常见宏定义

- #define IS_ALLOCATED(p) (READ(p) & 0x1)
- /* 如果给定指向某一内存块的指针，则计算其头部字段的地址*/
- #define CMPT_HEADPTR(bp)　　((char *)(bp) - WORD)
- /* 如果给定指向某一内存块的指针，则计算其尾部字段的地址*/
- #define CMPT_FOOTPTR(bp)　　((char *)(bp) + GET_BLKSIZE(CMPT_HEADPTR(bp)) - DWORD)
- /* 如果给定指向内存块的指针，则计算下一个内存块的地址*/
- #define GET_NEXT_BLKPTR(bp) ((char *)(bp) + GET_BLKSIZE(((char *)(bp) - WORD)))
- /* 如果给定指向内存块的指针，则计算前一个内存块的地址*/
- #define GET_PREV_BLKPTR(bp) ((char *)(bp) - GET_BLKSIZE(((char *)(bp) - DWORD)))
- /* 全局变量 */
- static char *g_pHeadList = 0; /* 指向第一个内存块的指针 */

测试

- 硬件平台：
  - 处理器：Intel(R) Core ™2 Duo CPU T5250 @1.5GHZ 1.5GHZ
  - 内存：2GB DRAM
- 软件平台：
  - 操作系统：Ubuntu 10.04LTS
  - Linux Kernel：2.6.32-24+gcc4.4


File Edit Source Refactor Navigate Search Run Project Window Help
C/C++
C/C++ Projects
fcyc.h
fsecs.c
fsecs.h
ftimer.c
ftimer.h
mdriver.c
memlib.c
memlib.h
mm.c
mm.h
mm.c
config.h
* traces.
*/
#define DEFAULT_TRACEFILES \
"amptjp-bal.rep",\
"cccp-bal.rep",\
"cp-decl-bal.rep"/*,\
"expr-bal.rep",\
"coalescing-bal.rep",\
"random2-bal.rep",\
"random-bal.rep",\
"binary-bal.rep",\
"binary2-bal.rep"*/
Console
Problems
Tasks
Call Hierarchy
<terminated> malloc_lab [C/C++ Application] /home/forhappy/Workspace/malloc_lab/Debug/malloc_lab (6/12/10 9:33 PM)
Team Name:forhappy
Member 1 :haipingf:haipingf@gmail.com&&258347328@qq.com
Using default tracefiles in /home/forhappy/Workspace/malloc_lab/Debug/traces/
Perf index = 65 (util) + 1 (thru) = 66/100

性能分析

$$P = wU + (1 - w)\min\left(1, \frac{T}{T_{libc}}\right)$$

- 性能测试分为两个部分，1、[虚拟]内存空间利用率，2、吞吐量。其中内存空间利用率表述为：由自己实现的内存分配器所利用的堆大小与驱动程序所分配的总的“虚拟”内存的最高比率；吞吐量为：每秒钟平均的操作“虚拟”内存操作次数，即（mm_malloc，mm_realloc以及mm_free的次数）。
- 评估指数内存空间利用率和吞吐量的加权平均，内存空间利用率默认权值为0.65，吞吐量的权值为1-w，但此时“吞吐量”间接的表述为：（函数模拟的吞吐量/Tlibc函数库的吞吐量）与1中的较小值，

```
Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.016228   351
 1       yes   99%    5851  0.014187   412
 2       yes   99%    6651  0.027563   241
 3       yes  100%    5383  0.020013   269
 4       yes   66%   14403  0.000391 36808
 5       yes   92%    4803  0.016847   285
 6       yes   93%    4803  0.017719   271
 7       yes   55%   12003  0.216503    55
 8       yes   51%   24003  0.643756    37
Total          84%   83594  0.973208    86

Perf index = 55 (util) + 1 (thru) = 56/100
forhappy@ubuntu:~/Workspace/testing$
```

相关优化

- 程序采用了边界标志算法，这个算法的优势是回收合并内存只需要常量的时间，而分配策略采用了首次拟合算法和循环首次拟合算法，如果采取这样方案，它的致命弱点就是需要线性时间进行内存分配，随着空闲块的链表长度越来越长，分配的时间开销将变大。
- 针对上面提出的缺陷，应该对分配策略进行改进或采用额外的存储空间存储空闲链表的信息，这样在进行分配时就可以尽量做到在常数时间之类完成

优化后的测试

```
Team Name:forhappy
Member 1 :haipingf:haipingf@gmail.com&&258347328@qq.com
Using default tracefiles in ./
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   97%    5694  0.001018  5596
 1       yes   98%    5851  0.001115  5250
 2       yes   98%    6651  0.001187  5606
 3       yes   99%    5383  0.000953  5647
 4       yes   98%   14403  0.001428 10087
 5       yes   90%    4803  0.001385  3467
 6       yes   92%    4803  0.001392  3451
 7       yes   54%   12003  0.001728  6945
 8       yes   47%   24003  0.003275  7329
Total          86%   83594  0.013480  6201

Perf index = 56 (util) + 35 (thru) = 91/100
forhappy@ubuntu:~/Workspace/testing$
```

参考文献

- 严蔚敏，吴伟民著. 数据结（C语言版）北京：清华大学出版社，2007
- Donald.Knuth著 The Art Of Computer Programming 3rd Vol1(Addison Wesley,1997)
- http://g.oswego.edu/dl/html/malloc.html
- http://www.malloc.de/en/
- http://code.google.com/p/google-perftools/
- http://www.malloc.de/en/index.html
- http://www.nedprod.com/programs/portable/nedmalloc/
- http://www.hoard.org/
- http://www.canonware.com/jemalloc/