Project 4: The SLOB SLAB

Andrew Chase

OS II

Spring 2015

Abstract: Describes work done to build and run a custom Memory Allocator.

CONTENTS

I. Work Log

Date	Work Done
June 3	Did research, modified slob.c, wrote frag.py

II. Questions

A. What do you think the main point of this assignment is?

I think the main point of the assignment is to learn about memory management in operating systems and how the Linux memory management system works.

B. How did you personally approach the problem? Design decisions, algorithm, etc.

As usual first I looked up other approaches to the problem. Next I designed a simple algorithm. I focused on trying to make as few changes to the slob.c module as possible. My algorithm is a little wasteful as it goes through the whole list of blocks twice, once to find the best fit, and once to re-find the best-fit block to select it for use.

For the memory fragmentation I found a reference for a linux system tool that allows me to grab the list of blocks easily so that I can perform operations on them.

C. How did you ensure your solution was correct? Testing details, for instance.

I ensured my solution was correct by reading the /proc/buddyinfo file which reveals the free memory blocks and their sizes. I also inserted printk messages to example the values of the variables at one point when my solution wasn't working correctly.

D. What did you learn?

I learned a bit about how the Linux block allocator works and how the different algorithms can effect performance. I also learned a bit about how to debug memory errors in QEMU. I learned how to perform research and wire up solutions to problems in Kernel development.

III. Git Log

acronym	meaning
V	`version`
tag	`git tag`
MF	Number of `modified files`.
AL	Number of `added lines`.
DL	Number of `deleted lines`.

V	tag	date	commit message	MF	AL	DL
1		2014-12-09	Merge tag 'v3.14.24'	45949	18281318	0
2		2016-06-03	Add plan	1	14	0
3		2016-06-03	Implement best-first	1	27	30
4		2016-06-03	Fix int declarations to c90	1	5	4
5		2016-06-03	Fix additional warnings	1	7	5
6		2016-06-03	Fix bug where slob would never allocate	1	3	3
7		2016-06-03	Fix reversed for loop args	1	1	1
8		2016-06-03	Fix base case	1	1	1
9		2016-06-03	Fix no memory available	1	4	2
10		2016-06-03	Fix memory freeze	1	5	2
11		2016-06-03	Add fragmentation percentage script	1	32	0

IV. Code Listing

A. *slob.c*

```
/*
 * SLOB Allocator: Simple List Of Blocks
 *
 * Matt Mackall <mpm@selenic.com> 12/30/03
 *
 * NUMA support by Paul Mundt, 2007.
 *
 * How SLOB works:
 *
 * The core of SLOB is a traditional K&R style heap allocator, with
 * support for returning aligned objects. The granularity of this
 * allocator is as little as 2 bytes, however typically most architectures
 * will require 4 bytes on 32-bit and 8 bytes on 64-bit.
 *
 * The slob heap is a set of linked list of pages from alloc_pages(),
 * and within each page, there is a singly-linked list of free blocks
 * (slob_t). The heap is grown on demand. To reduce fragmentation,
 * heap pages are segregated into three lists, with objects less than
```

```
 * 256 bytes, objects less than 1024 bytes, and all other objects.
 *
 * Allocation from heap involves first searching for a page with
 * sufficient free blocks (using a next-fit-like approach) followed by
 * a first-fit scan of the page. Deallocation inserts objects back
 * into the free list in address order, so this is effectively an
 * address-ordered first fit.
 *
 * Above this is an implementation of kmalloc/kfree. Blocks returned
 * from kmalloc are prepended with a 4-byte header with the kmalloc size.
 * If kmalloc is asked for objects of PAGE_SIZE or larger, it calls
 * alloc_pages() directly, allocating compound pages so the page order
 * does not have to be separately tracked.
 * These objects are detected in kfree() because PageSlab()
 * is false for them.
 *
 * SLAB is emulated on top of SLOB by simply calling constructors and
 * destructors for every SLAB allocation. Objects are returned with the
 * 4-byte alignment unless the SLAB_HWCACHE_ALIGN flag is set, in which
 * case the low-level allocator will fragment blocks to create the proper
 * alignment. Again, objects of page-size or greater are allocated by
 * calling alloc_pages(). As SLAB objects know their size, no separate
 * size bookkeeping is necessary and there is essentially no allocation
 * space overhead, and compound pages aren't needed for multi-page
 * allocations.
 *
 * NUMA support in SLOB is fairly simplistic, pushing most of the real
 * logic down to the page allocator, and simply doing the node accounting
 * on the upper levels. In the event that a node id is explicitly
 * provided, alloc_pages_exact_node() with the specified node id is used
 * instead. The common case (or when the node id isn't explicitly provided)
 * will default to the current node, as per numa_node_id().
 *
 * Node aware pages are still inserted in to the global freelist, and
```

```
 * these are scanned for by matching against the node id encoded in the
 * page flags. As a result, block allocations that can be satisfied from
 * the freelist will only be done so on pages residing on the same node,
 * in order to prevent random node placement.
 */

#include <linux/kernel.h>
#include <linux/slab.h>

#include <linux/mm.h>
#include <linux/swap.h> /* struct reclaim_state */
#include <linux/cache.h>
#include <linux/init.h>
#include <linux/export.h>
#include <linux/rcupdate.h>
#include <linux/list.h>
#include <linux/kmemleak.h>

#include <trace/events/kmem.h>

#include <linux/atomic.h>

#include "slab.h"
/*
 * slob_block has a field 'units', which indicates size of block if +ve,
 * or offset of next block if -ve (in SLOB_UNITs).
 *
 * Free blocks of size 1 unit simply contain the offset of the next block.
 * Those with larger size contain their size in the first SLOB_UNIT of
 * memory, and the offset of the next free block in the second SLOB_UNIT.
 */
#if PAGE_SIZE <= (32767 * 2)
typedef s16 slobidx_t;
#else
```

```
87  typedef s32 slobidx_t;
88  #endif
89
90  struct slob_block {
91          slobidx_t units;
92  };
93  typedef struct slob_block slob_t;
94
95  /*
96   * All partially free slob pages go on these lists.
97   */
98  #define SLOB_BREAK1 256
99  #define SLOB_BREAK2 1024
100 static LIST_HEAD(free_slob_small);
101 static LIST_HEAD(free_slob_medium);
102 static LIST_HEAD(free_slob_large);
103
104 /*
105  * slob_page_free: true for pages on free_slob_pages list.
106  */
107 static inline int slob_page_free(struct page *sp)
108 {
109         return PageSlobFree(sp);
110 }
111
112 static void set_slob_page_free(struct page *sp, struct list_head *list)
113 {
114         list_add(&sp->list, list);
115         __SetPageSlobFree(sp);
116 }
117
118 static inline void clear_slob_page_free(struct page *sp)
119 {
120         list_del(&sp->list);
```

```
121         __ClearPageSlobFree(sp);
122 }
123
124 #define SLOB_UNIT sizeof(slob_t)
125 #define SLOB_UNITS(size) DIV_ROUND_UP(size, SLOB_UNIT)
126
127 /*
128  * struct slob_rcu is inserted at the tail of allocated slob blocks, which
129  * were created with a SLAB_DESTROY_BY_RCU slab. slob_rcu is used to free
130  * the block using call_rcu.
131  */
132 struct slob_rcu {
133         struct rcu_head head;
134         int size;
135 };
136
137 /*
138  * slob_lock protects all slob allocator structures.
139  */
140 static DEFINE_SPINLOCK(slob_lock);
141
142 /*
143  * Encode the given size and next info into a free slob block s.
144  */
145 static void set_slob(slob_t *s, slobidx_t size, slob_t *next)
146 {
147         slob_t *base = (slob_t *)((unsigned long)s & PAGE_MASK);
148         slobidx_t offset = next - base;
149
150         if (size > 1) {
151                 s[0].units = size;
152                 s[1].units = offset;
153         } else
154                 s[0].units = -offset;
```

```
155 }
156
157 /*
158  * Return the size of a slob block.
159  */
160 static slobidx_t slob_units(slob_t *s)
161 {
162         if (s->units > 0)
163                 return s->units;
164         return 1;
165 }
166
167 /*
168  * Return the next free slob block pointer after this one.
169  */
170 static slob_t *slob_next(slob_t *s)
171 {
172         slob_t *base = (slob_t *)((unsigned long)s & PAGE_MASK);
173         slobidx_t next;
174
175         if (s[0].units < 0)
176                 next = -s[0].units;
177         else
178                 next = s[1].units;
179         return base+next;
180 }
181
182 /*
183  * Returns true if s is the last free block in its page.
184  */
185 static int slob_last(slob_t *s)
186 {
187         return !((unsigned long)slob_next(s) & ~PAGE_MASK);
188 }
```

```
189
190 static void *slob_new_pages(gfp_t gfp, int order, int node)
191 {
192         void *page;
193
194 #ifdef CONFIG_NUMA
195         if (node != NUMA_NO_NODE)
196                 page = alloc_pages_exact_node(node, gfp, order);
197         else
198 #endif
199                 page = alloc_pages(gfp, order);
200
201         if (!page)
202                 return NULL;
203
204         return page_address(page);
205 }
206
207 static void slob_free_pages(void *b, int order)
208 {
209         if (current->reclaim_state)
210                 current->reclaim_state->reclaimed_slab += 1 << order;
211         free_pages((unsigned long)b, order);
212 }
213
214 /*
215  * Allocate a slob block within a given slob_page sp.
216  */
217 static void *slob_page_alloc(struct page *sp, size_t size, int align, int apply, int *be
218 {
219         slob_t *prev, *cur, *aligned = NULL;
220         int delta = 0, units = SLOB_UNITS(size);
221         int fits;
222
```

```
223         for (prev = NULL, cur = sp->freelist; ; prev = cur, cur = slob_next(cur)) {
224                 slobidx_t avail = slob_units(cur);
225
226                 if (align) {
227                         aligned = (slob_t *)ALIGN((unsigned long)cur, align);
228                         delta = aligned - cur;
229                 }
230                 fits = avail >= units + delta;
231                 if (!apply && fits && (!*best_fit || (avail - units + delta < *best_fit)
232                         *best_fit = avail - units + delta;
233                 }
234
235                 if (apply && fits && (!*best_fit || avail - units + delta == *best_fit)
236                         slob_t *next;
237
238                         if (delta) { /* need to fragment head to align? */
239                                 next = slob_next(cur);
240                                 set_slob(aligned, avail - delta, next);
241                                 set_slob(cur, delta, aligned);
242                                 prev = cur;
243                                 cur = aligned;
244                                 avail = slob_units(cur);
245                         }
246
247                         next = slob_next(cur);
248                         if (avail == units) { /* exact fit? unlink. */
249                                 if (prev)
250                                         set_slob(prev, slob_units(prev), next);
251                                 else
252                                         sp->freelist = next;
253                         } else { /* fragment */
254                                 if (prev)
255                                         set_slob(prev, slob_units(prev), cur + units);
256                                 else
```

```
257                                             sp->freelist = cur + units;
258                                     set_slob(cur + units, avail - units, next);
259                             }
260
261                             sp->units -= units;
262                             if (!sp->units)
263                                     clear_slob_page_free(sp);
264                             return cur;
265                     }
266                     if (slob_last(cur))
267                             return NULL;
268             }
269     }
270
271     /*
272      * slob_alloc: entry point into the slob allocator.
273      */
274     static void *slob_alloc(size_t size, gfp_t gfp, int align, int node)
275     {
276             struct page *sp;
277             struct list_head *prev;
278             struct list_head *slob_list;
279             slob_t *b = NULL;
280             unsigned long flags;
281             int best_fit = 0;
282             int apply;
283
284             if (size < SLOB_BREAK1)
285                     slob_list = &free_slob_small;
286             else if (size < SLOB_BREAK2)
287                     slob_list = &free_slob_medium;
288             else
289                     slob_list = &free_slob_large;
290
```

```
291         spin_lock_irqsave(&slob_lock, flags);
292         /* Iterate through each partially free page, try to find room */
293         for(apply = 0; apply <= 1; apply++) {
294                 list_for_each_entry(sp, slob_list, list) {
295         #ifdef CONFIG_NUMA
296                         /*
297                          * If there's a node specification, search for a partial
298                          * page with a matching node id in the freelist.
299                          */
300                         if (node != NUMA_NO_NODE && page_to_nid(sp) != node)
301                                 continue;
302         #endif
303                         /* Enough room on this page? */
304                         if (sp->units < SLOB_UNITS(size))
305                                 continue;
306
307                         /* Attempt to alloc */
308                         prev = sp->list.prev;
309                         b = slob_page_alloc(sp, size, align, apply, &best_fit);
310                         if (!b)
311                                 continue;
312                         break;
313                 }
314         }
315         spin_unlock_irqrestore(&slob_lock, flags);
316
317         /* Not enough space: must allocate a new page */
318         if (!b) {
319                 b = slob_new_pages(gfp & ~__GFP_ZERO, 0, node);
320                 if (!b)
321                         return NULL;
322                 sp = virt_to_page(b);
323                 __SetPageSlab(sp);
324
```

```
325                     spin_lock_irqsave(&slob_lock, flags);
326                     sp->units = SLOB_UNITS(PAGE_SIZE);
327                     sp->freelist = b;
328                     INIT_LIST_HEAD(&sp->list);
329                     set_slob(b, SLOB_UNITS(PAGE_SIZE), b + SLOB_UNITS(PAGE_SIZE));
330                     set_slob_page_free(sp, slob_list);
331                     best_fit = 0;
332                     apply = 1;
333                     b = slob_page_alloc(sp, size, align, apply, &best_fit);
334                     BUG_ON(!b);
335                     spin_unlock_irqrestore(&slob_lock, flags);
336             }
337             if (unlikely((gfp & __GFP_ZERO) && b))
338                     memset(b, 0, size);
339             return b;
340     }
341
342     /*
343      * slob_free: entry point into the slob allocator.
344      */
345     static void slob_free(void *block, int size)
346     {
347             struct page *sp;
348             slob_t *prev, *next, *b = (slob_t *)block;
349             slobidx_t units;
350             unsigned long flags;
351             struct list_head *slob_list;
352
353             if (unlikely(ZERO_OR_NULL_PTR(block)))
354                     return;
355             BUG_ON(!size);
356
357             sp = virt_to_page(block);
358             units = SLOB_UNITS(size);
```

```
359
360             spin_lock_irqsave(&slob_lock, flags);
361
362             if (sp->units + units == SLOB_UNITS(PAGE_SIZE)) {
363                     /* Go directly to page allocator. Do not pass slob allocator */
364                     if (slob_page_free(sp))
365                             clear_slob_page_free(sp);
366                     spin_unlock_irqrestore(&slob_lock, flags);
367                     __ClearPageSlab(sp);
368                     page_mapcount_reset(sp);
369                     slob_free_pages(b, 0);
370                     return;
371             }
372
373             if (!slob_page_free(sp)) {
374                     /* This slob page is about to become partially free. Easy! */
375                     sp->units = units;
376                     sp->freelist = b;
377                     set_slob(b, units,
378                             (void *)((unsigned long)(b +
379                                             SLOB_UNITS(PAGE_SIZE)) & PAGE_MASK));
380                     if (size < SLOB_BREAK1)
381                             slob_list = &free_slob_small;
382                     else if (size < SLOB_BREAK2)
383                             slob_list = &free_slob_medium;
384                     else
385                             slob_list = &free_slob_large;
386                     set_slob_page_free(sp, slob_list);
387                     goto out;
388             }
389
390             /*
391              * Otherwise the page is already partially free, so find reinsertion
392              * point.
```

```
393             */
394             sp->units += units;
395
396             if (b < (slob_t *)sp->freelist) {
397                     if (b + units == sp->freelist) {
398                             units += slob_units(sp->freelist);
399                             sp->freelist = slob_next(sp->freelist);
400                     }
401                     set_slob(b, units, sp->freelist);
402                     sp->freelist = b;
403             } else {
404                     prev = sp->freelist;
405                     next = slob_next(prev);
406                     while (b > next) {
407                             prev = next;
408                             next = slob_next(prev);
409                     }
410
411                     if (!slob_last(prev) && b + units == next) {
412                             units += slob_units(next);
413                             set_slob(b, units, slob_next(next));
414                     } else
415                             set_slob(b, units, next);
416
417                     if (prev + slob_units(prev) == b) {
418                             units = slob_units(b) + slob_units(prev);
419                             set_slob(prev, units, slob_next(b));
420                     } else
421                             set_slob(prev, slob_units(prev), b);
422             }
423     out:
424             spin_unlock_irqrestore(&slob_lock, flags);
425     }
426
```

```
427 /*
428  * End of slob allocator proper. Begin kmem_cache_alloc and kmalloc frontend.
429  */
430
431 static __always_inline void *
432 __do_kmalloc_node(size_t size, gfp_t gfp, int node, unsigned long caller)
433 {
434         unsigned int *m;
435         int align = max_t(size_t, ARCH_KMALLOC_MINALIGN, ARCH_SLAB_MINALIGN);
436         void *ret;
437
438         gfp &= gfp_allowed_mask;
439
440         lockdep_trace_alloc(gfp);
441
442         if (size < PAGE_SIZE - align) {
443                 if (!size)
444                         return ZERO_SIZE_PTR;
445
446                 m = slob_alloc(size + align, gfp, align, node);
447
448                 if (!m)
449                         return NULL;
450                 *m = size;
451                 ret = (void *)m + align;
452
453                 trace_kmalloc_node(caller, ret,
454                                    size, size + align, gfp, node);
455         } else {
456                 unsigned int order = get_order(size);
457
458                 if (likely(order))
459                         gfp |= __GFP_COMP;
460                 ret = slob_new_pages(gfp, order, node);
```

```
461
462                 trace_kmalloc_node(caller, ret,
463                                    size, PAGE_SIZE << order, gfp, node);
464         }
465
466         kmemleak_alloc(ret, size, 1, gfp);
467         return ret;
468 }
469
470 void *__kmalloc(size_t size, gfp_t gfp)
471 {
472         return __do_kmalloc_node(size, gfp, NUMA_NO_NODE, _RET_IP_);
473 }
474 EXPORT_SYMBOL(__kmalloc);
475
476 #ifdef CONFIG_TRACING
477 void *__kmalloc_track_caller(size_t size, gfp_t gfp, unsigned long caller)
478 {
479         return __do_kmalloc_node(size, gfp, NUMA_NO_NODE, caller);
480 }
481
482 #ifdef CONFIG_NUMA
483 void *__kmalloc_node_track_caller(size_t size, gfp_t gfp,
484                                   int node, unsigned long caller)
485 {
486         return __do_kmalloc_node(size, gfp, node, caller);
487 }
488 #endif
489 #endif
490
491 void kfree(const void *block)
492 {
493         struct page *sp;
494
```

```
495         trace_kfree(_RET_IP_, block);
496
497         if (unlikely(ZERO_OR_NULL_PTR(block)))
498                 return;
499         kmemleak_free(block);
500
501         sp = virt_to_page(block);
502         if (PageSlab(sp)) {
503                 int align = max_t(size_t, ARCH_KMALLOC_MINALIGN, ARCH_SLAB_MINALIGN);
504                 unsigned int *m = (unsigned int *)(block - align);
505                 slob_free(m, *m + align);
506         } else
507                 __free_pages(sp, compound_order(sp));
508 }
509 EXPORT_SYMBOL(kfree);
510
511 /* can't use ksize for kmem_cache_alloc memory, only kmalloc */
512 size_t ksize(const void *block)
513 {
514         struct page *sp;
515         int align;
516         unsigned int *m;
517
518         BUG_ON(!block);
519         if (unlikely(block == ZERO_SIZE_PTR))
520                 return 0;
521
522         sp = virt_to_page(block);
523         if (unlikely(!PageSlab(sp)))
524                 return PAGE_SIZE << compound_order(sp);
525
526         align = max_t(size_t, ARCH_KMALLOC_MINALIGN, ARCH_SLAB_MINALIGN);
527         m = (unsigned int *)(block - align);
528         return SLOB_UNITS(*m) * SLOB_UNIT;
```

```
529 }
530 EXPORT_SYMBOL(ksize);
531
532 int __kmem_cache_create(struct kmem_cache *c, unsigned long flags)
533 {
534         if (flags & SLAB_DESTROY_BY_RCU) {
535                 /* leave room for rcu footer at the end of object */
536                 c->size += sizeof(struct slob_rcu);
537         }
538         c->flags = flags;
539         return 0;
540 }
541
542 void *slob_alloc_node(struct kmem_cache *c, gfp_t flags, int node)
543 {
544         void *b;
545
546         flags &= gfp_allowed_mask;
547
548         lockdep_trace_alloc(flags);
549
550         if (c->size < PAGE_SIZE) {
551                 b = slob_alloc(c->size, flags, c->align, node);
552                 trace_kmem_cache_alloc_node(_RET_IP_, b, c->object_size,
553                                             SLOB_UNITS(c->size) * SLOB_UNIT,
554                                             flags, node);
555         } else {
556                 b = slob_new_pages(flags, get_order(c->size), node);
557                 trace_kmem_cache_alloc_node(_RET_IP_, b, c->object_size,
558                                             PAGE_SIZE << get_order(c->size),
559                                             flags, node);
560         }
561
562         if (b && c->ctor)
```

```
563                     c->ctor(b);
564
565             kmemleak_alloc_recursive(b, c->size, 1, c->flags, flags);
566             return b;
567     }
568     EXPORT_SYMBOL(slob_alloc_node);
569
570     void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
571     {
572             return slob_alloc_node(cachep, flags, NUMA_NO_NODE);
573     }
574     EXPORT_SYMBOL(kmem_cache_alloc);
575
576     #ifdef CONFIG_NUMA
577     void *__kmalloc_node(size_t size, gfp_t gfp, int node)
578     {
579             return __do_kmalloc_node(size, gfp, node, _RET_IP_);
580     }
581     EXPORT_SYMBOL(__kmalloc_node);
582
583     void *kmem_cache_alloc_node(struct kmem_cache *cachep, gfp_t gfp, int node)
584     {
585             return slob_alloc_node(cachep, gfp, node);
586     }
587     EXPORT_SYMBOL(kmem_cache_alloc_node);
588     #endif
589
590     static void __kmem_cache_free(void *b, int size)
591     {
592             if (size < PAGE_SIZE)
593                     slob_free(b, size);
594             else
595                     slob_free_pages(b, get_order(size));
596     }
```

```
597
598 static void kmem_rcu_free(struct rcu_head *head)
599 {
600         struct slob_rcu *slob_rcu = (struct slob_rcu *)head;
601         void *b = (void *)slob_rcu - (slob_rcu->size - sizeof(struct slob_rcu));
602
603         __kmem_cache_free(b, slob_rcu->size);
604 }
605
606 void kmem_cache_free(struct kmem_cache *c, void *b)
607 {
608         kmemleak_free_recursive(b, c->flags);
609         if (unlikely(c->flags & SLAB_DESTROY_BY_RCU)) {
610                 struct slob_rcu *slob_rcu;
611                 slob_rcu = b + (c->size - sizeof(struct slob_rcu));
612                 slob_rcu->size = c->size;
613                 call_rcu(&slob_rcu->head, kmem_rcu_free);
614         } else {
615                 __kmem_cache_free(b, c->size);
616         }
617
618         trace_kmem_cache_free(_RET_IP_, b);
619 }
620 EXPORT_SYMBOL(kmem_cache_free);
621
622 int __kmem_cache_shutdown(struct kmem_cache *c)
623 {
624         /* No way to check for remaining objects */
625         return 0;
626 }
627
628 int kmem_cache_shrink(struct kmem_cache *d)
629 {
630         return 0;
```

```
631 }
632 EXPORT_SYMBOL(kmem_cache_shrink);
633
634 struct kmem_cache kmem_cache_boot = {
635         .name = "kmem_cache",
636         .size = sizeof(struct kmem_cache),
637         .flags = SLAB_PANIC,
638         .align = ARCH_KMALLOC_MINALIGN,
639 };
640
641 void __init kmem_cache_init(void)
642 {
643         kmem_cache = &kmem_cache_boot;
644         slab_state = UP;
645 }
646
647 void __init kmem_cache_init_late(void)
648 {
649         slab_state = FULL;
650 }
```

B. frag.py

```
1  #!/usr/bin/python
2  import sys
3
4  mult_list = [
5   4,
6   8,
7   16,
8   32,
9   64,
10  128,
11  256,
12  512,
```

```
 1024,
 2048,
 4096
]

example_data = "Node 0, zone   Normal     75     51     57     16      2      0      0
example_data_2 = "Normal    272    144     80     50     33     13      2      1      2

def get_number_list(input_string):
        return (int(i) for i in input_string.split("zone   Normal")[1].split())

def get_fragmentation(input_blocks):
        return (sum(input_blocks) - input_blocks[-1]) / float(sum(input_blocks))

def size_blocks(number_list):
        for block, size in zip(number_list, mult_list):
                yield block*size

data = open("/proc/buddyinfo", "r").read()
print(get_fragmentation(list(size_blocks(get_number_list(data)))))
```