ID2200 – Laboration 3: Malloc

Sam Henriksson	Elvis Stansvik
hensa@kth.se	stansvik@kth.se
910530-2773	831205-3971

23 maj 2014

Innehåll

1 Inledning

En minnesallokerare har till uppgift att på programs begäran dynamiskt allokera minne från operativsystemet, och att frigöra det på ett sådant sätt att det kan återanvändas vid senare minnesbegäran. En minnesallokerare abstraherar de operativsystemspecifika mekanismer som finns för att begära minne. I denna korta rapport beskriver vi vår implementation av en minneshanterare för UNIX-system, samt resultatet från en utvärdering av allokerarens prestanda.

2 Problembeskrivning

Uppgiften bestod i att skriva en minnesallokerare som tillhandahåller funktionerna `malloc`, `free` och `realloc`. Funktionerna skulle följa det gränssnitt som specificeras av ISO-standarden [1, s. 154–155]. Vi skulle implementera åtminstone två strategier för allokering. Som utgångspunkt tilläts vi använda koden från avsnitt 8.7 i Kernighan och Richies *The C Programming Language* [2, s. 185-189].

3 Lösning

Vi började med att bygga en enkellänkad datastruktur `header_t` för att representera block av tillgängligt eller allokerat minne. Datastrukturen innehåller en pekare `next` till nästa block och ett fält `size` som anger blockets storlek räknat i antal headerstorlekar (inklusive headern själv).

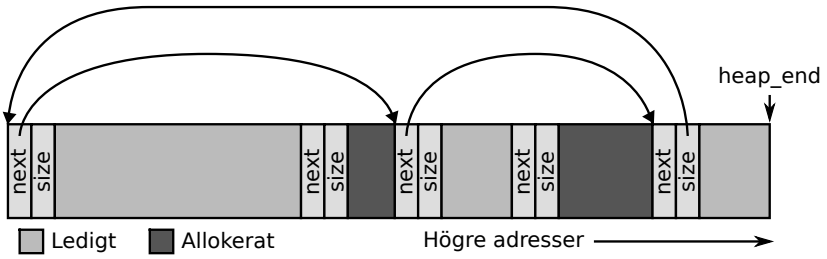
Datastrukturen bäddades in i en union med ett dummyfält av datatypen `long`, som är den mest restriktiva typen i C. Detta för att tvinga strukturen att hamna på addresser som är jämna multipler av `sizeof(long)`, s.k. alignment.

Minnesallokeraren håller koll på en lista av tillgängliga block. Initialt är listan tom. När användaren begär mer minne med funktionen `malloc` letas listan igenom efter ett block av tillräcklig storlek. Valet av block görs enligt en viss sökstrategi. Sökningen påbörjas där den senaste sökningen avslutades, detta för att allokeraren ska få ett homogent beteende och inte föredra att ta utrymme från block på låga adresser.

Hittas inget block av tillräcklig storlek så anropar `malloc` den interna funktionen `add_memory`, som begär mer minne (minst 1024 * headerstorleken) från operativsystemet genom att anropa `mmap` och lägger till det nya minnet i listan genom att anropa `free`.

Minnesutrymmet som begärs med `mmap` tas från slutet av heapen (pekaren `heap_end`), som efter en begäran flyttas fram med den begärda storleken. För att hitta slutet på heapen görs vid första användandet av `heap_end` ett anrop till `sbrk(2)` med 0 som parameter. Hade detta anrop inte gjorts skulle `heap_end`, eftersom den initialiseras till 0, pekat på början av det virtuella minnesområdet. Den begärda storleken rundas upp till en multipel av sidstorleken innan `mmap` anropas, eftersom `mmap` bara kan mappa in ett helt antal sidor. Hade `NULL`

skickats in till `mmap` istället för `heap_end` så hade operativsystemet varit fritt att mappa in utrymmet var helst det finner bäst i det virtuella adressområdet.



Figur 1: Schematisk illustration av blockens positioner i minnet.

I anropet till `mmap` anges flaggan `MAP_SHARED` vilket medför att skrivningar till det inmappade minnet blir synliga för andra processer. Hade till exempel `MAP_PRIVATE` angivits istället så skulle skrivningarna bara varit synliga i processen från vilken `mmap` anropades.

Funktionen `free` länkar tillbaks det frigjorda minnet som ett nytt block i listan. Listan är sorterad på blockens adress i minnet för att `free` enkelt ska kunna avgöra om block kan slås samman när minne frigörs.

Funktionen `realloc` ändrar storlek på ett tidigare allokerat utrymme. Om `NULL` ges som parameter fungerar funktionen precis som `malloc`, och om 0 anges som ny storlek så frigörs blocket med `free`. I annat fall allokeras ett nytt block av den givna storleken med `malloc`, och datan från det tidigare blocket kopieras över.

Figur 1 visar en schematisk illustration av blockens positioner i minnet. Två block är allokerade. De tre lediga blocken är resultatet av tidigare allokerat minne som frigjorts med `free`.

Vi valde att implementera två strategier för allokering: “first fit”, där `malloc` väljer det första lediga block med tillräcklig storlek, och “best fit”, där `malloc` väljer det block som passar bäst (resulterar i minst spill). Valet av strategi görs genom att definiera `STRATEGY` som 1 (first fit) eller 2 (best fit). Koden för vår lösning finns i listad i Appendix A. Den finns även att ladda hem på adressen `https://github.com/estan/ID2200` och kan därefter byggas med `make`.

4 Prestandautvärdering

4.1 Körtid

För att utvärdera körtiden för vår minnesallokerare skrevs ett mindre testprogram `benchmark-block-sizes.c` som mäter den genomsnittliga tiden för ett anrop till `malloc` för ett antal blockstorlekar. För varje blockstorlek 64 B, 128 B, ..., 65 KB utför programmet 1000 på varandra följande anrop till `malloc` för att ta fram den genomsnittliga tiden för ett anrop. Programmet kördes med vår minnesallokerare med strategierna first fit och best fit, samt med systemets

`malloc`-implementation från C-biblioteket (glibc). Resultatet av körningarna visas i tabell 1 och i figur 3 finns även ett stapeldiagram över de erhållna tiderna. Körningen mot varje implementation upprepades ett antal gånger för att kontrollera att resultatet inte hade för stor varians.

Värstafallet för båda strategierna inträffar då minnesallokeraren vid varje anrop till `malloc` behöver be om nytt minne från operativsystemet och listan med tillgängliga block ökar med en post. Detta kan provoceras fram genom att upprepat begära $(1024H)/2 - H + 1$ bytes, där H är headerstorleken. Algoritmen kommer då lägga till ett block i listan som är 2 byte för litet för att det ska kunna användas vid nästa begäran. På vårt testsystem var $H = 16$ bytes, vilket ger $(1024 \cdot 16)/2 - 16 + 1 = 8177$ bytes. Vi skrev därför även ett test `benchmark-worst.c` som anropar `malloc` 5000 gånger med denna blockstorlek. Diagrammet i figur 4 visar resultatet av körningarna med first fit, best fit och systemets `malloc`.

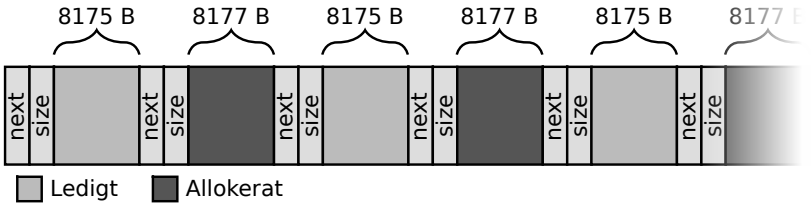
4.2 Minnesutnyttjande

Vi var även intresserade av att jämföra first fit med best fit vad gäller utnyttjande av minnet. Det vill säga hur stor andel av det minne som minnesallokeraren begär från operativsystemet användaren har till förfogande. Vi skrev därför ett testprogram `benchmark-memory.c` som försöker uppskatta detta genom att titta på hur mycket minne som har begärts av användaren gentemot hur mycket heapen har vuxit.

Programmet går in i en slinga som upprepas 10 gånger. Varje varv genom loopen skapas en ny barnprocess upp i vilken 1000 block av slumpmässig storlek 100-10000 bytes allokeras. Förflyttningen av `heap_end`-pekaren adderas till en räknare `heap_change` och det totala antalet allokerade bytes adderas till en räknare `allocated`. Programmet väntar på att barnprocessen ska bli klar innan slingan går vidare. Vid avslut skrivs `100 * allocated / heap_change` ut som en indikation på hur hur många procent av minnet som verkligen utnyttjas.

Programmet kördes med vår minnesallokerare med strategierna best fit och first. Varje körning upprepades 10 gånger och resultatet från de två sekvenserna av körningar finns listat i tabell 2.

Värstafallet med avseende på minnesutnyttjande sammanfaller med det för körtiden: Vid upprepad allokering av 8177 bytes kommer minnet få utseendet som visas i figur 2 och endast hälften av minnet utnyttjas.



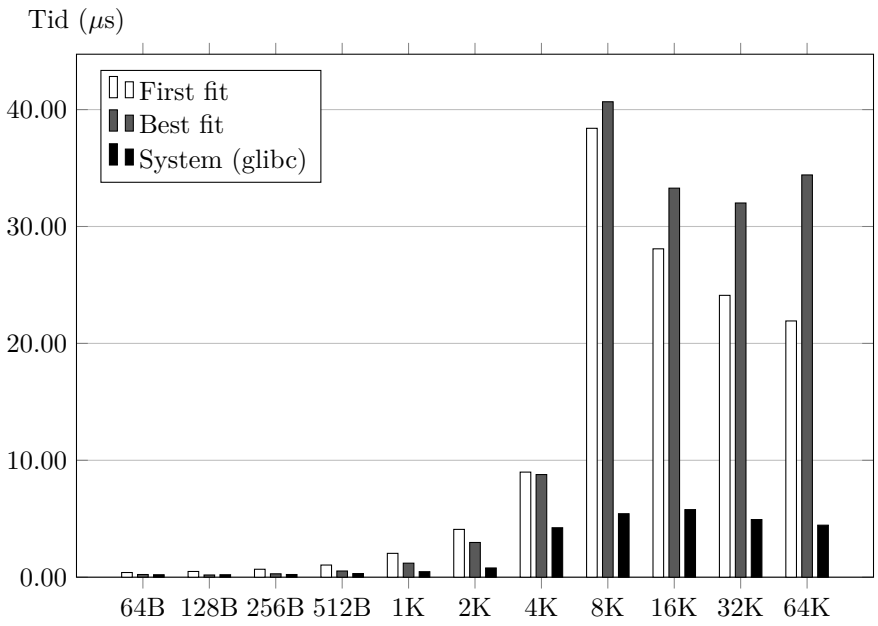
Figur 2: Resultatet av upprepad allokering av 8177 bytes.

5 Resultat

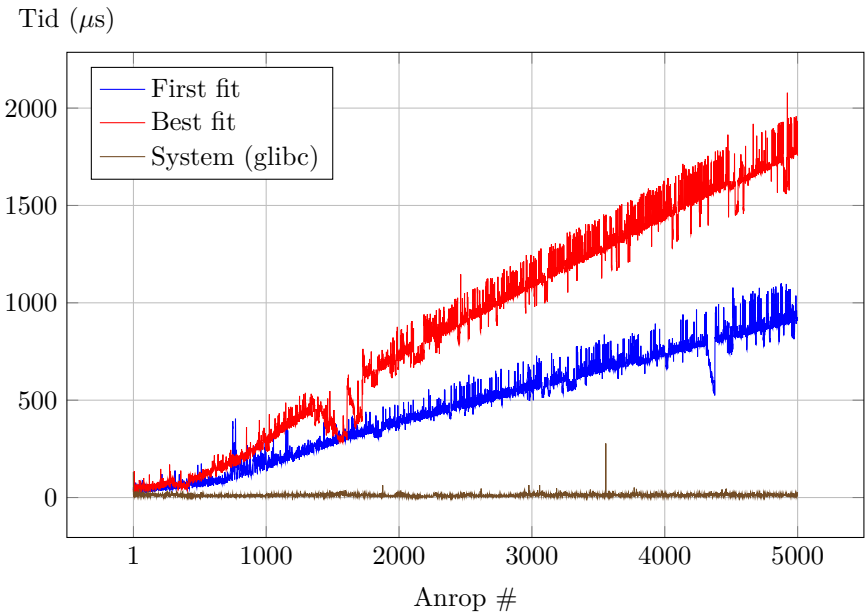
5.1 Körtid

Tabell 1: Genomsnittlig tid per anrop för 1000 anrop till `malloc`. Se figur 3 som visar ett stapeldiagram över tiderna.

Blockstorlek (B)	First fit (μs)	Best fit (μs)	System (glibc) (μs)
64	0.40	0.23	0.21
128	0.49	0.19	0.21
256	0.68	0.29	0.23
512	1.04	0.53	0.31
1024	2.04	1.20	0.47
2048	4.09	2.97	0.79
4096	8.99	8.78	4.23
8192	38.40	40.67	5.43
16384	28.09	33.28	5.78
32768	24.11	32.01	4.93
65536	21.92	34.41	4.45



Figur 3: Genomsnittlig tid per anrop för 1000 anrop till `malloc`. Värdena är hämtade ur tabell 1.



Figur 4: Tid för 5000 anrop till `malloc` med 8177 B som blockstorlek. Storleken 8177 B är att betrakta som värstafall för algoritmerna på maskinen där testet kördes.

5.2 Minnesutnyttjande

Tabell 2: Utnyttjat minne som procent av den ifrån operativsystemet begärda mängden vid allokering av 1000 block av slumpvis storlek 100–10000 B.

Körning #	First fit (%)	Best fit (%)
1	93.69	97.96
2	93.64	97.76
3	93.93	97.79
4	93.82	97.70
5	93.59	97.57
6	93.83	97.80
7	93.48	97.71
8	93.67	97.81
9	93.73	97.74
10	93.94	97.65

6 Diskussion

Resultatet från körningen av `benchmark-block-sizes.c` (figur 3) visar att vid små blockstorlekar (64-128 B) håller de två implementerade algoritmerna relativt jämna steg med systemets inbyggda `malloc`. Sedan ökar körtiderna gradvis allteftersom blockstorleken närmar sig den punkt (8177 bytes) där allokeraren kommer behöva be operativsystemet om minne vid varje anrop, och det är sedan tydligt att den systemets `malloc` med sina mer avancerade datastrukturer är betydligt snabbare. Bakgrunden till valet av blockstorlekar i detta test är att vi monitorerade minnesallokeringar för ett antal program som `find`, `ls` och en grafisk kalkylator (`kcalc`) med hjälp av `malloc_hook(2)` för att se vilka blockstorlekar som var vanligast.

Skillnaden i körtid jämfört med systemets `malloc` är ännu tydligare i resultatet från körningen av `benchmark-worst.c` (figur 4) där systemets `malloc` totalt dominerar de två algoritmerna när antalet allokeringar blir stort. Algoritmernas körtid ökar linjärt med antalet tidigare allokeringar, medan körtiden för systemets `malloc` hålls konstant. Det är här också tydligt att first fit-strategin är snabbare än best fit, som vid varje anrop går igenom hela listan av tillgängliga block.

När det gäller minnesutnyttjande visar resultatet från körningarna av testet `benchmark-memory.c` (tabell 2) att best fit som väntat utnyttjar minnet något bättre än first fit. Körningarna med first fit uppvisade ett genomsnittligt utnyttjande på 93.73% där standardfelet är ± 0.12 på säkerhetsnivån 99%, medan körningarna med best fit gav ett genomsnittligt nyttjande på 97.75% och standardfel ± 0.06 på samma säkerhetsnivå. Skillnaden är alltså inte så stor (3-4%), men statistiskt säkerställd.

7 Slutsatser

Våra implementationer av `malloc` kan möjligtvis vara till användning i små program där antalet allokeringar och blockstorlekarna som allokeras är små. Implementationerna är enkla och består av få rader kod, vilket kan vara en fördel på små inbäddade system där programmet som ska köras ska få plats på ett litet ROM-minne. Om kort körtid är av stor vikt bör first fit-strategin användas, medan best-fit eventuellt lämpar sig bättre om effektivt utnyttjande av RAM-minnet är viktigt.

För fullskaliga program för persondatorer och servrar står det dock klart att standardimplementationen av `malloc` i C-biblioteket är att föredra: På grund av dess mer avancerade datastrukturer och kraftfullare algoritmer ger den mycket bättre prestanda när antalet allokeringar är stort.

8 Övrigt

8.1 Tidsuppskattning

Vi uppskattar att vi lagt cirka 40 timmar på programmering och testning och 20 timmar på arbete med rapporten, så totalt cirka 60 timmar nedlagt arbete.

8.2 Betygssättning av lab-PM

Lab-PM är bra utformat med många detaljer. Vi ger det betyget 4/5. Några förslag på förbättringar ges nedan.

8.3 Förslag på förbättringar

- LAB-PM SID 1: "[...] . med `mmap(2)` [...]" → "[...] . Med `mmap(2)` [...]" (stor bokstav).
- LAB-PM SID 7: Tag bort " - därför finns den heller inte upplagd i någon kurskatalog." (den finns där nu).
- KURSKATALOGEN: Lägg till högst upp i `malloc.c`:

  ```
  #ifdef __linux__
  #define _GNU_SOURCE /* For MAP_ANONYMOUS on Linux */
  #endif /* __linux__ */
  ```

  Detta eftersom `MAP_ANONYMOUS` (som är icke-standard) inte finns tillgängligt på Linux utan att `_BSD_SOURCE`, `_SVID_SOURCE` eller `_GNU_SOURCE` är definierat när `sys/mman.h` inkluderas, vilket i nuläget resulterar ett kompileringsfel.
- KURSKATALOGEN: Ändra datatypen för `highbreak` och `lowbreak` från `caddr_t` till `void*` i `tstcrash_simple.c` och `tstcrash_complex.c`. Datatypen `caddr_t` är en gammal BSD-ism som inte längre används och `sbrk(2)` returnerar numera `void*`.
- KURSKATALOGEN: Skriv ut addresserna (`highbreak` och `lowbreak`) som `unsigned long` (`%lx`) istället för `unsigned int` (`%x`) i `tstcrash_simple.c` och `tstcrash_complex.c` för att undvika varningar på 64-bitarssystem.
- KURSKATALOGEN: Lägg till

  ```
  #pragma GCC diagnostic ignored "-Wfree-nonheap-object"
  ```

  i `tstcrash_simple.c` och `tstcrash_complex.c` för att ignorera varning om `free(2)` på icke-heapallokerad data. Detta är en del av testet och varningen kan således ignoreras.

- KURSKATALOGEN: Inkludera `stdio.h` i `malloc.c` för att undvika varning om implicit deklarerad `perror(3)`.
- KURSKATALOGEN: Skriptet `RUN_TESTS` deklarerar sig som CSH men använder `read`, som inte är ett inbyggt kommando i CSH, för att läsa data. Skriptet måste ha skrivits på ett system där `read` finns som en separat kommando i `/usr/bin`. Vi föreslår att skriptet helt enkelt ändras till att deklarera sig som ett Bourne-shellskript med `#!/bin/sh` eftersom det följer den syntaxen och då kommer fungera på system utan kommandot `read` (de flesta Linux-system). Alternativt kan `$<` användas för inläsning (som är giltig CSH).

Referenser

[1] ISO. *C90 Standard*. ISO/IEC 9899:1990. 1990.

[2] Brian W. Kernighan och Dennis Ritchie. *C Programming Language*. 2. utg. Prentice Hall, 1988. ISBN: 978-0-13-308621-8.

Appendix A: Källkod

Källkod 1: `malloc.h`

```
#ifndef MALLOC_H
#define MALLOC_H

void *malloc(size_t);
void free(void *);
void *realloc(void *, size_t);

#endif /* MALLOC_H */
```

Källkod 2: `malloc.c`

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>

#include "malloc.h"

#define MIN_UNITS 1024

/* The most restrictive type of the machine. */
typedef long align_t;

/* Block header data structure. */
typedef union header_u {
    struct {
        union header_u *next; /* Next free block list. */
        size_t size;          /* Size of block, including header. */
    } s;
    align_t dummy; /* Force alignment to align_t boundaries. */
} header_t;

static header_t base = { { &base, 0 } }; /* Base of free list. */
static header_t *freep = &base;          /* Free block pointer. */
static void *heap_end = 0;               /* End of heap. */

/* Returns a pointer to the end of the heap. */
void *endHeap() {
    if (heap_end == 0)
        heap_end = sbrk(0);
    return heap_end;
}

size_t min(size_t a, size_t b) {
    return a < b ? a : b;
}

```

```
38 /*
39  * Adds a new block of least min(nunits, MIN_UNITS) * sizeof(header_t) bytes
40  * of memory to the free list. Returns a pointer to the added block, or NULL
41  * if the request failed.
42  */
43 static header_t *add_memory(size_t nunits) {
44     const void *mem;
45     header_t *header;
46     size_t npages;
47 
48     const static int flags = MAP_SHARED | MAP_ANONYMOUS;
49     const static int protection = PROT_READ | PROT_WRITE;
50     const long page_size = sysconf(_SC_PAGESIZE);
51 
52     if (nunits < MIN_UNITS)
53         nunits = MIN_UNITS;
54 
55     if (heap_end == 0)
56         heap_end = sbrk(0);
57 
58     npages = (nunits * sizeof(header_t) - 1) / page_size + 1;
59     mem = mmap(heap_end, npages * page_size, protection, flags, -1, 0);
60     if (mem == MAP_FAILED) {
61         perror("mmap");
62         return NULL;
63     }
64     nunits = (npages * page_size) / sizeof(header_t);
65     heap_end += npages * page_size;
66     header = (header_t *)mem;
67     header->s.size = nunits;
68     free(header + 1);
69     return freep;
70 }
71 
72 /* Allocates size bytes and return a pointer to the allocated memory. */
73 void *malloc(size_t size) {
74     header_t *p, *prevp;
75 #if STRATEGY == 2
76     header_t *best = NULL;
77     size_t diff = (size_t)-1;
78 #endif
79 
80     /* Calculate number of header-sized units required. */
81     size_t nunits = (size + sizeof(header_t) - 1) / sizeof(header_t) + 1;
82 
83     if (size == 0)
84         return NULL; /* No-op */
85 
86 #if STRATEGY == 1
87     for (prevp = freep, p = prevp->s.next; ; prevp = p, p = p->s.next) {
88         if (p->s.size >= nunits) {
89             /* p is big enough. */
```

```
90              if (p->s.size == nunits) {
91                  /* p is exactly big enough, remove from free list. */
92                  prevp->s.next = p->s.next;
93              } else {
94                  /* p is more than big enough, allocate from tail end. */
95                  p->s.size -= nunits;
96                  p += p->s.size;
97                  p->s.size = nunits;
98              }
99              freep = prevp;
100             return (void *)(p + 1);
101         }
102         if (p == freep) {
103             /* No suitably large block found. */
104             if ((p = add_memory(nunits)) == NULL) {
105                 /* Out of memory */
106                 return NULL;
107             }
108         }
109     }
110 #elif STRATEGY == 2
111     for (prevp = freep, p = prevp->s.next; ; prevp = p, p = p->s.next) {
112         if (p->s.size >= nunits) {
113             if (p->s.size == nunits) {
114                 prevp->s.next = p->s.next;
115                 freep = prevp;
116                 return (void *)(p + 1);
117             }
118             if (p->s.size - nunits < diff) {
119                 best = p;
120                 diff = p->s.size - nunits;
121             }
122         }
123         if (p == freep) {
124             if (best == NULL) {
125                 /* No suitably large block found. */
126                 if ((p = add_memory(nunits)) == NULL) {
127                     /* Out of memory */
128                     return NULL;
129                 }
130             } else {
131                 /* Allocate from tail end */
132                 best->s.size -= nunits;
133                 best += best->s.size;
134                 best->s.size = nunits;
135                 return (void *)(best + 1);
136             }
137         }
138     }
139 #endif
140 }
141
```

```
142 /* Frees the memory pointed to by ptr. */
143 void free(void *ptr) {
144     header_t *freed, *p;
145 
146     if (ptr == NULL) {
147         return; /* No-op */
148     }
149 
150     freed = (header_t *) ptr - 1; /* Point to block header. */
151 
152     /* Find insertion point p for bp. */
153     for (p = freep; !(p < freed && freed < p->s.next); p = p->s.next) {
154         if (p >= p->s.next && (p < freed || freed < p->s.next)) {
155             /* Insertion point is either start or end of list. */
156             break;
157         }
158     }
159     if (freed + freed->s.size == p->s.next) {
160         /* Merge with upper neighbor. */
161         freed->s.size += p->s.next->s.size;
162         freed->s.next = p->s.next->s.next;
163     } else {
164         freed->s.next = p->s.next;
165     }
166     if (p + p->s.size == freed) {
167         /* Merge with lower neighbor. */
168         p->s.size += freed->s.size;
169         p->s.next = freed->s.next;
170     } else {
171         p->s.next = freed;
172     }
173     freep = p;
174 }
175 
176 /* Change size of memory block pointer to by ptr to size bytes. */
177 void *realloc(void *ptr, size_t size) {
178     header_t *old_header;
179     size_t old_size;
180     void *new_ptr;
181 
182     if (ptr == NULL) {
183         return malloc(size);
184     }
185     if (size == 0) {
186         free(ptr);
187         return NULL;
188     }
189 
190     /* Get old header and size. */
191     old_header = (header_t *) ptr - 1;
192     old_size = (old_header->s.size - 1) * sizeof(header_t);
193 
```

```
194     if (size == old_size) {
195         return ptr; /* No-op */
196     }
197
198     /* Allocate new memory, copy contents and free old memory. */
199     if ((new_ptr = malloc(size)) == NULL) {
200         return NULL;
201     }
202     memcpy(new_ptr, ptr, size < old_size ? size : old_size);
203     free(ptr);
204
205     return new_ptr;
206 }
```

Källkod 3: benchmark-block-sizes.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <time.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7
8  #include "timer.h"
9  #include "malloc.h"
10
11 #define NUM_ALLOCS 1000
12 #define NUM_BLOCK_SIZES 11
13
14 int main(int argc, char *argv[]) {
15     int i;
16
17     printf("blocksize,avgtime\n");
18     for (i = 0; i < NUM_BLOCK_SIZES; ++i) {
19         pid_t pid = fork();
20         if (pid == -1) {
21             perror("fork");
22             return EXIT_FAILURE;
23         } else if (pid == 0) {
24             int j;
25             double avg_time = 0;
26             void *blocks[NUM_ALLOCS];
27             size_t block_sizes[NUM_BLOCK_SIZES] = {
28                 64,   128,  256,  512,   1024,  2048,
29                 4096, 8192, 16384, 32768, 65536
30             };
31
32             malloc(42); /* Get things into cache */
33
34             /* Measure allocations */
35             for (j = 0; j < NUM_ALLOCS; ++j) {
36                 TIMER_START();
```

```
37                  blocks[j] = malloc(block_sizes[i]);
38                  TIMER_STOP();
39                  avg_time += TIMER_ELAPSED_US();
40              }
41              avg_time /= NUM_ALLOCS;
42              printf("%lu,%.2f\n", block_sizes[i], avg_time);
43
44              /* Free allocations */
45              for (j = 0; j < NUM_ALLOCS; ++j) {
46                  free(blocks[j]);
47              }
48
49              exit(EXIT_SUCCESS);
50          } else {
51              int status;
52              waitpid(pid, &status, 0);
53          }
54      }
55
56      return EXIT_SUCCESS;
57  }
```

Källkod 4: `benchmark-worst.c`

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <time.h>
4   #include <unistd.h>
5
6   #include "timer.h"
7   #include "malloc.h"
8
9   #define NUM_BLOCKS 5000
10
11  int main(int argc, char *argv[]) {
12      int i;
13      void *blocks[NUM_BLOCKS];
14      srand(time(0));
15
16      printf("op_nr,time\n");
17
18      /* Measure allocation time */
19      for (i = 0; i < NUM_BLOCKS; ++i) {
20          usleep(rand() % 1000);
21          TIMER_START();
22          blocks[i] = malloc(8177);
23          TIMER_STOP();
24          printf("%d,%.2f\n", i, TIMER_ELAPSED_US());
25      }
26
27      /* Free blocks */
28      for (i = 0; i < NUM_BLOCKS; ++i) {
```

```
        free(blocks[i]);
    }

    return EXIT_SUCCESS;
}
```

Källkod 5: benchmark-memory.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <stddef.h>

#include "brk.h"
#include "malloc.h"

#define NUM_REPS 10
#define NUM_ALLOCS 1000

#define MIN_SIZE 100
#define MAX_SIZE 10000

int main(int argc, char *argv[]) {
    int i;

    int prot = PROT_READ | PROT_WRITE;
    int flags = MAP_SHARED | MAP_ANONYMOUS;

    ptrdiff_t *heap_change = mmap(NULL, sizeof(ptrdiff_t), prot, flags, -1, 0);
    size_t *allocated = mmap(NULL, sizeof(size_t), prot, flags, -1, 0);

    *heap_change = 0;
    *allocated = 0;

    for (i = 0; i < NUM_REPS; ++i) {
        pid_t pid = fork();
        if (pid == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        } else if (pid == 0) {
            int j, diff;
            size_t size[NUM_ALLOCS];
            char *block[NUM_ALLOCS];
            void *heap_before;

            srand(time(NULL) ^ (getpid() << 16));

            /* Generate random block sizes */
```

```
45              for (j = 0; j < NUM_ALLOCS; ++j) {
46                  diff = MAX_SIZE - MIN_SIZE;
47                  size[j] = MIN_SIZE + (diff == 0 ? 0 : (rand() % diff));
48              }
49
50              /* Allocate blocks and measure heap change */
51              heap_before = endHeap();
52              for (j = 0; j < NUM_ALLOCS; ++j) {
53                  block[j] = malloc(size[j]);
54                  *allocated += size[j];
55              }
56              *heap_change += endHeap() - heap_before;
57
58              /* Free allocated blocks */
59              for (j = 0; j < NUM_ALLOCS; ++j) {
60                  free(block[j]);
61              }
62
63              exit(EXIT_SUCCESS);
64          } else {
65              int status;
66              waitpid(pid, &status, 0);
67          }
68      }
69
70      /* Print the overall memory usage in percent */
71      printf("%.2f\n", 100.0 * *allocated / *heap_change);
72
73      return EXIT_SUCCESS;
74  }
```

Källkod 6: `timer.h`

```
1   #ifndef TIMER_H
2   #define TIMER_H
3
4   struct timespec t0, t1; /* Start/end time for timer */
5
6   /* Timer macros */
7   #define TIMER_START() clock_gettime(CLOCK_MONOTONIC, &t0)
8   #define TIMER_STOP() clock_gettime(CLOCK_MONOTONIC, &t1)
9   #define TIMER_ELAPSED_NS() \
10      (t1.tv_sec * 1000000000 + t1.tv_nsec) - \
11      (t0.tv_sec * 1000000000 + t0.tv_nsec)
12  #define TIMER_ELAPSED_US() \
13      (t1.tv_sec * 1000000 + (double)t1.tv_nsec / 1000) - \
14      (t0.tv_sec * 1000000 + (double)t0.tv_nsec / 1000)
15
16  #endif /* TIMER_H */
```