IN-LAB- 7

Allocating Memory on the Heap

A process can allocate memory by increasing the size of the heap, a variable size segment of contiguous virtual memory that begins just after the uninitialized data segment of a process and grows and shrinks as memory is allocated and freed. The current limit of the heap is referred to as the program break.To allocate memory, C programs normally use the malloc family of functions, which we describe shortly. However, we begin with a description of brk() and sbrk(), upon which the malloc functions are based

Adjusting the Program Break: brk() and sbrk()

Resizing the heap (i.e., allocating or deallocating memory) is actually as simple as telling the kernel to adjust its idea ofwhere the process’s program break is. Initially, the program break lies just past the end of the uninitialized data segment (i.e., the location where heap starts above bss.After the program break is increased, the program may access any address in the newly allocated area, but no physical memory pages are allocated yet. The kernel automatically allocates new physical pages on the first attempt by the process toaccess addresses in those pages.Traditionally, the UNIX system has provided two system calls for manipulating the program break, and these are both availableon Linux: brk() and sbrk(). Although these system calls are seldom used directly in programs, understanding them helps clarify how memory allocation works.

#include <unistd.h>int brk(void \*end\_data\_segment);

Returns 0 on success, or –1 on errorvoid \*sbrk(intptr\_tincrement);

Returns previous program break on success, or (void \*) –1 on error

The brk() system call sets the program break to the location specified by end\_data\_segment. Since virtual memory is allocated in units of pages, end\_data\_segment is effectivelyrounded up to the next page boundary. Attempts to set the program break below its initial value (i.e., below &end) are likely to result in unexpected behavior, such as a segmentation fault (the SIGSEGV signal) when trying to access data in now nonexistentparts of the initialized or uninitialized data segments. The precise upper limit on where the program break can be set depends on a range of factors, including: the process resource limit for the size of the data segment (RLIMIT\_DATA); and the location ofmemory mappings, shared memory segments, and sharedlibraries.A call to sbrk() adjusts the program break by adding increment to it. (On Linux, sbrk() is a library function implemented on top of brk().) The intptr\_t type used to declare increment is an integer data type. On success, sbrk() returns the previousaddress of the program break. In other words, if we have increased the program break, then the return value is a pointer to the start of the newly allocated block of memory.The call sbrk(0) returns the current setting of the program break without changing it. This can be useful if we want to track the size of the heap, perhaps in order to monitor the behavior of a memory allocation package.

Allocating Memory on the Heap: malloc() and free()In general, C programs use the malloc family of functions to allocate and deallocate memory on the heap. These functions offer several advantages over brk() and sbrk(). In particular, they:ℵ are standardized as part of the C language;ℵ are easier to use in threaded programs;ℵ provide a simple interface that allows memory to be allocated in small units; andℵ allow us to arbitrarily deallocate blocks of memory, which are maintained on a free list and recycled in future calls to allocate memory.The malloc() function allocates size bytes from the heap and returns a pointer to the start of the newly allocated block of memory. The allocated memory is not initialized.

**In-lab Programs**

7 ( 1 ).

/\* It prints the addresses of all the segments and the address of variables residing in their respective segments. segments-during-runtime.c \*/

#include<stdio.h>

#include<malloc.h>

int glb\_uninit; /\* Part of BSS Segment -- global uninitialized variable, at runtime it is

initialized to zero \*/

int glb\_init = 10;

/\* Part of DATA Segment -- global initialized variable \*/

void foo(void)

{

static int num = 0;

/\* stack frame count \*/

int autovar;

/\* automatic variable/Local variable \*/

int \*ptr\_foo = (int\*)malloc(sizeof(int));

if (++num == 4)

/\* Creating four stack frames \*/

return;

printf("Stack frame number %d: address of autovar: %p\n", num, & autovar);

printf("Address of heap allocated inside foo() %p\n",ptr\_foo);

foo();

/\* function call \*/

}

int main()

{

char \*p, \*b, \*nb;

int \*ptr\_main = (int\*)malloc(sizeof(int));

printf("Text Segment:\n");

printf("Address of main: %p\n", main);

printf("Address of afunc: %p\n",foo);

printf("Stack Locations:\n");

foo();

printf("Data Segment:\n");

printf("Address of glb\_init: %p\n", & glb\_init);

printf("BSS Segment:\n");

printf("Address of glb\_uninit: %p\n", & glb\_uninit);

printf("Heap Segment:\n");

printf("Address of heap allocated inside main() %p\n",ptr\_main);

return 0;

}

/\*

OUTPUT:

Text Segment:

Address of main: 0x4006b4

Address of afunc: 0x400626

Stack Locations:

Stack frame number 1: address of autovar: 0x7fffa5e3fefc

Address of heap allocated inside foo() 0x2137440

Stack frame number 2: address of autovar: 0x7fffa5e3fecc

Address of heap allocated inside foo() 0x2137460

Stack frame number 3: address of autovar: 0x7fffa5e3fe9c

Address of heap allocated inside foo() 0x2137480

Data Segment:

Address of glb\_init: 0x601050

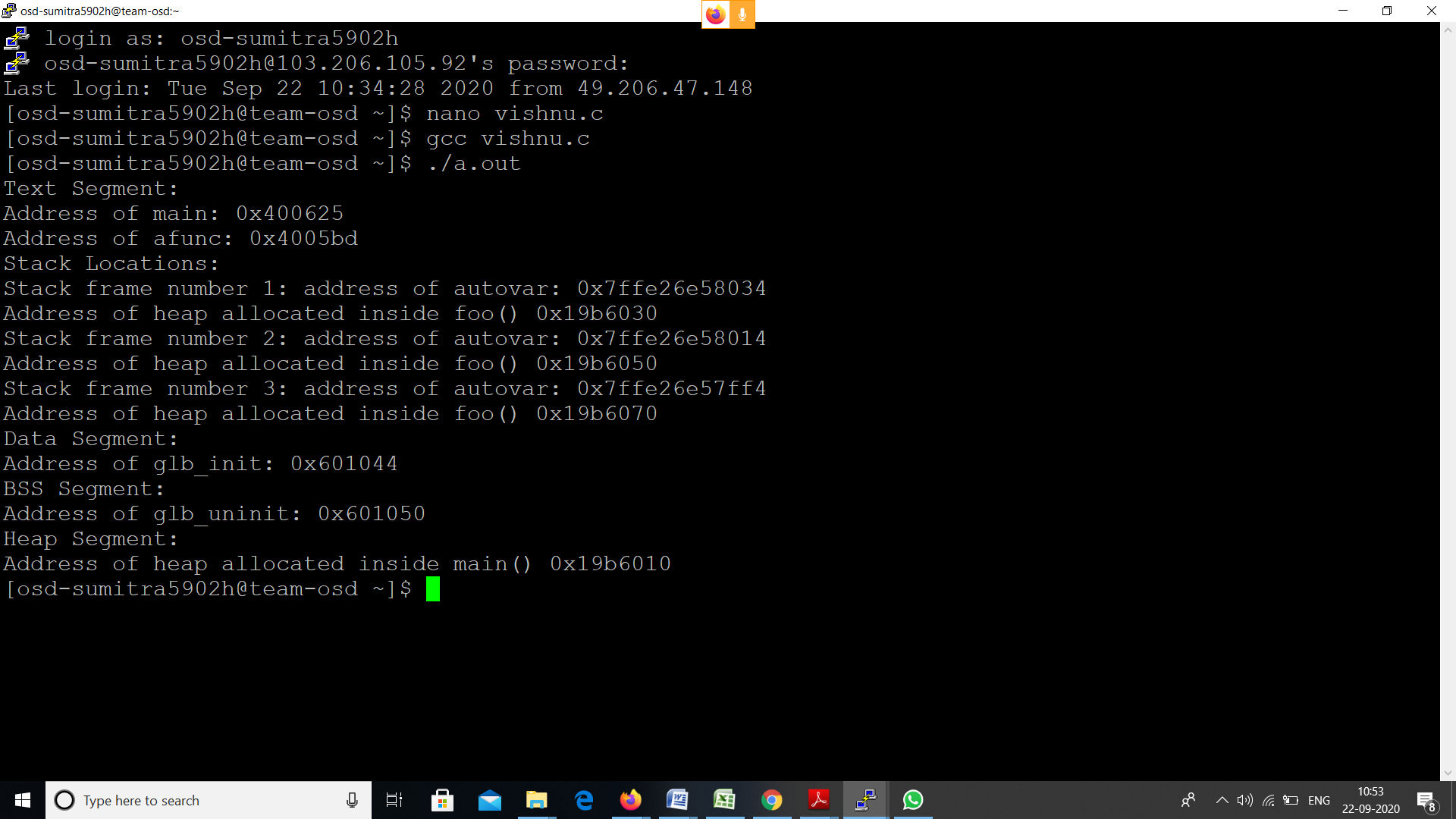
BSS Segment:

Address of glb\_uninit: 0x60105c

Heap Segment:

Address of heap allocated inside main() 0x2137010

\*/



**7 ( 2 )**

Develop a program to illustrate the effect of free() on the program break. This program allocates multiple blocks of memory and then frees some or all of them, depending on its (optional) command-line arguments.

Example : programThe program in Listing 1 can be used to illustrate the effect of free() on the program break. This program allocates multiple blocks of memory and then frees some or all of them, depending on its (optional) command-line arguments. The first two command-line arguments specify the number and size of blocks to allocate. The third command-line argument specifies the loop step unit to be used when freeing memory blocks. If we specify 1 here (which is also the default if this argument is omitted), then the program frees every memory block; if 2, then every second allocated block; and so on. The fourth and fifth command-line arguments specify the range of blocks that we wish to free. If these arguments are omitted, then all allocated blocks (insteps given by the third command-line argument) are freed

\* free\_and\_sbrk.c

Test if free(3) actually lowers the program break.

Usage: free\_and\_sbrk num-allocs block-size [step [min [max]]]\*/

#define MAX\_ALLOCS 1000000

#include <stdio.h> /\* Standard I/O functions \*/

#include <stdlib.h> /\* Prototypes of commonly used library functions,plus EXIT\_SUCCESS and EXIT\_FAILURE constants \*/

#include <unistd.h> /\* Prototypes for many system calls \*/

#include <errno.h> /\* Declares errno and defines error constants \*/

#include <string.h> /\* Commonly used string-handling functions \*/

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

char \*ptr[MAX\_ALLOCS];

int freeStep, freeMin, freeMax, blockSize, numAllocs,j;

printf("\n");

if (argc < 3 || strcmp(argv[1], "--help") == 0){

printf("%s num-allocs block-size [step [min [max]]]\n" argv[0]);

exit(5); }

numAllocs = strtol(argv[1], NULL, 10);

if (numAllocs > MAX\_ALLOCS){

printf("num-allocs > %d\n", MAX\_ALLOCS);

exit(5); }

blockSize = strtol(argv[2], NULL, 10);

freeStep = (argc > 3) ? strtol(argv[3], NULL, 10): 1;

freeMin = (argc > 4) ? strtol(argv[4], NULL, 10) : 1;

freeMax = (argc > 5) ? strtol(argv[5], NULL, 10) : numAllocs;

if (freeMax > numAllocs){ printf("free-max > num-allocs\n");

exit(5); }

printf("Initial program break: %10p\n", sbrk(0));

printf("Allocating %d\*%d bytes\n", numAllocs, blockSize);

for (j = 0; j < numAllocs; j++) {ptr[j] = malloc(blockSize);

if (ptr[j] == NULL){ perror("malloc");

exit(5); }}

printf("Program break is now: %10p\n", sbrk(0));

printf("Freeing blocks from %d to %d in steps of %d\n",freeMin, freeMax, freeStep);

for (j = freeMin -1;

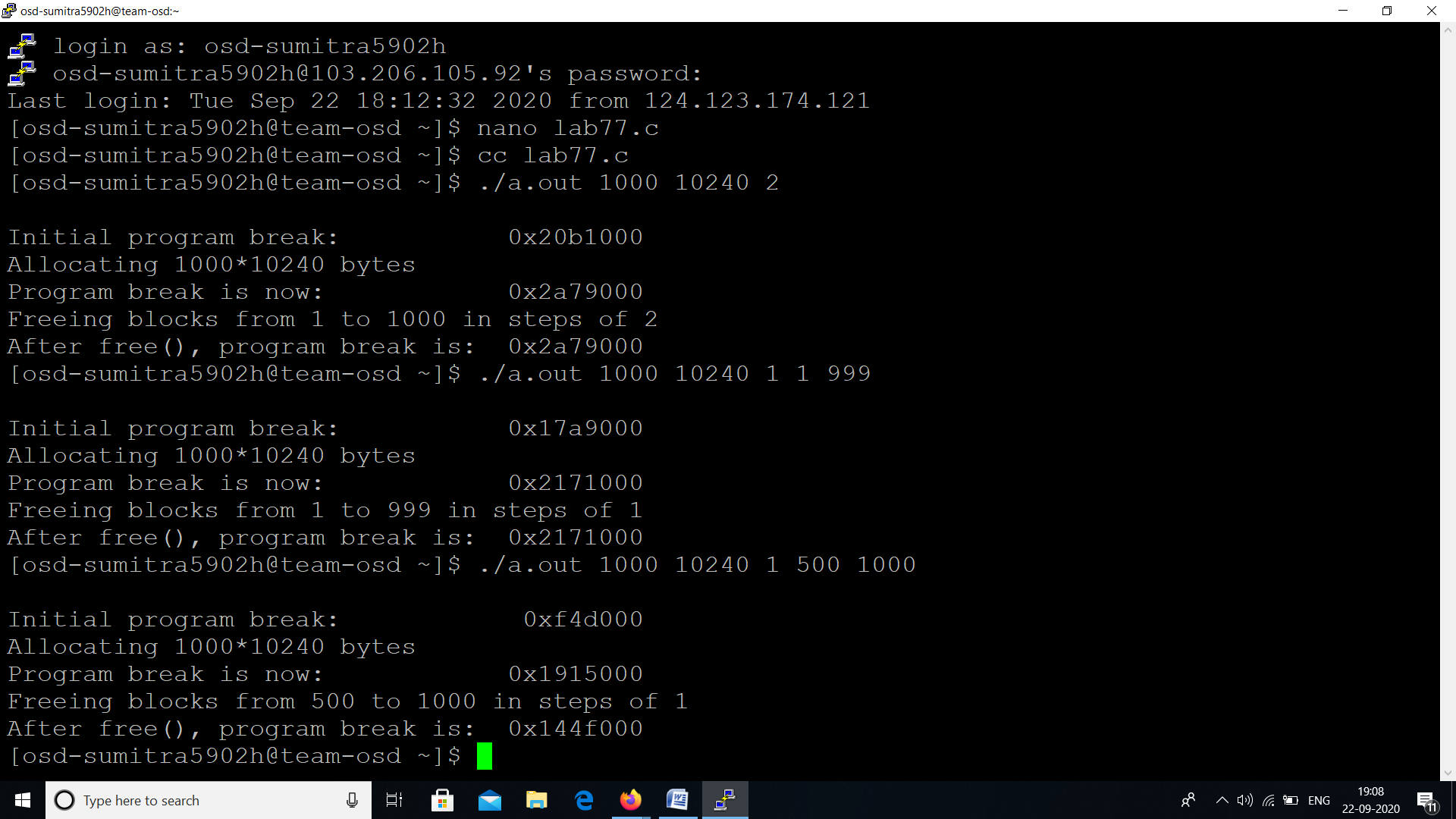
j < freeMax;

j += freeStep)free(ptr[j]);

printf("After free(), program break is: %10p\n", sbrk(0));

exit(10);

Running the program in Listing 1 with the following command line causes the program to allocate 1000 blocks of memory and then free every second block:



osdlab:~$ cc free\_and\_sbrk.c

osdlab:~$ ./a.out 1000 10240 2

The output shows that after these blocks have been freed, the program break is left unchanged from the level it reached when all memory blocks were allocated:

Initial program break: 0x1e1f000Allocating 1000\*10240 bytes

Program break is now: 0x27e7000

Freeing blocks from 1 to 1000 in steps of 2After free(),

program break is: 0x27e7000

The following command line specifies that all but the last of the allocated blocks should be freed. Again, the program break remains at its “high-water mark.”

osdlab:~$ ./a.out 1000 10240 1 1 999

Initial program break:

0xed7000Allocating 1000\*10240 bytes

Program break is now: 0x189f000

Freeing blocks from 1 to 999 in steps of 1

After free(), program break is: 0x189f000

If, however, we free a complete set of blocks at the top end of the heap, we see that the program break decreases from its peak value, indicating that free() has used sbrk() to lower the program break. Here, we free the last 500 blocks of allocated memory:

osdlab:~$./a.out 1000 10240 1 500 1000

Initial program break: 0x211c000

Allocating 1000\*10240

bytesProgram break is now: 0x2ae4000

Freeing blocks from 500 to 1000 in steps of 1

After free(), program break is: 0x25fd000

In this case, the (glibc) free() function is able to recognize that an entire region at the top end of the heap is free, since, when releasing blocks, it coalesces neighboring free blocks into a single larger block. (Such coalescing is done to avoid having a large number of small fragments on the free list, all of which may be too small to satisfy subsequent malloc() requests.

Other Methods of Allocating Memory on the Heap

As well as malloc(), the C library provides a range of other functions for allocating memory on the heap, and we describe those functions here.

Allocating memory with calloc() and realloc()

The calloc() function allocates memory for an array of identical items.

#include <stdlib.h>

void \*calloc(size\_t numitems, size\_t size);

Returns pointer to allocated memory on success, or NULL on errorThe numitems argument specifies how many items to allocate, and size specifies their size. After allocating a block of memory of the appropriate size, calloc()returns a pointer to the start of the block (or NULL if the memory could not be allocated). Unlike malloc(), calloc() initializes the allocated memory to 0.

Here is an example of the use of calloc():struct { /\* Some field definitions \*/ } myStruct;struct myStruct \*p;p = calloc(1000, sizeof(struct myStruct));if (p == NULL)perror("calloc");

The realloc() function is used to resize (usually enlarge) a block of memory previouslyallocated by one of the functions in the malloc package.#include <stdlib.h>void \*realloc(void \*ptr, size\_t size);

void \*realloc(void \*ptr, size\_t size);

Returns pointer to allocated memory on success, or NULL on errorThe ptr argument is a pointer to the block of memory that is to be resized. The sizeargument specifies the desired new size of the block.

On success, realloc() returns a pointer to the location of the resized block. Thismay be different from its location before the call.

On error, realloc() returns NULL and leaves the block pointed to by ptr untouched (SUSv3 requires this).When realloc() increases the size of a block of allocated memory, it doesn’t initialize the additionally allocated bytes. Memory allocated using calloc() or realloc() should be deallocated with free().

The call realloc(ptr, 0) is equivalent to calling free(ptr) followed by malloc(0). Ifptr is specified as NULL, then realloc() is equivalent to calling malloc(size).For the usual case, where we are increasing the size of the block of memory, realloc()attempts to coalesce the block with an immediately following block of memory onthe free list, if one exists and is large enough. If the block lies at the end of the heap,then realloc() expands theheap. If the block of memory lies in the middle of the heap,and there is insufficient free space immediately following it, realloc() allocates a newblock of memory and copies all existing data from the old block to the new block.

This last case is common and CPU-intensive. In general, it is advisable to minimizethe use of realloc().Since realloc() may relocate the block of memory, we must use the returned pointer from realloc() for future references to the memory block.

We can employrealloc() to reallocate a block pointed to by the variable ptr as follows:

nptr = realloc(ptr, newsize);if (nptr == NULL) {/\* Handle error \*/}else { /\* realloc() succeeded \*/ptr = nptr;}

In this example, we didn’t assign the return value of realloc() directly to ptr because, if realloc() had failed, then ptr would have been set to NULL, making the existingblock inaccessible.

Because realloc() may move the block of memory, any pointers that referred to locations inside the block before the realloc() call may no longer be valid after the call. The only type of reference to a location within the block that is guaranteed to remain valid is one formed by adding an offset to the pointer to the start of the block.

/\*malloc.c

implement malloc and free this is probably horrible c code.deficencies:\* it doesn't increase the page break by huge sizes to avoid too many system calls\* it doesn't lower the page break when memory gets freed\* it requires space to store pointers to the last and next block even for non-free blocksperformance\*/

#include <sys/types.h> /\* Type definitions used by many programs \*/

#include <stdio.h> /\* Standard I/O functions \*/

#include <stdlib.h> /\* Prototypes of commonly used library functions,plus EXIT\_SUCCESS and EXIT\_FAILURE constants \*/

#include <unistd.h> /\* Prototypes for many system calls \*/

#include <errno.h> /\* Declares errno and defines error constants \*/

#include <string.h> /\* Commonly used string-handling functions \*/

extern char end;

void \*my\_malloc (size\_t);

void my\_free(void \*);

struct blk {size\_t size;

struct blk \*prev;

struct blk \*next;};

struct blk \*first = NULL;

struct blk \*last = NULL;

void \*my\_malloc (size\_t size) {size\_t required\_size = size + sizeof(struct blk);

struct blk \*curr = first;

while (curr != NULL && curr->size < required\_size) {curr = curr->next;

}if (curr == NULL) {void \*new = sbrk((intptr\_t) required\_size);

if (new == (void \*) -1) { return NULL; }

struct blk \*new\_blk = (struct blk \*) new;

new\_blk->size = required\_size;

return (void \*) (new\_blk + 1);}

if (curr == first) { first = first->next; }

else { curr->prev->next = curr->next; }

if (curr == last) { last = last->prev; }

else {curr->next->prev = curr->prev; }if (curr->size > required\_size + sizeof(struct blk)) {struct blk \*left = (struct blk \*) (((char \*) curr) + required\_size);

left->size = curr->size -required\_size;

curr->size = required\_size;

my\_free((char \*) (left + 1));}return (void \*) (curr + 1);}

void my\_free (void \*ptr) {struct blk \*blk\_ptr = ((struct blk \*) ptr) -1;

if (first == NULL) {first = last = blk\_ptr;return;}if (blk\_ptr < first) {blk\_ptr->prev = NULL;

if (((char \*) blk\_ptr) + blk\_ptr->size == (char \*) first) {blk\_ptr->size += first->size;

blk\_ptr->next = first->next;}

else {first->prev = blk\_ptr;blk\_ptr->next = first;}first = blk\_ptr;return;}

if (blk\_ptr > last) {if (((char \*) last) + last->size == (char \*) blk\_ptr) {last->size += blk\_ptr->size;}

else {blk\_ptr->next = NULL;

blk\_ptr->prev = last;

last->next = blk\_ptr;

last = blk\_ptr;}

return;}

struct blk \*curr = first;

while (curr < blk\_ptr) {curr = curr->next;}

struct blk \*before = curr->prev;

if (((char \*) before) + before->size == (char \*) blk\_ptr) {before->size += blk\_ptr->size;

blk\_ptr = before;}

else {blk\_ptr->prev = before;

before->next = blk\_ptr;}

if (((char \*) blk\_ptr) + blk\_ptr->size == (char \*) curr) {blk\_ptr->size += curr->size;

blk\_ptr->next = curr->next;

curr->next->prev = blk\_ptr;

} else {blk\_ptr->next = curr;

curr->prev = blk\_ptr;}}

#define MAX\_ALLOCS 1000000

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

/\* copied from free\_and\_sbrk.c --licensed by Michael Kerrisk under the GPLv3 \*/

char \*ptr[MAX\_ALLOCS];

int freeStep, freeMin, freeMax, blockSize, numAllocs, j;

printf("\n");

if (argc < 3 || strcmp(argv[1], "--help") == 0) {printf("%s num-allocs block-size [step [min [max]]]\n", argv[0]);

perror("num-allocs block-size");}

numAllocs = strtol(argv[1], NULL, 10);

if (numAllocs > MAX\_ALLOCS) {printf("num-allocs > %d\n", MAX\_ALLOCS);

perror("num-allocs");}

blockSize = strtol(argv[2], NULL, 10);

freeStep = (argc > 3) ? strtol(argv[3], NULL, 10) : 1;

freeMin = (argc > 4) ? strtol(argv[4], NULL, 10) : 1;

freeMax = (argc > 5) ? strtol(argv[5], NULL, 10): numAllocs;

if (freeMax > numAllocs) {perror("free-max > num-allocs");}

printf("Initial program break: %10p\n", sbrk(0));

printf("Allocating %d\*%d bytes\n", numAllocs, blockSize);

for (j = 0; j < numAllocs; j++) {

ptr[j] = my\_malloc(blockSize);

if (ptr[j] == NULL) {perror("malloc");}

printf("%10p\n", sbrk(0));}

printf("Program break is now: %10p\n", sbrk(0));

printf("Freeing blocks from %d to %d in steps of %d\n",freeMin, freeMax, freeStep);

for (j = freeMin -1; j < freeMax; j += freeStep) {my\_free(ptr[j]);}

printf("After my\_free(), program break is: %10p\n", sbrk(0));

exit(EXIT\_SUCCESS);}

osdlab:~/programs$ ./a.out 1000 10240 1 1 999

Initial program break: 0x8ad0000

Allocating 1000\*10240 bytes0x8ad280c

0x8ad5018

0x8ad7824

0x8ada030

0x8adc83c

0x8adf048

0x8ae1854

0x8ae4060

0x8ae686c

0x8ae9078

0x8aeb884

0x8aee090

0x8af089c

0x8af30a8..

Program break is now: 0x9496ee0

Freeing blocks from 1 to 999 in steps of 1

After my\_free(), program break is: 0x9496ee0

