7

MEMORY ALLOCATION

Many system programs need to be able to allocate extra memory for dynamic data structures (e.g., linked lists and binary trees), whose size depends on information that is available only at run time. This chapter describes the functions that are used to allocate memory on the heap or the stack.

7.1 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 (see Figure 6-1 on page 119). 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.

7.1.1 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 of where the process's program break is. Initially, the program break lies just past the end of the uninitialized data segment (i.e., the same location as *&end*, shown in Figure 6-1).

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 to access addresses in those pages.

Traditionally, the UNIX system has provided two system calls for manipulating the program break, and these are both available on 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 error
void *sbrk(intptr_t increment);
                    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 effectively rounded 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, described in Section 20.2) when trying to access data in now nonexistent parts 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, described in Section 36.3); and the location of memory mappings, shared memory segments, and shared libraries.

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 previous address 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.

> SUSv2 specified *brk()* and *sbrk()* (marking them LEGACY). SUSv3 removed their specifications.

7.1.2 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:

- are standardized as part of the C language;
- are easier to use in threaded programs;

- provide a simple interface that allows memory to be allocated in small units; and
- 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.

```
#include <stdlib.h>

void *malloc(size_t size);
                    Returns pointer to allocated memory on success, or NULL on error
```

Because *malloc()* returns *void **, we can assign it to any type of C pointer. The block of memory returned by *malloc()* is always aligned on a byte boundary suitable for any type of C data structure. In practice, this means that it is allocated on an 8-byte or 16-byte boundary on most architectures.

> SUSv3 specifies that the call *malloc(0)* may return either NULL or a pointer to a small piece of memory that can (and should) be freed with *free()*. On Linux, *malloc(0)* follows the latter behavior.

If memory could not be allocated (perhaps because we reached the limit to which the program break could be raised), then *malloc()* returns NULL and sets *errno* to indicate the error. Although the possibility of failure in allocating memory is small, all calls to *malloc()*, and the related functions that we describe later, should check for this error return.

The *free()* function deallocates the block of memory pointed to by its *ptr* argument, which should be an address previously returned by *malloc()* or one of the other heap memory allocation functions that we describe later in this chapter.

```
#include <stdlib.h>

void free(void *ptr);
```

In general, *free()* doesn't lower the program break, but instead adds the block of memory to a list of free blocks that are recycled by future calls to *malloc()*. This is done for several reasons:

- The block of memory being freed is typically somewhere in the middle of the heap, rather than at the end, so that lowering the program break is not possible.
- It minimizes the number of *sbrk()* calls that the program must perform. (As noted in Section 3.1, system calls have a small but significant overhead.)
- In many cases, lowering the break would not help programs that allocate large amounts of memory, since they typically tend to hold on to allocated memory or repeatedly release and reallocate memory, rather than release it all and then continue to run for an extended period of time.

If the argument given to *free()* is a NULL pointer, then the call does nothing. (In other words, it is not an error to give a NULL pointer to *free()*.)

Making any use of *ptr* after the call to *free()*—for example, passing it to *free()* a second time—is an error that can lead to unpredictable results.

Example program

The program in Listing 7-1 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 (in steps given by the third command-line argument) are freed.

Listing 7-1: Demonstrate what happens to the program break when memory is freed

memalloc/free_and_sbrk.c

```
#include "tlpi_hdr.h"

#define MAX_ALLOCS 1000000

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

    numAllocs = getInt(argv[1], GN_GT_0, "num-allocs");
    if (numAllocs > MAX_ALLOCS)
        cmdLineErr("num-allocs > %d\n", MAX_ALLOCS);

    blockSize = getInt(argv[2], GN_GT_0 | GN_ANY_BASE, "block-size");

    freeStep = (argc > 3) ? getInt(argv[3], GN_GT_0, "step") : 1;
    freeMin =  (argc > 4) ? getInt(argv[4], GN_GT_0, "min") : 1;
    freeMax =  (argc > 5) ? getInt(argv[5], GN_GT_0, "max") : numAllocs;

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

    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)
            errExit("malloc");
    }

    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(EXIT_SUCCESS);
}
```

memalloc/free_and_sbrk.c

Running the program in Listing 7-1 with the following command line causes the program to allocate 1000 blocks of memory and then free every second block:

```
$ ./free_and_sbrk 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:          0x804a6bc
Allocating 1000*10240 bytes
Program break is now:           0x8a13000
Freeing blocks from 1 to 1000 in steps of 2
After free(), program break is:  0x8a13000
```

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."

```
$ ./free_and_sbrk 1000 10240 1 1 999
Initial program break:          0x804a6bc
Allocating 1000*10240 bytes
Program break is now:           0x8a13000
Freeing blocks from 1 to 999 in steps of 1
After free(), program break is:  0x8a13000
```

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:

```
$ ./free_and_sbrk 1000 10240 1 500 1000
Initial program break:          0x804a6bc
Allocating 1000*10240 bytes
Program break is now:           0x8a13000
Freeing blocks from 500 to 1000 in steps of 1
After free(), program break is:  0x852b000
```

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.)

> The *glibc free()* function calls *sbrk()* to lower the program break only when the free block at the top end is "sufficiently" large, where "sufficient" is determined by parameters controlling the operation of the *malloc* package (128 kB is a typical value). This reduces the number of *sbrk()* calls (i.e., the number of *brk()* system calls) that must be made.

To *free()* or not to *free()*?

When a process terminates, all of its memory is returned to the system, including heap memory allocated by functions in the *malloc* package. In programs that allocate memory and continue using it until program termination, it is common to omit calls to *free()*, relying on this behavior to automatically free the memory. This can be especially useful in programs that allocate many blocks of memory, since adding multiple calls to *free()* could be expensive in terms of CPU time, as well as perhaps being complicated to code.

Although relying on process termination to automatically free memory is acceptable for many programs, there are a couple of reasons why it can be desirable to explicitly free all allocated memory:

- Explicitly calling *free()* may make the program more readable and maintainable in the face of future modifications.
- If we are using a *malloc* debugging library (described below) to find memory leaks in a program, then any memory that is not explicitly freed will be reported as a memory leak. This can complicate the task of finding real memory leaks.

7.1.3 Implementation of *malloc()* and *free()*

Although *malloc()* and *free()* provide an interface for allocating memory that is much easier to use than *brk()* and *sbrk()*, it is still possible to make various programming errors when using them. Understanding how *malloc()* and *free()* are implemented provides us with insights into the causes of these errors and how we can avoid them.

The implementation of *malloc()* is straightforward. It first scans the list of memory blocks previously released by *free()* in order to find one whose size is larger than or equal to its requirements. (Different strategies may be employed for this scan, depending on the implementation; for example, *first-fit* or *best-fit*.) If the block is exactly the right size, then it is returned to the caller. If it is larger, then it is split, so that a block of the correct size is returned to the caller and a smaller free block is left on the free list.

If no block on the free list is large enough, then *malloc()* calls *sbrk()* to allocate more memory. To reduce the number of calls to *sbrk()*, rather than allocating exactly the number of bytes required, *malloc()* increases the program break in larger units (some multiple of the virtual memory page size), putting the excess memory onto the free list.

Looking at the implementation of *free()*, things start to become more interesting. When *free()* places a block of memory onto the free list, how does it know what size that block is? This is done via a trick. When *malloc()* allocates the block, it allocates extra bytes to hold an integer containing the size of the block. This integer is located at the beginning of the block; the address actually returned to the caller points to the location just past this length value, as shown in Figure 7-1.

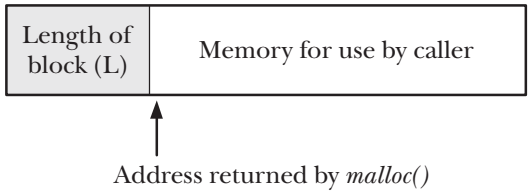

Figure 7-1: Memory block returned by *malloc()*

When a block is placed on the (doubly linked) free list, *free()* uses the bytes of the block itself in order to add the block to the list, as shown in Figure 7-2.

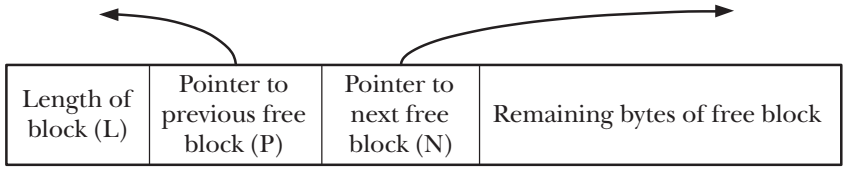

Figure 7-2: A block on the free list

As blocks are deallocated and reallocated over time, the blocks of the free list will become intermingled with blocks of allocated, in-use memory, as shown in Figure 7-3.

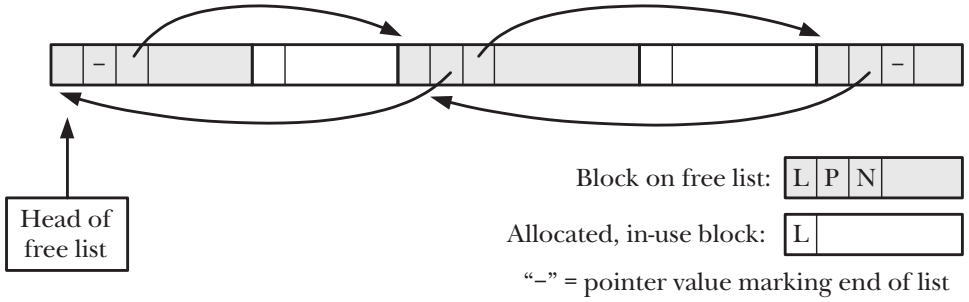

Figure 7-3: Heap containing allocated blocks and a free list

Now consider the fact that C allows us to create pointers to any location in the heap, and modify the locations they point to, including the *length*, *previous free block*, and *next free block* pointers maintained by *free()* and *malloc()*. Add this to the preceding description, and we have a fairly combustible mix when it comes to creating obscure programming bugs. For example, if, via a misdirected pointer, we accidentally increase one of the length values preceding an allocated block of memory, and subsequently deallocate that block, then *free()* will record the wrong size block of memory on the free list. Subsequently, *malloc()* may reallocate this block, leading to a scenario where the program has pointers to two blocks of allocated memory that it understands to be distinct, but which actually overlap. Numerous other pictures of what could go wrong can be drawn.

To avoid these types of errors, we should observe the following rules:

- After we allocate a block of memory, we should be careful not to touch any bytes outside the range of that block. This could occur, for example, as a result of faulty pointer arithmetic or off-by-one errors in loops updating the contents of a block.
- It is an error to free the same piece of allocated memory more than once. With *glibc* on Linux, we often get a segmentation violation (`SIGSEGV` signal). This is good, because it alerts us that we've made a programming error. However, more generally, freeing the same memory twice leads to unpredictable behavior.
- We should never call *free()* with a pointer value that wasn't obtained by a call to one of the functions in the *malloc* package.
- If we are writing a long-running program (e.g., a shell or a network daemon process) that repeatedly allocates memory for various purposes, then we should ensure that we deallocate any memory after we have finished using it. Failure to do so means that the heap will steadily grow until we reach the limits of available virtual memory, at which point further attempts to allocate memory fail. Such a condition is known as a *memory leak*.

Tools and libraries for *malloc* debugging

Failure to observe the rules listed above can lead to the creation of bugs that are obscure and difficult to reproduce. The task of finding such bugs can be eased considerably by using the *malloc* debugging tools provided by *glibc* or one of a number of *malloc* debugging libraries that are designed for this purpose.

Among the *malloc* debugging tools provided by *glibc* are the following:

- The *mtrace()* and *muntrace()* functions allow a program to turn tracing of memory allocation calls on and off. These functions are used in conjunction with the `MALLOC_TRACE` environment variable, which should be defined to contain the name of a file to which tracing information should be written. When *mtrace()* is called, it checks to see whether this file is defined and can be opened for writing; if so, then all calls to functions in the *malloc* package are traced and recorded in the file. Since the resulting file is not easily human-readable, a script—also called *mtrace*—is provided to analyze the file and produce a readable summary. For security reasons, calls to *mtrace()* are ignored by set-user-ID and set-group-ID programs.
- The *mcheck()* and *mprobe()* functions allow a program to perform consistency checks on blocks of allocated memory; for example, catching errors such as attempting to write to a location past the end of a block of allocated memory. These functions provide functionality that somewhat overlaps with the *malloc* debugging libraries described below. Programs that employ these functions must be linked with the *mcheck* library using the *cc −lmcheck* option.
- The `MALLOC_CHECK_` environment variable (note the trailing underscore) serves a similar purpose to *mcheck()* and *mprobe()*. (One notable difference between the two techniques is that using `MALLOC_CHECK_` doesn't require modification and recompilation of the program.) By setting this variable to different integer values, we can control how a program responds to memory allocation errors. Possible

settings are: 0, meaning ignore errors; 1, meaning print diagnostic errors on *stderr*; and 2, meaning call *abort()* to terminate the program. Not all memory allocation and deallocation errors are detected via the use of `MALLOC_CHECK_`; it finds just the common ones. However, this technique is fast, easy to use, and has low run-time overhead compared with the use of *malloc* debugging libraries. For security reasons, the setting of `MALLOC_CHECK_` is ignored by set-user-ID and set-group-ID programs.

Further information about all of the above features can be found in the *glibc* manual.

A *malloc* debugging library offers the same API as the standard *malloc* package, but does extra work to catch memory allocation bugs. In order to use such a library, we link our application against that library instead of the *malloc* package in the standard C library. Because these libraries typically operate at the cost of slower run-time operation, increased memory consumption, or both, we should use them only for debugging purposes, and then return to linking with the standard *malloc* package for the production version of an application. Among such libraries are *Electric Fence* (*http://www.perens.com/FreeSoftware/*), *dmalloc* (*http://dmalloc.com/*), *Valgrind* (*http://valgrind.org/*), and *Insure++* (*http://www.parasoft.com/*).

> Both *Valgrind* and *Insure++* are capable of detecting many other kinds of bugs aside from those associated with heap allocation. See their respective web sites for details.

Controlling and monitoring the *malloc* package

The *glibc* manual describes a range of nonstandard functions that can be used to monitor and control the allocation of memory by functions in the *malloc* package, including the following:

- The *mallopt()* function modifies various parameters that control the algorithm used by *malloc()*. For example, one such parameter specifies the minimum amount of releasable space that must exist at the end of the free list before *sbrk()* is used to shrink the heap. Another parameter specifies an upper limit for the size of blocks that will be allocated from the heap; blocks larger than this are allocated using the *mmap()* system call (refer to Section 49.7).
- The *mallinfo()* function returns a structure containing various statistics about the memory allocated by *malloc()*.

Many UNIX implementations provide versions of *mallopt()* and *mallinfo()*. However, the interfaces offered by these functions vary across implementations, so they are not portable.

7.1.4 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 error
```

The *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)
    errExit("calloc");
```

The *realloc()* function is used to resize (usually enlarge) a block of memory previously allocated by one of the functions in the *malloc* package.

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
                    Returns pointer to allocated memory on success, or NULL on error
```

The *ptr* argument is a pointer to the block of memory that is to be resized. The *size* argument specifies the desired new size of the block.

On success, *realloc()* returns a pointer to the location of the resized block. This may 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)*. If *ptr* 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 on the free list, if one exists and is large enough. If the block lies at the end of the heap, then *realloc()* expands the heap. If the block of memory lies in the middle of the heap, and there is insufficient free space immediately following it, *realloc()* allocates a new block 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 minimize the 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 employ *realloc()* 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 existing block 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. We discuss this point in more detail in Section 48.6.

Allocating aligned memory: *memalign()* and *posix_memalign()*

The *memalign()* and *posix_memalign()* functions are designed to allocate memory starting at an address aligned at a specified power-of-two boundary, a feature that is useful for some applications (see, for example, Listing 13-1, on page 247).

```
#include <malloc.h>

void *memalign(size_t boundary, size_t size);
```

Returns pointer to allocated memory on success, or `NULL` on error

The *memalign()* function allocates *size* bytes starting at an address aligned to a multiple of *boundary*, which must be a power of two. The address of the allocated memory is returned as the function result.

The *memalign()* function is not present on all UNIX implementations. Most other UNIX implementations that provide *memalign()* require the inclusion of `<stdlib.h>` instead of `<malloc.h>` in order to obtain the function declaration.

SUSv3 doesn't specify *memalign()*, but instead specifies a similar function, named *posix_memalign()*. This function is a recent creation of the standards committees, and appears on only a few UNIX implementations.

```
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);
```

Returns 0 on success, or a positive error number on error

The *posix_memalign()* function differs from *memalign()* in two respects:

- The address of the allocated memory is returned in *memptr*.
- The memory is aligned to a multiple of *alignment*, which must be a power-of-two multiple of *sizeof(void *)* (4 or 8 bytes on most hardware architectures).

Note also the unusual return value of this function—rather than returning −1 on error, it returns an error number (i.e., a positive integer of the type normally returned in *errno*).

If *sizeof(void *)* is 4, then we can use *posix_memalign()* to allocate 65,536 bytes of memory aligned on a 4096-byte boundary as follows:

```
int s;
void *memptr;

s = posix_memalign(&memptr, 1024 * sizeof(void *), 65536);
if (s != 0)
    /* Handle error */
```

Blocks of memory allocated using *memalign()* or *posix_memalign()* should be deallocated with *free()*.

> On some UNIX implementations, it is not possible to call *free()* on a block of memory allocated via *memalign()*, because the *memalign()* implementation uses *malloc()* to allocate a block of memory, and then returns a pointer to an address with a suitable alignment in that block. The *glibc* implementation of *memalign()* doesn't suffer this limitation.

7.2 Allocating Memory on the Stack: *alloca()*

Like the functions in the *malloc* package, *alloca()* allocates memory dynamically. However, instead of obtaining memory from the heap, *alloca()* obtains memory from the stack by increasing the size of the stack frame. This is possible because the calling function is the one whose stack frame is, by definition, on the top of the stack. Therefore, there is space above the frame for expansion, which can be accomplished by simply modifying the value of the stack pointer.

```
#include <alloca.h>

void *alloca(size_t size);
```

Returns pointer to allocated block of memory

The *size* argument specifies the number of bytes to allocate on the stack. The *alloca()* function returns a pointer to the allocated memory as its function result.

We need not—indeed, must not—call *free()* to deallocate memory allocated with *alloca()*. Likewise, it is not possible to use *realloc()* to resize a block of memory allocated by *alloca()*.

Although *alloca()* is not part of SUSv3, it is provided on most UNIX implementations and is thus reasonably portable.

> Older versions of *glibc*, and some other UNIX implementations (mainly BSD derivatives), require the inclusion of `<stdlib.h>` instead of `<alloca.h>` to obtain the declaration of *alloca()*.

If the stack overflows as a consequence of calling *alloca()*, then program behavior is unpredictable. In particular, we don't get a `NULL` return to inform us of the error. (In fact, in this circumstance, we may receive a `SIGSEGV` signal. Refer to Section 21.3 for further details.)

Note that we can't use *alloca()* within a function argument list, as in this example:

```
func(x, alloca(size), z);           /* WRONG! */
```

This is because the stack space allocated by *alloca()* would appear in the middle of the space for the function arguments (which are placed at fixed locations within the stack frame). Instead, we must use code such as this:

```
void *y;

y = alloca(size);
func(x, y, z);
```

Using *alloca()* to allocate memory has a few advantages over *malloc()*. One of these is that allocating blocks of memory is faster with *alloca()* than with *malloc()*, because *alloca()* is implemented by the compiler as inline code that directly adjusts the stack pointer. Furthermore, *alloca()* doesn't need to maintain a list of free blocks.

Another advantage of *alloca()* is that the memory that it allocates is automatically freed when the stack frame is removed; that is, when the function that called *alloca()* returns. This is so because the code executed during function return resets the value of the stack pointer register to the end of the previous frame (i.e., assuming a downwardly growing stack, to the address just above the start of the current frame). Since we don't need to do the work of ensuring that allocated memory is freed on all return paths from a function, coding of some functions becomes much simpler.

Using *alloca()* can be especially useful if we employ *longjmp()* (Section 6.8) or *siglongjmp()* (Section 21.2.1) to perform a nonlocal goto from a signal handler. In this case, it is difficult or even impossible to avoid memory leaks if we allocated memory in the jumped-over functions using *malloc()*. By contrast, *alloca()* avoids this problem completely, since, as the stack is unwound by these calls, the allocated memory is automatically freed.

7.3 Summary

Using the *malloc* family of functions, a process can dynamically allocate and release memory on the heap. In considering the implementation of these functions, we saw that various things can go wrong in a program that mishandles the blocks of allocated memory, and we noted that a number of debugging tools are available to help locate the source of such errors.

The *alloca()* function allocates memory on the stack. This memory is automatically deallocated when the function that calls *alloca()* returns.

7.4 Exercises

7-1. Modify the program in Listing 7-1 (`free_and_sbrk.c`) to print out the current value of the program break after each execution of *malloc()*. Run the program specifying a small allocation block size. This will demonstrate that *malloc()* doesn't employ *sbrk()* to adjust the program break on each call, but instead periodically allocates larger chunks of memory from which it passes back small pieces to the caller.

7-2. (Advanced) Implement *malloc()* and *free()*.