LAB 7: MEMORY AND DATA MANAGEMENT

Simple Memory Allocation

You allocate memory using the malloc call in the standard C library:

```
#include <stdlib.h>
void *malloc(size_t size);
```

Notice that Linux (following the X/Open specification) differs from some UNIX implementations by not requiring a special malloc.h include file. Note also that the size parameter that specifies the number of bytes to allocate isn't a simple int, although it's usually an unsigned integer type. You can allocate a great deal of memory on most Linux systems. Let's start with a very simple program, but one that would defeat old MS-DOS-based programs, because they cannot access memory outside the base 640K memory map of PCs.

Sample Program (memory1.c) :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define A_MEGABYTE (1024 * 1024)
int main()
{
	char *some_memory;
	int megabyte = A_MEGABYTE;
	int exit_code = EXIT_FAILURE;
	some_memory = (char *)malloc(megabyte);
	if (some_memory != NULL) {
		sprintf(some_memory, “Hello World\n”);
		printf(“%s”, some_memory);
		exit_code = EXIT_SUCCESS;
	}
	exit(exit_code);
}
```

When you run this program, it gives the following output:

```
$ ./memory1

Hello World
```

How It Works

This program asks the malloc library to give it a pointer to a megabyte of memory. You check to ensure that malloc was successful and then use some of the memory to show that it exists. When you run the program, you should see Hello World printed out, showing that malloc did indeed return the megabyte of usable memory. We don't check that all of the megabytes are present; we have to put some trust in the malloc code!

Notice that because malloc returns a void * pointer, you cast the result to the char * that you need.

The malloc function is guaranteed to return memory that is aligned so that it can be cast to a pointer of any type. The simple reason is that most current Linux systems use 32-bit integers and 32-bit pointers for pointing to memory, which allows you to specify up to 4 gigabytes. This ability to address directly with a 32-bit pointer, without needing segment registers or other tricks, is termed a flat 32-bit memory model.

Allocating Lots of Memory

Now that you've seen Linux exceed the limitations of the MS-DOS memory model, let's give it a more difficult problem. The next program asks to allocate somewhat more memory than is physically present in the machine, so you might expect malloc to start failing somewhere a little short of the actual amount of memory present, because the kernel and all the other running processes are using some memory.

Sample Program (memory2.c) :

Asking for All Physical Memory
With memory2.c , we're going to ask for more than the machine's physical memory. You should adjust the define PHY_MEM_MEGS depending on your physical machine:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define A_MEGABYTE (1024 * 1024)
#define PHY_MEM_MEGS 1024 /* Adjust this number as required */
int main()
{
    char *some_memory;
    size_t size_to_allocate = A_MEGABYTE;
    int megs_obtained = 0;
    while (megs_obtained < (PHY_MEM_MEGS * 2)) {
        some_memory = (char *)malloc(size_to_allocate);
    if (some_memory != NULL) {
        megs_obtained++;
        sprintf(some_memory, "Hello World");
        printf("%s - now allocated %d Megabytes\n", some_memory,
    megs_obtained);
    }
    else {
        exit(EXIT_FAILURE);
    }
    }
        exit(EXIT_SUCCESS);
}
```

The output, somewhat abbreviated, is as follows:

```
$ ./memory2
Hello World
Hello World
...
Hello World
Hello World
- now allocated 1 Megabytes
- now allocated 2 Megabytes
- now allocated 2047 Megabytes
- now allocated 2048 Megabytes
```

How It Works

The program is very similar to the previous example. It simply loops, asking for more and more memory, until it has allocated twice the amount of memory you said your machine had when you adjusted the define PHY_MEM_MEGS . The surprise is that it works at all because we appear to have created a program that uses every single byte of physical memory on the author's machine. Notice that we use the size_t type for our call to malloc.

The other interesting feature is that, at least on this machine, it ran the program in the blink of an eye. So not only have we used up all the memory, but we've done it very quickly indeed.

Let's investigate further and see just how much memory we can allocate on this machine with memory3.c . Since it's now clear that Linux can do some very clever things with memory requests, we'll allocate memory just 1K at a time and write to each block that we obtain.

Demand Paged Virtual Memory

Sample Program (memory3.c) :

```
Available Memory
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define ONE_K (1024)
int main()
{
      char *some_memory;
      int size_to_allocate = ONE_K;
      int megs_obtained = 0;
      int ks_obtained = 0;
      while (1) {
            for (ks_obtained = 0; ks_obtained < 1024; ks_obtained++) {
            some_memory = (char *)malloc(size_to_allocate);
      if (some_memory == NULL) exit(EXIT_FAILURE);
            sprintf(some_memory, "Hello World");
      }
      megs_obtained++;
      printf("Now allocated %d Megabytes\n", megs_obtained);
      }
      exit(EXIT_SUCCESS);
}
```

Output:

```
$ ./memory3
Now allocated 1 Megabytes
...
Now allocated 1535 Megabytes
Now allocated 1536 Megabytes
Out of Memory: Killed process 2365
Killed
```

and then the program ends. It also takes quite a few seconds to run, and slows down significantly around the same number as the physical memory in the machine, and exercises the hard disk quite noticeably. However, the program has allocated, and accessed, more memory than this author physically has in his machine at the time of writing. Finally, the system protects itself from this rather aggressive program and kills it. On some systems, it may simply exit quietly when malloc fails.

How It Works

The application's allocated memory is managed by the Linux kernel. Each time the program asks for memory or tries to read or write to memory that it has allocated, the Linux kernel takes charge and decides how to handle the request. Initially, the

kernel was simply able to use free physical memory to satisfy the application's request for memory, but once physical memory was full, it started using what's called swap space.
On Linux, this is a separate disk area allocated when the system was installed. If you're familiar with Windows, the Linux swap space acts a little like the hidden Windows swap file. However, unlike Windows, there are no local heap, global heap, or discardable memory segments to worry about in code — the Linux kernel does all the management for you. The kernel moves data and program code between physical memory and the swap space so that each time you read or write memory, the data always appears to have been in physical memory, wherever it was located before you attempted to access it.

• In more technical terms, Linux implements a demand paged virtual memory system. All memory seen by user programs is virtual; that is, it doesn't exist at the physical address the program uses. Linux divides all memory into pages, commonly 4,096 bytes per page. When a program tries to access memory, a virtual-to-physical translation is made, although how this is implemented and the time it takes depend on the particular hardware you're using. When the access is to memory that isn't physically resident, a page fault results and control is passed to the kernel. The Linux kernel checks the address being accessed and, if it's a legal address for that program, determines which page of physical memory to make available. It then either allocates it, if it has never been written before or if it has been stored on the disk in the swap space, reads the memory page containing the data into physical memory (possibly moving an existing page out to disk). Then, after mapping the virtual memory address to match the physical address, it allows the user program to continue. Linux applications don't need to worry about this activity because the implementation is all hidden in the kernel. Eventually, when the application exhausts both the physical memory and the swap space, or when the maximum stack size is exceeded, the kernel finally refuses the request for further memory and may pre-emptively terminate the program.

This "killing the process" behavior is different from early versions of Linux and many other flavors of UNIX, where malloc simply fails. It's termed the "out of memory (OOM) killer," and although it may seem rather drastic, it is a good compromise between letting processes allocate memory rapidly and efficiently and having the Linux kernel protect itself from a total lack of resources, which is a serious issue.

So what does this mean to the application programmer? It's all good news. Linux is very good at managing memory and will allow applications to use very large amounts of memory and even very large single blocks of memory. However, you must remember that allocating two blocks of memory won't result in a single continuously addressable block of memory. What you get is what you ask for: two separate blocks of memory.

Does this limitless supply of memory, followed by the preemptive killing of the process, mean that there's no point in checking the return from malloc? No. One of the most common problems in C programs using dynamically allocated memory is writing beyond the end of an allocated block. When this happens, the program may not terminate immediately, but you have probably overwritten some data used internally by the malloc library routines.

Usually, the result is that future calls to malloc may fail, not because there's no memory to allocate, but because the memory structures have been corrupted. These problems can be quite difficult to track down, and in programs the sooner the error is detected, the better the chances of tracking down the cause.

Abusing Memory

Suppose you try to do “bad” things with memory. In this exercise, you allocate some memory and then attempt to write past the end, in memory4.c.

Sample Program (memory4.c) :

```
#include <stdlib.h>
#define ONE_K (1024)
int main()
{
    char *some_memory;
    char *scan_ptr;
    some_memory = (char *)malloc(ONE_K);
    if (some_memory == NULL) exit(EXIT_FAILURE);
        scan_ptr = some_memory;
    while(1) {
    *scan_ptr = ‘\0’;
    scan_ptr++;
 }
    exit(EXIT_SUCCESS);
}
```

The output is simply

```
$ /memory4
Segmentation fault
```

How It Works

The Linux memory management system has protected the rest of the system from this abuse of memory. To ensure that one badly behaved program (this one) can’t damage any other programs, Linux has terminated it. Each running program on a Linux system sees its memory map, which is different from every other program’s. Only the operating system knows how physical memory is arranged, and not only manages it for user programs but also protects user programs from each other.

The Null Pointer

Unlike MS-DOS, but more like newer flavors of Windows, modern Linux systems are very protective about writing or reading from the address referred to by a null pointer, although the actual behavior is implementation-specific.

Sample Program (memory5a.c) :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *some_memory = (char *)0;
    printf(“A read from null %s\n”, some_memory);
    sprintf(some_memory, “A write to null\n”);
    exit(EXIT_SUCCESS);
}
```

The output is

```
$ ./memory5a
A read from null (null)
Segmentation fault
```

How It Works

The first printf attempts to print out a string obtained from a null pointer; then the sprintf attempts to write to a null pointer. In this case, Linux (in the guise of the GNU "C" library) has been forgiving about the read and has simply provided a "magic" string containing the characters (n u l l) \0 . It hasn't been so forgiving about the write and has terminated the program. This can sometimes help track down program bugs.

If you try this again but this time don't use the GNU "C" library, you'll discover that reading from location zero is not permitted. Here is memory5b.c :

Sample Program (memory5b.c) :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
      char z = *(const char *)0;
      printf("I read from location zero\n");
      exit(EXIT_SUCCESS);
}
```

The output is

```
$ ./memory5b
Segmentation fault
```

This time you attempt to read directly from location zero. There is no GNU libc library between you and the kernel now, and the program is terminated. Note that some versions of UNIX do permit reading from location zero, but Linux doesn't.

Freeing Memory

Up to now, we've been simply allocating memory and then hoping that when the program ends, the memory we've used hasn't been lost. Fortunately, the Linux memory management system is quite capable of reliably ensuring that memory is returned to the system when a program ends. However, most programs don't simply want to allocate some memory, use it for a short period, and then exit. A much more common use is dynamically using memory as required.

Programs that use memory on a dynamic basis should always release unused memory back to the malloc memory manager using the free call. This enables separate blocks to be remerged and enables the malloc library to look after memory, rather than have the application manage it. If a running program (process) uses and then frees memory, that free memory remains allocated to the process. Behind the scenes, Linux is managing the blocks of memory the programmer is using as a set of physical "pages, " usually 4K bytes each, in memory. However, if a page of memory is not being used, then the Linux memory manager will be able to move it from physical memory to swap space (termed paging), where it has little impact on the use of resources. If the program tries to access data inside the memory page that has been moved to swap space, then Linux will very briefly suspend the program, move the memory page back from swap space into physical memory again, and then allow the program to continue, just as though the data had been in memory all along.

#include <stdlib.h>
void free(void *ptr_to memory);
A call to free should be made only with a pointer to memory allocated by a call to malloc, calloc, or realloc.

Sample Program (memory6.c) :

```
#include <stdlib.h>
#include <stdio.h>
#define ONE_K (1024)
int main()
{
	char *some_memory;
	int exit_code = EXIT_FAILURE;
	some_memory = (char *)malloc(ONE_K);
	if (some_memory != NULL) {
		free(some_memory);
		printf("Memory allocated and freed again\n");
		exit_code = EXIT_SUCCESS;
	}
	exit(exit_code);
}
```

The output is

```
$ ./memory6
Memory allocated and freed again
```

How It Works

This program simply shows how to call free with a pointer to some previously allocated memory.Remember that once you've called free on a block of memory, it no longer belongs to the process. It's not being managed by the malloc library.

Never try to read or write memory after calling free on it.

Other Memory Allocation Functions

Two other memory allocation functions are not used as often as malloc and free: calloc and realloc .
The prototypes are-

```
#include <stdlib.h>
void *calloc(size_t number_of_elements, size_t element_size);
void *realloc(void *existing_memory, size_t new_size);
```

Although calloc allocates memory that can be freed with free , it has somewhat different parameters from malloc : It allocates memory for an array of structures and requires the number of elements and the size of each element as its parameters. The allocated memory is filled with zeros; and if calloc is successful, a pointer to the first element is returned. Like malloc , subsequent calls are not guaranteed to return contiguous space, so you can't enlarge an array created by calloc by simply calling calloc again and expecting the second call to return memory appended to that returned by the first call.

The realloc function changes the size of a block of memory that has been previously allocated. It's passed a pointer to some memory previously allocated by malloc , calloc , or realloc and resizes it up or down as requested. The realloc function may have to move data around to achieve this, so it's important to ensure that once the memory has been realloced, you always use the new pointer and never try to access the memory using pointers set up before realloc was called.

Another problem to watch out for is that realloc returns a null pointer if it has been unable to resize the memory. This means that in some applications you should avoid writing code like this:

```
my_ptr = malloc(BLOCK_SIZE);
....
my_ptr = realloc(my_ptr, BLOCK_SIZE * 10);
```

If realloc fails, then it returns a null pointer; my_ptr will point to null, and the original memory allocated with malloc can no longer be accessed via my_ptr . It may, therefore,be to your advantage to request the new memory first with malloc and then copy data from the old block to the new block using memcpy before freeing the old block. On error, this would allow the application to retain access to the data stored in the original block of memory, perhaps while arranging a clean termination of the program.

Lab Exercises

1) If you wish to implement Best Fit, First Fit, Next Fit, or Worst Fit memory allocation policy, it is probably best to do this by describing the memory as a structure in a linked list:

```
struct mab {
 int offset;
 int size;
 int allocated;
 struct mab * next;
 struct mab * prev;
};

typedef struct mab Mab;
typedef Mab * MabPtr;
```

Either way, the following set of prototypes give a guide as to the functionality you will need to provide:

```
MabPtr memChk(MabPtr m, int size); // check if memory available
MabPtr memAlloc(MabPtr m, int size); // allocate a memory block
MabPtr memFree(MabPtr m); // free memory block
MabPtr memMerge(MabPtr m); // merge two memory blocks
MabPtr memSplit(MabPtr m, int size); // split a memory block
```

2) Write a C program using Malloc for implementing Multilevel feedback queue using three queues with each of them working with different scheduling policies.

3) We have five segments numbered 0 through 4. The segments are stored in physical memory as shown in the following Fig. Write a C program to create segment table. Write methods for converting logical address to physical address. Compute the physical address for the following.
(i) 53 byte of segment 2 (ii) 852 byte of segment 3 (iii) 1222 byte of segment 0

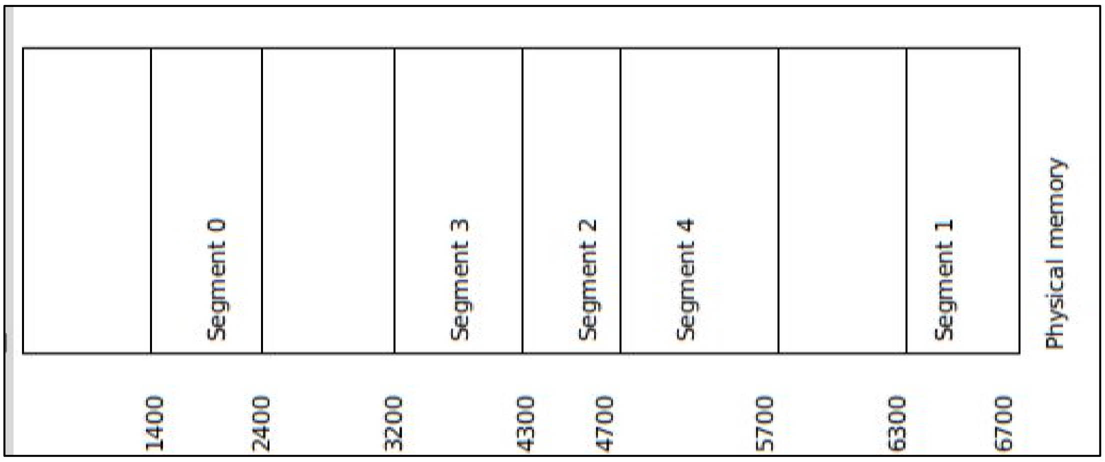

Fig.

4) Write a C program to simulate LRU approximation page replacement using second chance algorithm. Find the total number of page faults and hit ratio for the algorithm.

Additional Exercises:

1) Implement the same concept for the Buddy system.
2) How do you resize and release memory using realloc ?