Memory Management[1]

Objectives

- ✓ The purpose of this lab is to study the memory layout of a process. Unix/Linux is a particularly good environment to show you memory management, as there are often hundreds of running processes (started by dozens of people) who each require memory in order to work. Unix/Linux must distribute the pages of available memory fairly and equitably. Methods such as demand paging, page sharing and the Least Recently Used victim page selection scheme are used to manage the memory.

Prelab Activities

- ✓ Read the manual and try to do the experiment yourself before the lab.

Lab Procedure

- ✓ Write (copy and paste) and compile the codes given in Program 1-6. Also, perform the exercises.

General Information

As with most high-level languages, C creates space for your declared variables when your program is compiled, so you don't have to manually do anything before you use your variables. Global variables live in the data-segment of your process, and local variables live in the stack-segment.

However, often you need to allocate memory space dynamically, for example, when you are building linked lists with pointers. In C, the routines to do this are malloc and free.

malloc Library Routine

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

The function malloc allocates a region of memory large enough to hold an object whose size (as measured by the sizeof operator) is size. A pointer to the beginning of the region is returned. If it is impossible for some reason to perform the requested allocation, or if size is 0, a NULL pointer is returned. The region of memory is not specially initialized in any way and the caller must assume that it will contain garbage information.

Notice that malloc returns a pointer. Often you want to malloc structures. You therefore must coerce the pointer returned from malloc into the type you need. For example:

[1] M. Akbar Badhusha, King Fahd University of Petroleum and Minerals, Saudi Arabia.

```
/* Type Declarations */
struct client{
  char *name;           /* Pointer to the string holding the name */
  int age;              /* Client's age */
  int size;             /* Client's size;
  struct client *next; /* Pointer to the next element in the list */
 }
    ...

struct client *c;      /* A pointer to a client structure. Initially,
                          it points to nothing */
c=(struct client *) malloc( sizeof(struct client)); /* Create enough
                                            memory to hold a client */
if (c==NULL)
   printf("Could not malloc a client\n");
else{
   c->name= "Ahmed"; c-
   >age= 25;
   c->size= 160; c-
   >next= NULL;
}
```

free Library Routine

```
#include <stdlib.h>
void free(void *ptr);
```

The function free deallocates a region of memory previously allocated by malloc. The argument to free must be a pointer that is equivalent (except for possible intermediate type casting) to a pointer previously returned by malloc. (If the argument to free is a null pointer then no action should occur, but this is known to cause trouble in some C implementations.) Once a region of memory has been explicitly freed it must not be used for any other purpose.

To free the client struct malloc'd above, you would do free(c).

Using ps to see Memory Allocation

In the first lab, you saw that ps could give you details of all of the processes running on the Unix machine. Run the following ps command:

```
$ ps -o user,vsz,rss,pmem,fname -e
```

user:
The user who started the process running.

vsz:
The total size of the process in (virtual) memory in kilobytes as a decimal integer.

rss:
The resident set size of the process, in kilobytes as a decimal integer.

pmem:

The ratio of the process's resident set size to the physical memory on the machine, expressed as a percentage.

fname:

The first few character's of the process' name.

One reason why the resident set is smaller than the process size is that Unix processes use shared libraries, similar to DLLs on Windows systems. The operating system doesn't include the size of any shared libraries in the resident set, because the libraries are loaded into memory only once.

see Swap Usage

With a paging virtual memory system using LRU, those least recently used pages are copied out to disk until they are required again (if ever). Use the following commands:

The following command will give you the amount of RAM and SWAP in Megs.

```
$ /usr/bin/free -m
```

You can also cat the /proc/swaps file to see which partition was alocated to SWAP and it will show you the size of SWAP as well as the amount being used.

```
$ /bin/cat /proc/swaps
```

Remember that if the swap space is too small, then there is not enough room to keep the unused pages, and thrashing is likely to occur. On the other hand, if the swap space is too large, you waste disk space as it cannot be used to store files (except temporary files in /tmp).

Monitoring Paging Activity

The vmstat program is the best utility to monitor paging activity.

```
$ vmstat 2 5
```

will give 5 vmstat reports, one every 2 seconds, and the first report is an average since the system was started. Read the manual on vmstat to see what information it provides. The main memory stats columns are:

FIELD DESCRIPTION FOR VM MODE

Procs

r: The number of processes waiting for run time.

b: The number of processes in uninterruptible sleep.

Memory

swpd: the amount of virtual memory used.

free: the amount of idle memory.

buff: the amount of memory used as buffers.
cache: the amount of memory used as cache.
inact: the amount of inactive memory. (-a option)
active: the amount of active memory. (-a option)

Swap
si: Amount of memory swapped in from disk (/s).
so: Amount of memory swapped to disk (/s).

IO
bi: Blocks received from a block device (blocks/s).
bo: Blocks sent to a block device (blocks/s).

System
in: The number of interrupts per second, including the clock.
cs: The number of context switches per second.

CPU
These are percentages of total CPU time.
us: Time spent running non-kernel code. (user time, including nice time)
sy: Time spent running kernel code. (system time)
id: Time spent idle. Prior to Linux 2.5.41, this includes IO-wait time.
wa: Time spent waiting for IO. Prior to Linux 2.5.41, included in idle.
st: Time stolen from a virtual machine. Prior to Linux 2.6.11, unknown.

The page out column is often zero. Therefore, there must be many pages in memory which are unused but are not paged out to disk.

Components of a Process

A Unix process has several memory components:

- A text section which holds the process' machine code.
- A data section which holds the process' global variables. Initially, some of the global variables have values, and some do not. The latter are kept in a section known as the bss section.
- A heap section which is where newly created global variables are kept.
- A stack section which is where newly created local variables are kept, as well as function parameters and function return information.

The size program shows the sizes of the text, data and bss sections in a program's disk image. For example:

```
$ which ls                     # Where on disk is ls kept?
  /bin/ls
$ size /bin/ls
  15678 + 1241 + 1963 = 18882 # code + data + bss == total
```

Memory Structure

This section is an introduction to memory as we see it in Unix.

Memory is like a huge array with (say) 0xffffffff elements. A pointer in C is an index to this array. Thus when a C pointer is 0xefffe034, it points to the 0xefffe035th element in the memory array (memory being indexed starting with zero).

Unfortunately, you cannot access all elements of memory. One example that we have seen a lot is element 0. If you try to dereference a pointer with a value of 0, you will get a segmentation violation. **This is Unix's way of telling you that memory location is illegal.**

For example, the following code will generate a segmentation violation:

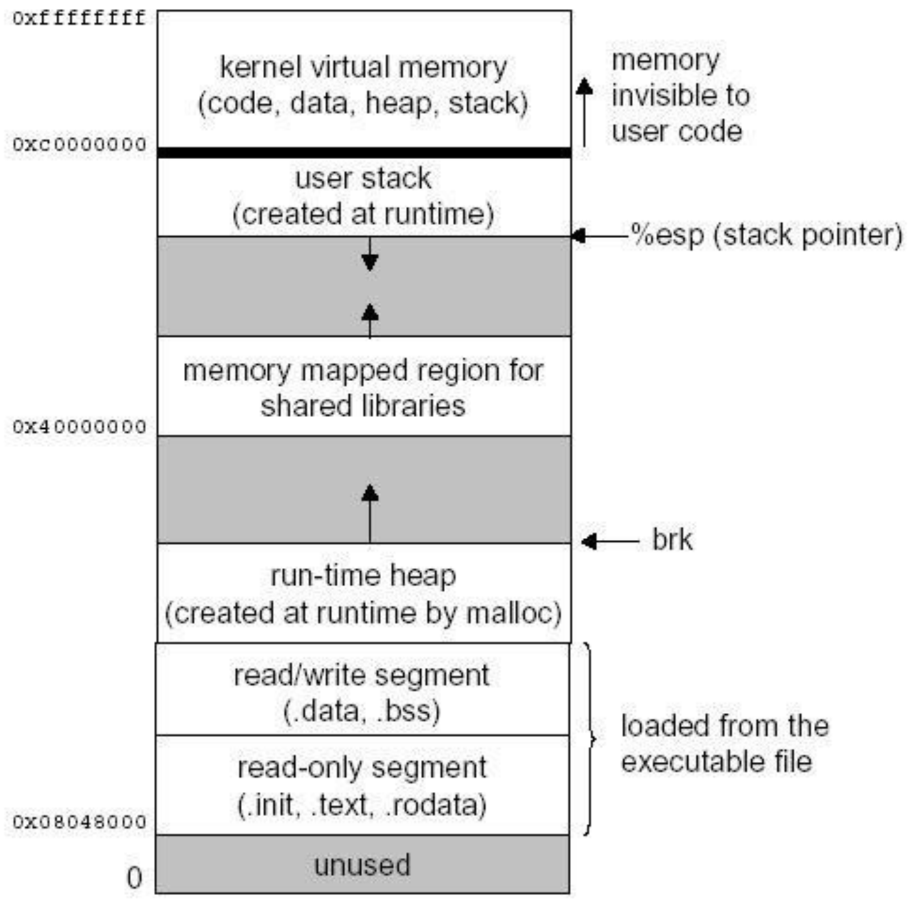
```
#include <stdio.h>

main( )
{
   char *s;
   char c;

   s=(char *)0;
   c=*s;
}
```

Program 1.

As it turns out, there are 4 regions of memory that are legal. They are:

1. The code (or "text"): These are the instructions of your program.
2. The globals: These are your global variables.
3. The heap: This is memory that you get from malloc( ).
4. The stack: This contains your local variables and procedure arguments.

If we view memory as a big array, the regions (or ``segments'') look as follows:



The figure above shows the memory layout of a Linux process. A process image starts with the program's code and data. Code and data consists of the program's instructions and the initialized and uninitialized static and global data, respectively. After that is the run-time heap (created using malloc/calloc), and then at the top is the users stack. This stack is used whenever a function call is made.

Note, the heap grows down as you make more malloc( ) calls, and the stack goes up as you make nested procedure calls.

Paging

On most machines, memory is broken up into 4096 bytes chunks. These are called pages. To display size of a page in bytes,

```
$ getconf PAGESIZE
```

The way memory works is as follows: The operating system allocates certain pages of memory for you. Whenever you try to read to or write from an address in memory, the hardware first checks with the operating system to see if that address belongs to a page that has been allocated for you. If so, then it goes ahead and performs the read/write. If not, you'll get a segmentation violation.

This is what happens when you do:

```
s = (char *) 0;
c = *s;
```

When you say "c = *s", the hardware sees that you want to read memory location zero. It checks with the operating system, which says "I haven't allocated the page containing location zero for you". This results in asegmentation violation.

As it turns out, some pages on our machines are void. This means that trying to read to or write from any address from these will result in a segmentation violation.

The next page (starting with address 0x08048000) starts the code segment. This segment ends at the variable &etext. The globals segment starts after the code segment. It goes until the variable &end. The heap starts immediately after &end, and goes up to sbrk(0). The stack ends with address 0xbfffffff. Its beginning changes with the different procedure calls you make. Every page between the end of the heap and the beginning of the stack is void, and will generate a segmentation violation upon accessing.

&etext and &end

These are two external variables that are defined as follows:

```
extern   etext;
extern   end;
```

Note that they are typeless. You never use just "etext" and end". Instead, you use their addresses -- these point to the end of the text and globals segments respectively.

Look at the Program 2. This prints out the addresses of etext and end. Then it prints out 6 values:

```
#include <stdio.h>

extern  end;
extern  etext;

extern int  I;
extern int  J;

int  I;

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

     printf("&etext = 0x%lx\n", &etext);
     printf("&end = 0x%lx\n", &end);

     printf("\n");
     ii = (int *) malloc(sizeof(int));

     printf("main = 0x%lx\n", main);
     printf("&I = 0x%lx\n", &I);
     printf("&i = 0x%lx\n", &i);
     printf("&argc = 0x%lx\n", &argc);
     printf("&ii = 0x%lx\n", &ii);
     printf("ii = 0x%lx\n", ii);

}
```

```
/*
You can type cast as
follows to avoid warnings.
(long unsigned int)&xxx)
*/
```

Program 2.

main is a pointer to the first instruction of the main() procedure. This is simply a location in the code segment. I is a global variable. Thus &I should be an address in the globals segment. i is a local variable. Thus &i should be an address in the stack. argc is an argument to main(). Thus, &argc should be an address in the stack. ii is another local variable. Thus, &ii should be an address in the stack. However, ii is a pointer to memory that has been malloc'd. Thus, ii should be an address in the heap.

When we run Program 2, we get something like the following:

```
&etext = 0x80485b8
&end = 0x80497e0

main = 0x8048424
&I = 0x80497dc
&i = 0xbfb496ec
&argc = 0xbfb49700
&ii = 0xbfb496e8
ii = 0x8aea008
```

So, what this says is that the code segment goes from 0x08048000 to 0x080485b8. The globals segment goes to 0x080497e0. The heap goes from 0x080497e0 to some address greater than 0x08aea008 (since ii allocated 4 bytes starting at 0x08aea008). The stack goes from some address less than 0xc0000000. All values that are printed by Program 2 make sense.

Now, look at Program 3.

```
#include <stdio.h>

extern end;
extern etext;

main( )
{
   char *s;
   char c;

   printf("&etext = 0x%lx\n", &etext);
   printf("&end = 0x%lx\n", &end);

   printf("\n");

   printf("Enter memory location in hex (start with 0x): ");
   fflush(stdout);

   scanf("0x%x", &s);

   printf("Reading 0x%x: ", s);
   fflush(stdout);
   c = *s;
   printf("%d\n", c);
   printf("Writing %d back to 0x%x: ", c, s);
   fflush(stdout);
   *s = c;
   printf("ok\n");

}
```

Program 3.

This is the first really gross piece of C code that you'll see. What it does is print out &etext and &end, and then prompt the user for an address in hexidecimal. It puts that address into the pointer variable s. **You should never do this unless you are writing code like this which is testing memory.** The first thing that it does with s is try to read from that memory location (c = *s). Then it tries to write to the memory location (*s = c). This is a way to see which memory locations are legal.

So, lets try it out with an illegal memory value of zero:

```
$ ./program3

    &etext = 0x8048678
    &end = 0x80498cc

    Enter memory location in hex (start with 0x): 0x0
    Reading 0x0: Segmentation Fault
```

When we tried to read from memory location zero, we got a Segmentation fault. This is because memory location zero is in the void -- the hardware recognized this by asking the operating system, and then generating a segmentation violation.

Memory locations 0x0 to 0x08048000 are illegal -- if we try any address in that range, we will get a segmentation violation:

```
$ program3
    &etext = 0x8048678
    &end = 0x80498cc

    Enter memory location in hex (start with 0x): 0x8044444
    Reading 0x80444444: Segmentation Fault


$ program3
    &etext = 0x8048678
    &end = 0x80498cc

    Enter memory location in hex (start with 0x): 0x8000000
    Reading 0x8000000: Segmentation Fault
```

Memory location 0x8048000 is in the code segment. This should be a legal address:

```
$ program3
    &etext = 0x8048678
    &end = 0x80498cc

    Enter memory location in hex (start with 0x): 0x8048000
    Reading 0x8048000: 127
    Writing 127 back to 0x8048000: Segmentation Fault
```

You'll note that we were able to read from 0x8048000 -- it gave us the byte 127, which begins some instruction in the program. However, we got a segmentation fault when we wrote to 0x8048000. This is by design: The code segment is read-only. You can read from it, but you can't write to it. This makes sense, because you can't change your program while it's running -- instead you have to recompile it, and rerun it.

Now, what if we try memory location 0x8048700? This is above &etext, so it should be outside of the code segment:

```
$ program3
     &etext = 0x8048678
     &end = 0x80498cc

     Enter memory location in hex (start with 0x): 0x8048000
     Reading 0x8048700: 127
     Writing  127  back  to  0x8048700:  Segmentation  Fault
```

You'll note that even though 0x8048700 is an address outside the code segment, we're still allowed to read from it. This is because the hardware checks the with operating system to see if an address's page has been allocated. Since page 32840 (0x8048000 - 0x8048fff) has been allocated for the code segment, the hardware treats any address between 0x8048000 and 0x8048fff as a legal address. You can read from it, but its value is meaningless.

The globals starts at 0x8049000, so we see that the 32841th page is readable and writable:

```
$ program3
     &etext = 0x8048678
     &end = 0x80498cc

     Enter memory location in hex (start with 0x): 0x8049000
     Reading 0x8049000: 127
     Writing 127 back to 0x8049000: ok
```

We can read from and write to any location (0x8049000 to 0x8049fff) in this page. The next page (starting at 0x804a000) is unreachable:

```
$ program3
     &etext = 0x8048678
     &end = 0x80498cc

     Enter memory location in hex (start with 0x): 0x8049fff
     Reading 0x8049fff: 0
     Writing 0 back to 0x8049fff: ok

$ program3
     &etext = 0x8048678
     &end = 0x80498cc

     Enter memory location in hex (start with 0x): 0x804a000
Reading 0x804a000: Segmentation Fault
```

What this tells us is that the globals go from 0x8049000 to 0x80498cc. The heap goes from 0x80498cc up to some higher address in the same page.

Sbrk(0)

sbrk( ) is a system call. sbrk(0) returns to the user the current end of the heap. Since we can keep calling malloc(), sbrk(0) can change over time. Program 4 shows the value of sbrk(0) :

```
#include <stdio.h>

extern end;
extern etext;

main( )
{
  char *s;
  char c;

  printf("&etext = 0x%lx\n", &etext);
  printf("&end = 0x%lx\n", &end);
  printf("sbrk(0)= 0x%lx\n", sbrk(0));
  printf("&c = 0x%lx\n", &c);

  printf("\n");

  printf("Enter memory location in hex (start with 0x): ");
  fflush(stdout);

  scanf("0x%x", &s);

  printf("Reading 0x%x: ", s);
  fflush(stdout);
   c = *s;
  printf("%d\n", c);
  printf("Writing %d back to 0x%x: ", c, s);
  fflush(stdout);
  *s = c;
  printf("ok\n");
}
```

Program 4.

```
$ program4
     &etext = 0x80486e8
     &end = 0x804996c
     sbrk(0)= 0x80b5000
     &c = 0xbf89049b

     Enter memory location in hex (start with 0x): 0x8049000
     Reading 0x8049000: 127
     Writing 127 back to 0x8049000: ok
```

The Stack

So, where's the beginning of the stack? If we try addresses near the address of c, we see that most of them are legal:

```
$ program4
     &etext = 0x80486e8
     &end = 0x804996c
     sbrk(0)= 0x8aca000
     &c = 0xbfa62b37

     Enter memory location in hex (start with 0x): 0xbfa62b00
     Reading 0xbfa62b00: 24
     Writing 24 back to 0xbfa62b00: ok

$ program4
     &etext = 0x80486e8
     &end = 0x804996c
     sbrk(0)= 0x978f000
     &c = 0xbfc88807

     Enter memory location in hex (start with 0x): 0xbfc88000
     Reading 0xbfc88000: 0
     Writing 0 back to 0xbfc88000: ok

$ program4
     &etext = 0x80486e8
     &end = 0x804996c
     sbrk(0)= 0x9555000
     &c = 0xbfb268a7

     Enter memory location in hex (start with 0x): 0xbfb27000
     Reading 0xbfb27000: 0
     Writing 0 back to 0xbfb27000: ok
```

You can print out the default stack size, and change it using the ulimit command (man page):

```
$ ulimit -a
     core file size          (blocks, -c) 0
     data seg size           (kbytes, -d) unlimited
     scheduling priority             (-e) 0
     file size               (blocks, -f) unlimited
     pending signals                 (-i) 7907
     max locked memory       (kbytes, -l) 64
     max memory size         (kbytes, -m) unlimited
     open files                      (-n) 1024
     pipe size            (512 bytes, -p) 8
     POSIX message queues     (bytes, -q) 819200
     real-time priority              (-r) 0
     stack size              (kbytes, -s) 8192
     cpu time               (seconds, -t) unlimited
     max user processes              (-u) 1024
     virtual memory          (kbytes, -v) unlimited
     file locks                      (-x) unlimited
     Using semaphores - a short example - Serialisability
     Problem
```

Whenever you call a procedure, it allocates local variables and arguments (plus a few other things) on the stack. Whenever you return from a procedure, those variables are popped off the stack.

So, look at Program 5. It has main() call itself recursively as many times as there are arguments. You'll see that at each recursive call, the addresses of argc and argv and the local variable i are smaller addresses -- this is because each time the procedure is called, the stack grows downward to allocate its arguments and local variables.

```
#include <stdio.h>

extern end;
extern etext;

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

      printf("argc = %d. &argc = 0x%x, &argv = 0x%x, &i = 0x%x\n",
      argc, &argc, &argv, &i);

      if (argc > 0) main(argc-1, argv);
}
```

Program 5.

```
$ program5
argc = 1. &argc = 0xbfe4abf0, &argv = 0xbfe4abf4, &i = 0xbfe4abdc
argc = 0. &argc = 0xbfe4abb0, &argv = 0xbfe4abb4, &i = 0xbfe4ab9c

$ program5 v
argc = 2. &argc = 0xbff85a20, &argv = 0xbff85a24, &i = 0xbff85a0c
argc = 1. &argc = 0xbff859e0, &argv = 0xbff859e4, &i = 0xbff859cc
argc = 0. &argc = 0xbff859a0, &argv = 0xbff859a4, &i = 0xbff8598c

$ program5 v o l s
argc = 5. &argc = 0xbfff6210, &argv = 0xbfff6214, &i = 0xbfff61fc
argc = 4. &argc = 0xbfff61d0, &argv = 0xbfff61d4, &i = 0xbfff61bc
argc = 3. &argc = 0xbfff6190, &argv = 0xbfff6194, &i = 0xbfff617c
argc = 2. &argc = 0xbfff6150, &argv = 0xbfff6154, &i = 0xbfff613c
argc = 1. &argc = 0xbfff6110, &argv = 0xbfff6114, &i = 0xbfff60fc
argc = 0. &argc = 0xbfff60d0, &argv = 0xbfff60d4, &i = 0xbfff60bc
```

Now, lets break the stack. This can be done by writing a program that allocates too much stack memory. One such program is in breakstack.c. It performs infinite recursion, and at each recursive step it allocates 10000 bytes of stack memory in the variable iptr. When you run this, you'll see that you get a segmentation violation when the recursive call is made and the stack is about to dip:

```
#include <stdio.h>

extern end;
extern etext;

main( )
{
      char c;
      char iptr[10000];

      printf("&c = 0x%lx, iptr = 0x%x ... ", &c, iptr);
      fflush(stdout);
      c = iptr[0];
      printf("ok\n");
      main( );
}
```

Program 6.

```
$ program6
. . .
&c = 0xbf363a4f, iptr = 0xbf36133f ... ok
&c = 0xbf36130f, iptr = 0xbf35ebff ... ok
&c = 0xbf35ebcf, iptr = 0xbf35c4bf ... ok
&c = 0xbf35c48f, iptr = 0xbf359d7f ... ok
Segmentation fault (core dumped)
```