

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
Information and Computer Science Department
ICS 431 OPERATING SYSTEMS
Lab # 11
Memory Management

Objective:

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) that 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.

Introduction:

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:

```
char *malloc(unsigned 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 **char** pointer. Often you don't want to **malloc** characters, but structures etc. You therefore must *coerce* the pointer returned from **malloc** into the type you need. For example:

```
/* 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 */

/* Create enough memory to hold a client */
c= (struct client *) malloc( sizeof(struct 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:

```
void free(char *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. After logging into **Unix server**, run the following **ps** command:

```
vlsi> ps -o user,vsz,rss,pmem,fname -e | more
```

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.

Using **df** to see Swap Usage:

With a paging virtual memory system using LRU, those least recently used pages are swapped out to disk until they are required again (if ever). Under Solaris this *swap space* is also used to keep temporary files in the directory `/tmp`. To see the amount of *swap space* in use, use the command:

```
vlsi> df /tmp
```

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.

```
vlsi> 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:

swap:

Amount of swap space currently available in Kbytes.

free:

Size of the free list of pages in Kbytes.

pi:

Kilobytes paged in per second. These are pages which are required for processes to continue execution.

po:

Kilobytes paged out per second. These are LRU unused pages which can be paged to the disk.

fr:

Kilobytes freed due to pageouts or to process termination.

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:

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

Sharing Memory:

Because UNIX runs on page architectures, it can use page protections to share sections of memory *read-only* between processes. For example, the text section for all **kshs** is shared read-only. Another use of **page sharing** is for *shared libraries*. These are subroutines, which are common to many programs. The **printf()** function is used by nearly all C programs, and so it makes sense to load it once into memory, and share its page amongst all C processes.

The **ldd** command can show you what shared libraries each program uses:

```
vlsi> which ls      # Where on disk is ls kept?
/bin/ls

vlsi> ldd /bin/ls   # Show the shared libraries used
libc.so.1 => /usr/lib/libc.so.1
libdl.so.1 => /usr/lib/libdl.so.1

vlsi> size /usr/lib/libc.so.1 # Size of the shared library?
670256 + 25284 + 6500 = 702040

vlsi> ldd a.out
libpthread.so.1 => /usr/lib/libpthread.so.1
libc.so.1 => /usr/lib/libc.so.1
libdl.so.1 => /usr/lib/libdl.so.1
libthread.so.1 => /usr/lib/libthread.so.1
```

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 0xffffe034, it points to the 0xffffe035th 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 that memory location is illegal.

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

```

/* Lab0.c */
main( )
{
    char *s;
    char c;

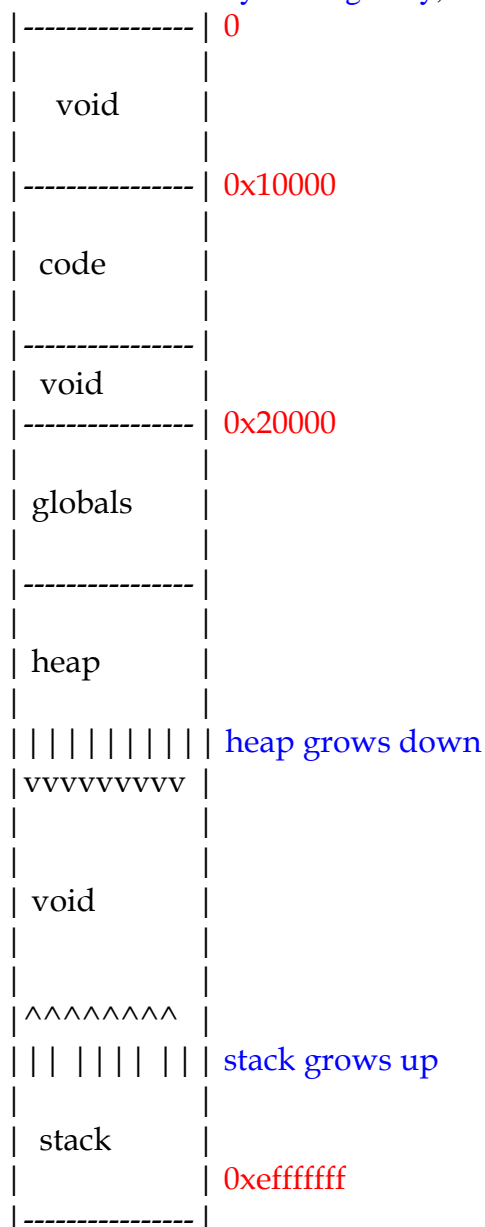
    s = (char *) 0;
    c = *s;
}

```

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:



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 8192-byte chunks. These are called pages. On some machines, pages are 4096 bytes -- this is something set by the hardware.

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 a segmentation violation.

As it turns out, the first 8 pages on our machines are void. This means that trying to read to or write from any address from 0 to 0xffff will result in a segmentation violation.

The next page (starting with address 0x10000) starts the code segment. This segment ends at the variable `&etext`. The globals segment starts at page 0x20000. It goes until the variable `&end`. The heap starts immediately after `&end`, and goes up to `sbrk(0)`. The stack ends with address 0xffffffff. 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 `lab1.c`. This prints out the addresses of `etext` and `end`. Then it prints out 6 values:

```
/* Lab1.c */

#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);
}

```

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 `Lab1.c`, we get something like the following:

```

vlsi> testaddr1
&etext = 0x10b64
&end = 0x20cf0

main = 0x1095c
&I = 0x20ce8
&i = 0xffbefbac
&argc = 0xffbefc04
&ii = 0xffbefba8
ii = 0x20d00

```

So, what this says is that the **code segment** goes from `0x10000` to `0x10b64`. The **globals segment** goes from `0x20000` to `0x20cf0`. The **heap** goes from `0x20cf0` to some address greater than `0x20d00` (since `ii` allocated 4 bytes starting at `0x20d00`). The **stack** goes from some address less than `0xefff8f8` to `0xffffffff`. All values that are printed by `lab12_1.c` make sense.

Now, look at Lab2.c.

```
/* Lab2.c */

#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");
}
```

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, let's try it out with an illegal memory value of zero:

```
vlsi> Lab2
```

```
&etext = 0x10c0c
&end = 0x20ee8
```

```
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 **0xffff** are **illegal** -- if we try any address in that range, we will get a **segmentation violation**:

```
vlsi> Lab2
&etext = 0x10c0c
&end = 0x20ee8
```

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

```
vlsi> Lab2
&etext = 0x10c0c
&end = 0x20ee8
```

```
Enter memory location in hex (start with 0x): 0x4abc
Reading 0x4abc: Segmentation Fault
```

Memory location **0x10000** is in the code segment. This should be a legal address:

```
vlsi> Lab2
&etext = 0x10c0c
&end = 0x20ee8
```

```
Enter memory location in hex (start with 0x): 0x10000
Reading 0x10000: 127
Writing 127 back to 0x10000: Segmentation Fault
```

You'll note that we were able to read from **0x10000** -- it gave us the **byte 127**, which begins some instruction in the program. However, we got a **segmentation fault** when we wrote to **0x10000**. 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 **0x11fff**? This is above **&etext**, so it should be outside of the code segment:

```
vlsi> Lab2
&etext = 0x10c0c
&end = 0x20ee8
```

```
Enter memory location in hex (start with 0x): 0x11fff
Reading 0x11fff: 0
Writing 0 back to 0x11fff: Segmentation Fault
```

You'll note that even though **0x11fff** is an address outside the **code segment**, we're still allowed to read from it. This is because the hardware checks with the operating system to see if an address's page has been allocated. Since **page 8 (0x10000 - 0x11fff)** has been allocated for the **code segment**, the hardware treats any address between **0x10000** and **0x11fff** as a **legal address**.

You can read from it, but its value is meaningless.

Now, pages 9 to 15 are unreadable again:

```
vlsi> Lab2
&etext = 0x10c0c
&end = 0x20ee8
```

```
Enter memory location in hex (start with 0x): 0x12000
Reading 0x12000: Segmentation Fault
```

```
vlsi> Lab2
&etext = 0x10c0c
&end = 0x20ee8
```

```
Enter memory location in hex (start with 0x): 0x1f000
Reading 0x1f000: Segmentation Fault
```

The globals starts at 0x20000, so we see that the 16th page is readable and writable:

```
vlsi> Lab2
&etext = 0x10c0c
&end = 0x20ee8
```

```
Enter memory location in hex (start with 0x): 0x20000
Reading 0x20000: 127
Writing 127 back to 0x20000: ok
```

We can read from and write to any location (0x20000 to 0x21fff) in this page. The next page (starting at 0x22000) is unreachable:

```
vlsi> Lab2
&etext = 0x10c0c
&end = 0x20ee8
```

```
Enter memory location in hex (start with 0x): 0x21dff
Reading 0x21dff: 0
Writing 0 back to 0x21dff: ok
```

```
vlsi> Lab2
&etext = 0x10c0c
&end = 0x20ee8
```

```
Enter memory location in hex (start with 0x): 0x22000
Reading 0x22000: Segmentation Fault
```

What this tells us is that the **globals** go from 0x20000 to 0x20ee8. The **heap** goes from 0x20ee8 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**. `testaddr3.c` shows the value of `sbrk(0)` -- note it is in **page 16** (**0x20000 - 0x21fff**). Since the hardware performs its check in **8192-byte** intervals, we can get at any byte in **page 16**, even though `sbrk(0)` returns **0x20c78**:

```
/* Lab3.c */

#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");
}
```

vlsi> Lab3

&etext = 0x10c84

&end = 0x20f68

sbrk(0)= 0x20f68

&c = 0xffbefbab

Enter memory location in hex (start with 0x): 0x21fff

Reading 0x21fff: 0

Writing 0 back to 0x21fff: ok

We haven't called `malloc()` in `Lab3.c`. This is the reason why `&end` and `sbrk(0)` return the same value. In `Lab3a.c` we make a `malloc()` call in the beginning of the program, and as you see, `&end` and `sbrk(0)` return different values:

```
/* Lab3a.c */

#include <stdio.h>

extern end;
extern etext;

main( )
{
    char *s;
    char c;
    char *buf;

    buf = (char *) malloc(1000);

    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");
}
```

```
vlsi> Lab3a
&etext = 0x10cc4
&end = 0x20fb8
sbrk(0)= 0x22fb8
&c = 0xffbefbab
```

```
Enter memory location in hex (start with 0x): 0x23fff
Reading 0x23fff: 0
Writing 0 back to 0x23fff: ok
```

```
vlsi> Lab3a
&etext = 0x10cc4
&end = 0x20fb8
sbrk(0)= 0x22fb8
&c = 0xffbefbab
```

Enter memory location in hex (start with 0x): 0x24000

Reading 0x24000: Segmentation Fault

The Stack:

So, where's the beginning of the [stack](#)? If we try addresses above [0xffbee103](#) in [Lab3.c](#), we see that most of them are legal:

```
vlsi> Lab3
&etext = 0x10c84
&end = 0x20f68
sbrk(0)= 0x20f68
&c = 0xffbefbab
```

Enter memory location in hex (start with 0x): 0xffb00000

Reading 0xffb00000: 0

Writing 0 back to 0xffb00000: ok

```
vlsi> Lab3
&etext = 0x10c84
&end = 0x20f68
sbrk(0)= 0x20f68
&c = 0xffbefbab
```

Enter memory location in hex (start with 0x): 0xff3f0000

Reading 0xff3f0000: 0

Writing 0 back to 0xff3f0000: ok

```
vlsi> Lab3
&etext = 0x10c84
&end = 0x20f68
sbrk(0)= 0x20f68
&c = 0xffbefbab
```

Enter memory location in hex (start with 0x): 0xff3effff

Reading 0xff3effff: Segmentation Fault

What gives? As it turns out, the operating system allocates all pages from [0xff3f0000](#) to the bottom of the [stack](#). Where is the bottom of the stack? Let's probe:

```
vlsi> Lab3
&etext = 0x10c84
&end = 0x20f68
sbrk(0)= 0x20f68
&c = 0xffbefbab
```

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

```
vlsi> Lab3
&etext = 0x10c84
&end = 0x20f68
sbrk(0)= 0x20f68
&c = 0xffbefbab
```

Enter memory location in hex (start with 0x): 0xffbf0000
Reading 0xffbf0000: Segmentation Fault

So the stack goes from 0xff3f0000 to 0xffbefff. That is roughly 8 megabytes.

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

```
vlsi> limit
cputime unlimited
filesize unlimited
datasize unlimited
stacksize 8192 kbytes
coredumpsize 0 kbytes
descriptors 64
memorysize unlimited
```

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 `Lab4.c`. 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.

```
/* Lab4.c */

#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);
}
```

vlsi> Lab4

argc = 1. &argc = 0xffbfc04, &argv = 0xffbfc08, &i = 0xffbefbac
argc = 0. &argc = 0xffbefb8c, &argv = 0xffbefb90, &i = 0xffbefb34

vlsi> Lab4 v

argc = 2. &argc = 0xffbefbfc, &argv = 0xffbfc00, &i = 0xffbefba4
argc = 1. &argc = 0xffbefb84, &argv = 0xffbefb88, &i = 0xffbefb2c
argc = 0. &argc = 0xffbefb0c, &argv = 0xffbefb10, &i = 0xffbefab4

vlsi> Lab4 v o l s

argc = 5. &argc = 0xffbefbec, &argv = 0xffbefbf0, &i = 0xffbefb94
argc = 4. &argc = 0xffbefb74, &argv = 0xffbefb78, &i = 0xffbefb1c
argc = 3. &argc = 0xffbefafc, &argv = 0xffbefb00, &i = 0xffbefaa4
argc = 2. &argc = 0xffbefa84, &argv = 0xffbefa88, &i = 0xffbefa2c
argc = 1. &argc = 0xffbefa0c, &argv = 0xffbefa10, &i = 0xffbef9b4
argc = 0. &argc = 0xffbef994, &argv = 0xffbef998, &i = 0xffbef93c

Now, lets **break the stack**. Writing a program that allocates too much stack memory can do this. 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 below **0xff3f0000**:

```
/* breakstack.c */  
  
#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( );  
}
```

vlsi> breakstack

...

&c = 0xff3fa347, iptr = 0xff3f7c30 ... ok
&c = 0xff3f7bbf, iptr = 0xff3f54a8 ... ok
&c = 0xff3f5437, iptr = 0xff3f2d20 ... ok
Segmentation Fault

Exercises

Note:

Lab Problems will be given during the lab based on material covered in this lab manual.