Lab 6: System calls

Objectives

- To become more familiar with the Intel x86 assembly language.
- To practice the use of interrupts for initiating system calls to Linux.
- To learn about how parameters typed on a command line are given to a program under Linux.

Assignment

The *getcs* command will copy two files into your directory: `mycat.s`, which contains the x86 code on page 99 of the text, which is a helpful starting point for for this assignment; and `debug.s`, which includes the printregs and printstack routines for debugging.

Your job is to develop a Linux assembly program that echoes its input to the output.

```
% ./a.out
I love x86 assembly!
I love x86 assembly!
Don't you?
Don't you?
control-D
```

When you type control-D, this signals an end to the keyboard file. Your program can identify the end of a file by the fact that the `read` system call returns 0 when it knows there are no more characters left in the file.

This program should also be able to handle a command-line argument naming a separate file. (The next section of this handout discusses how command-line arguments work.) If the file `tmp` contains

```
I love x86 assembly!
Don't you?
```

Then the user typing "`./a.out tmp`" should echo the file.

```
% ./a.out tmp
I love x86 assembly!
Don't you?
```

You should check to make sure that your solution works with very large files — too large to fit into whatever buffer you are using.

The program should handle errors gracefully. Namely, if the user enters a bad command line (with too many arguments), it should report a meaningful error message. And if the named file isn't found, it should also report a meaningful error message.

Your program must use direct system calls exclusively. It must not use `printf` or other library subroutines, including those that merely wrap around system calls.

Your lab report for this lab need only the commented assembly code you wrote, plus the normal introduction and conclusion. Run *handincs* to submit your solution.

Command-line parameters

Before Linux enters the `main()` function of a program, it pushes two parameters on the stack as parameters for gaining access to the command-line parameters.

```
int main(int argc, char **argv) {
```

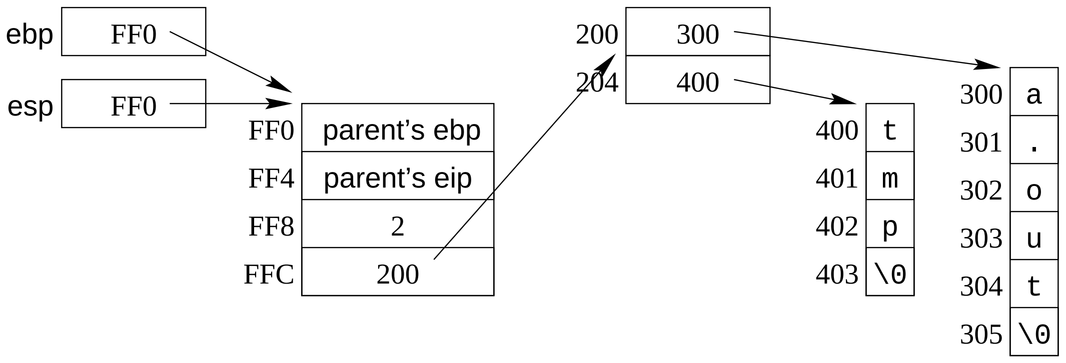

Figure 6.1: Diagram of memory containing the `argv` parameter.

The first argument is the number of command-line arguments, including the command name itself as an argument. The second argument is a pointer to an array of pointers, each array element pointing to the first letter of a NUL-terminated string holding the argument.

If we run "`a.out tmp`" at the command line, then after executing the entry template at the beginning of the assembly code for `main`, the memory will look like the diagram of Figure 6.1. (The memory addresses appearing in the diagram will vary each time the program executes.)