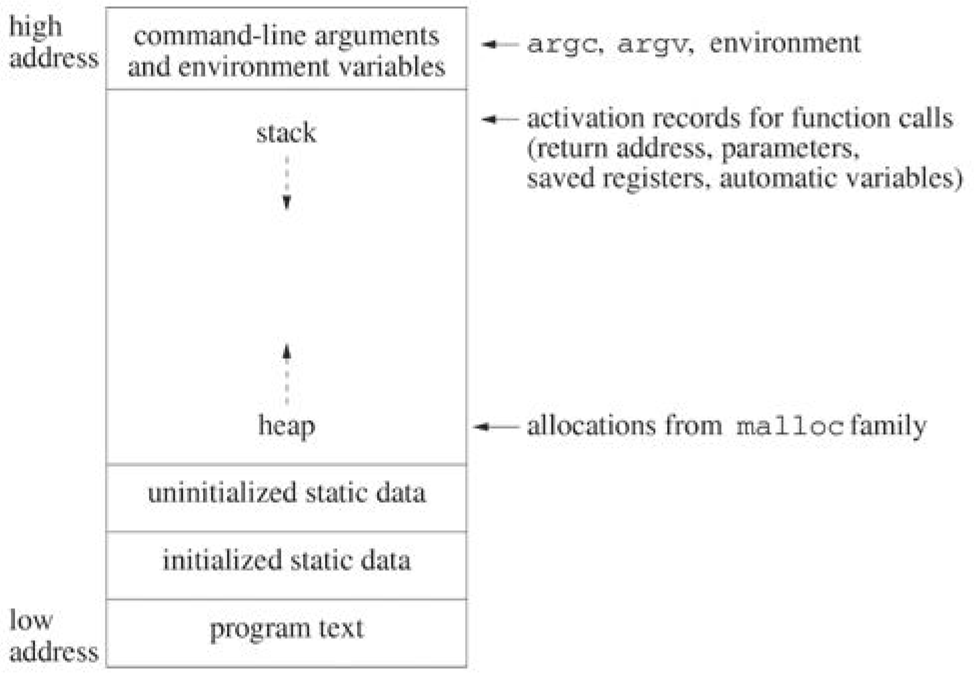
Lab 02: Layout of Program, Library Function Calls and error handling

2.1. Layout of a Program Image


high address
command-line arguments and environment variables
argc, argv, environment
stack
activation records for function calls (return address, parameters, saved registers, automatic variables)
heap
allocations from malloc family
uninitialized static data
initialized static data
low address
program text

After loading, the program executable appears to occupy a contiguous block of memory called a program image. Above figure shows a sample layout of a program image in its logical address space. The program image has several distinct sections. The program text or code is shown in low-order memory. The initialized and uninitialized static variables have their own sections in the image. Other sections include the heap, stack and environment.

An activation record is a block of memory allocated on the top of the process stack to hold the execution context of a function during a call. Each function call creates a new activation record on the stack. The activation record is removed from the stack when the function returns, providing the last-called-first-returned order for nested function calls.

The activation record contains the return address, the parameters (whose values are copied from the corresponding arguments), status information and a copy of some of the CPU register values at the time of the call. The process restores the register values on return from the call represented by the record. The activation record also contains automatic variables that are allocated within the function while it is executing. The particular format for an activation record depends on the hardware and on the programming language.

In addition to the static and automatic variables, the program image contains space for `argc` and `argv` and for allocations by `malloc`. The `malloc` family of functions allocates storage from a free memory pool called the heap. Storage allocated on the heap persists until it is freed or until the program exits. If a function calls `malloc`, the storage remains allocated after the function returns. The program cannot access the storage after the return unless it has a pointer to the storage that is accessible after the function returns.

Static variables that are not explicitly initialized in their declarations are initialized to 0 at run time. Notice that the initialized static variables and the uninitialized static variables occupy different sections in the program image. Typically, the initialized static variables are part of the executable module on disk, but the uninitialized static variables are not. Of course, the automatic variables are not part of the executable module because they are only allocated when their defining block is called. The initial values of automatic variables are undetermined unless the program explicitly initializes them.

Task 1: Write two C programs for the analysis of the difference in executable file sizes while using:

1. **Uninitialized Static Variables**
2. **Initialized Static Variables**

Use `ls -l` to compare the sizes of the executable modules for the above two C programs.

2.2. Return Values and Error Handling

Traditional UNIX functions usually return –1 (or sometimes `NULL`) and set `errno` to indicate the error. The POSIX standards committee decided that all new functions would not use `errno` and would instead directly return an error number as a function return value.

2.2.1. Perror:

```
SYNOPSIS

   #include <stdio.h>
   void perror(const char *s);

No return values and no errors are defined for perror.
```

The `perror` function outputs to standard error a message corresponding to the current value of `errno`. If `s` is not `NULL`, `perror` outputs the string (an array of characters terminated by a null character) pointed to by `s` and followed by a colon and a space. Then, `perror` outputs an error message corresponding to the current value of `errno` followed by a newline.

2.2.2. Strerror:

The `strerror` function returns a pointer to the system error message corresponding to the error code `errnum`.

```
SYNOPSIS

   #include <string.h>
   char *strerror(int errnum);
```

If successful, `strerror` returns a pointer to the error string. No values are reserved for failure.

Use `strerror` to produce informative messages, or use it with functions that return error codes directly without setting `errno`.

The `strerror` function may change `errno`. You should save and restore `errno` if you need to use it again.

Correctly handing `errno` is a tricky business. Because its implementation may call other functions that set `errno`, a library function may change `errno`, even though the man page doesn't explicitly state that it does. Also, applications cannot change the string returned from `strerror`, but subsequent calls to either `strerror` or `perror` may overwrite this string.

Another common problem is that many library calls abort if the process is interrupted by a signal. Functions generally report this type of return with an error code of `EINTR`. For example, the `close` function may be interrupted by a signal. In this case, the error was not due to a problem with its execution but was a result of some external factor. Usually the program should not treat this interruption as an error but should restart the call. The problem of restarting library calls is so common that a library of restarted calls is provided with prototypes defined in `restart.h`. The functions are designated by a leading `r_` prepended to the regular library name. For example, the restart library designates a restarted version of `close` by the name `r_close`.

Task 2: Analyze the return values and error numbers/error stings of *wait* system call on:

1. **Success**
2. **Failure**

Using:

1. **Strerror**
2. **Perror**

```
SYNOPSIS

   #include <sys/wait.h>
   pid_t wait(int *stat_loc);
```

Task 3: Analyze the return values and error numbers/error stings of *close* system call on:

3. **Success**
4. **Failure**

Using:

3. **Strerror**
4. **Perror**

```
SYNOPSIS

   #include <unistd.h>
   int close(int fildes);
```