Advanced Programming in the UNIX Environment — *Process Environment*

Hop Lee
hoplee@bupt.edu.cn

School of Information and Communication Engineering

Table of Contents

main Function

- A C program starts execution with a function called main:

```
1 int main(int argc, char *argv[]);
```

- *argc* is the number of command-line arguments and *argv* is an array of pointers to the arguments.
- When a C program is started by the kernel (by one of the exec functions), a special **start-up routine** is called before the main function is called.
- The executable program file specifies this start-up routine as the starting address for the program. This start-up routine takes values from the kernel and sets things up so that the main function is called as shown earlier.

Process Termination

- There are eight ways for a process to terminate:
  1. Normal termination
     - 1.1 return from `main`
     - 1.2 calling `exit`
     - 1.3 calling `_exit` or `_Exit`
     - 1.4 Return of the last thread from its start routine (Section 11.5)
     - 1.5 Calling `pthread_exit` (Section 11.5) from the last thread
  2. Abnormal termination
     - 2.1 calling `abort`
     - 2.2 terminated by a signal
     - 2.3 Response of the last thread to a cancellation request (Sections 11.5 and 12.7)
- The start-up routine that we mentioned in the previous section is also written so that if the `main` function returns, the `exit` function is called:

```
1 exit(main(argc, argv));
```

Exit Functions I

- These functions terminate a program normally:
  - `_exit`, which returns to the kernel immediately
  - `exit`, which performs certain cleanup processing and then returns to the kernel.

```
1 #include <stdlib.h>
2 void exit(int status);
3 void _Exit(int status);
4 #include <unistd.h>
5 void _exit(int status);
```

- Both these functions expect a single integer argument, which we call the **exit status**.
- If

Exit Functions II

1. either of these functions is called without an exit status
2. `main` does a `return` without a return value
3. `main` falls off the end (an implicit return)

the exit status of the process is undefined.

- Returning an integer value from the `main` function is equivalent to calling `exit` with the same value.
- Example (Figure 7.1, `environ/hello1.c`).

atexit Function I

- With ISO C a process can register up to 32 functions that are automatically called by exit. These so called **exit handlers** are registered by atexit function.

```
1 #include <stdio.h>
2 int atexit(void (*func)(void));
```

- When exit handler is called it is not passed any arguments and it is not expected to return a value.
- The exit function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.
- Example (Figure 7.3, environ/doatexit.c).

atexit Function II

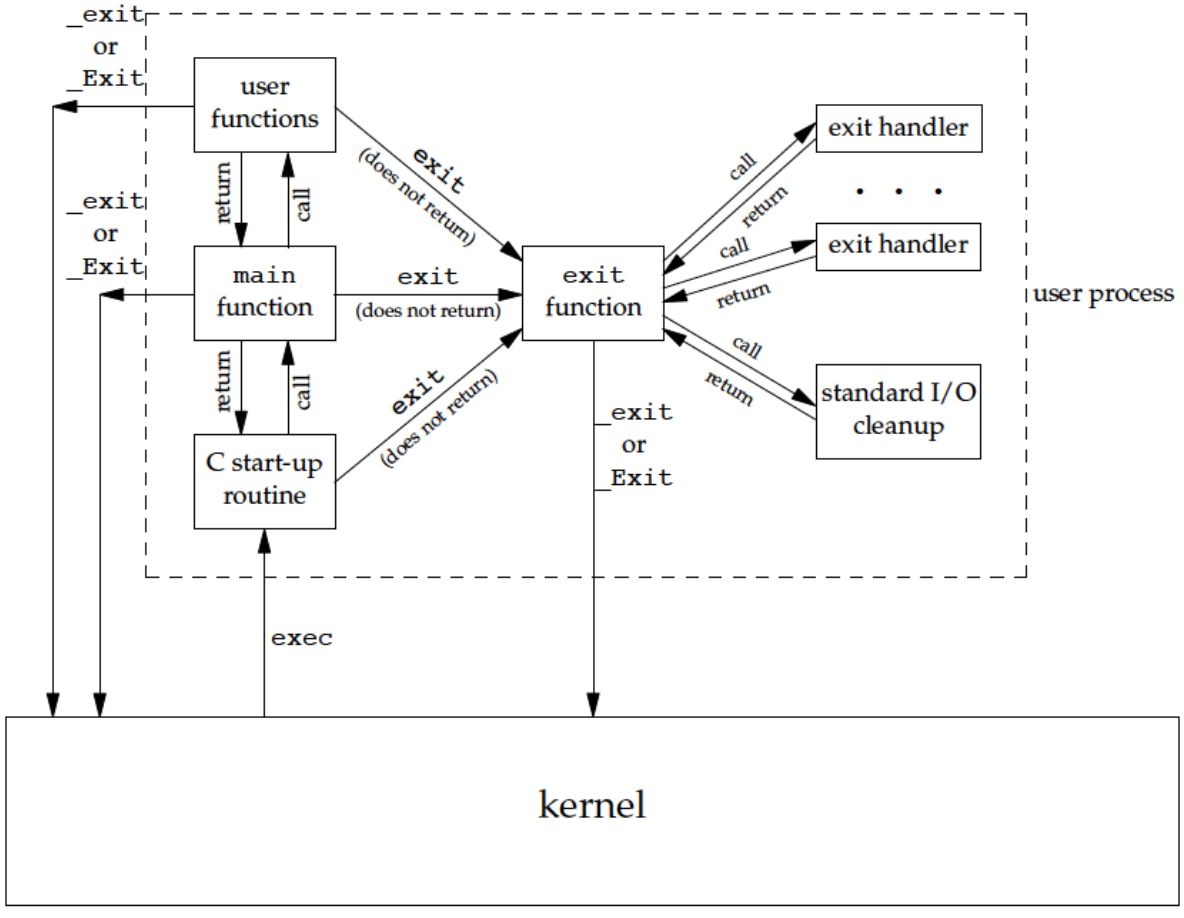

Figure: How a C program is started and how it terminates

Command-Line Arguments

- When a program is executed, the process that does the `exec` can pass command-line arguments to the new program.
- Example (Figure 7.4, `environ/echoarg.c`).
- We are guaranteed by both ISO C and POSIX.1 that *argv[argc]* is a null pointer. This let us alternatively code the argument processing loop as

```
1 for(i = 0; argv[i] != NULL; i++)
```

Environment List I

- Each program is also passed an **environment list**.
- The environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string. The address of the array of pointers is contained in the global variable `environ`

```
1 extern char **environ
```

Environment List II

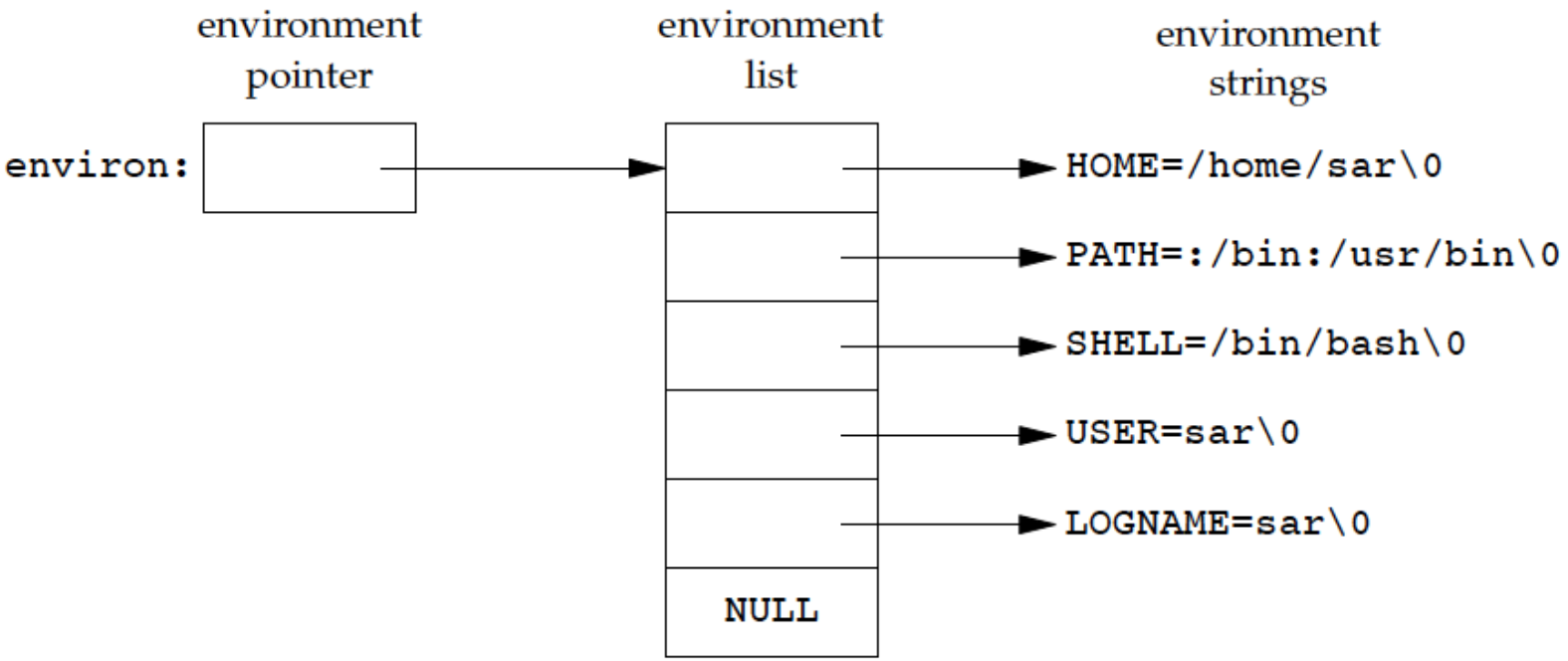

Figure: Environment consisting of five C character strings

Memory Layout of a C Program I

- Historically a C program has been composed of the following pieces:

  Text segment The machine instructions that are executed by the CPU. Usually the text segment is sharable and read only.

  Initialized data segment Also called **data segment** and it contains variables that are specifically initialized in the program.

  Uninitialized data segment Also called **bss segment**, named after "block started by symbol". Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing.

Memory Layout of a C Program II

Stack This is where automatic variables are stored, along with information that is saved each time a function is called.

Heap Dynamic memory allocation usually takes place on the heap. Historically the heap has been located between the top of the uninitialized data and the bottom of the stack.

Arguments and Environments This area contains command-line arguments and working environment-list.

- The `size`(1) command reports the sizes (in bytes) of the text, data, and bss segments.

Memory Layout of a C Program III

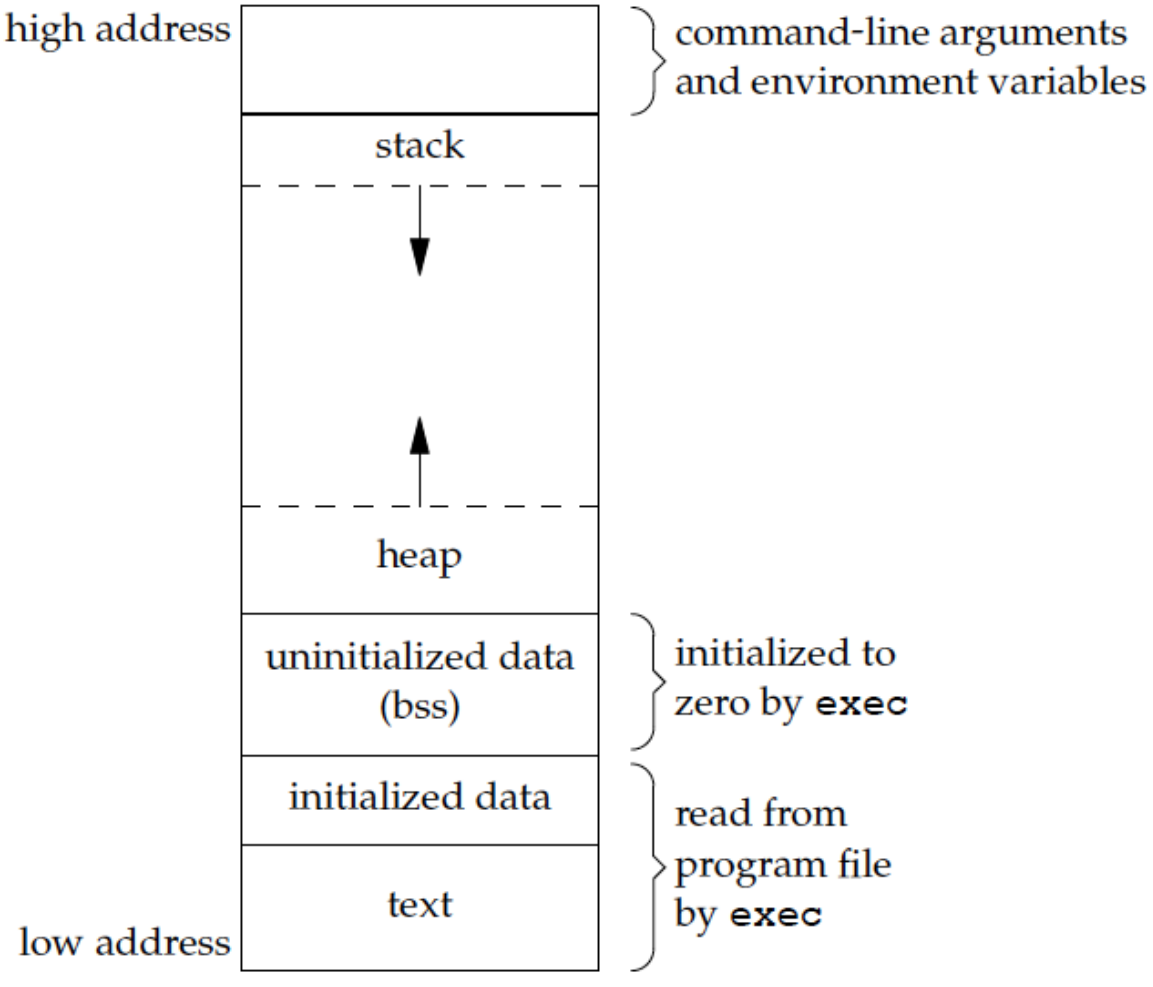

Figure: Typical memory arrangement

Shared Library

- Most modern UNIX implementation support dynamic shared library.
- This technology make it possible that the executable file need NOT include the popular library functions, but put a copy of these routines in an area which is readable for all process.
- This method reduce the length of program file, but increase the run time cost.
- Another advantage of shared libraries is that library functions can be replaced with new versions without having to relink edit every program that uses the library
- Under Linux, `gcc`, use `-static` option will force the program to be static linked.

Memory Allocation I

- There are three functions specified by ISO C for memory allocation.
  - `malloc`. Allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
  - `calloc`. Allocates space for a specified number of objects of a specified size. The space is initialized to all bit 0.
  - `realloc`. Change the size of a previously allocated area (increases or decreases).

```
1 #include <stdlib.h>
2 void *malloc(size_t size);
3 void *calloc(size_t nobj, size_t size);
4 void *realloc(void *ptr, size_t newsize);
5 viod free(void *ptr);
```

Memory Allocation II

- The function `free` causes the space pointed to by *ptr* to be de-allocated.
- Many replacements for `malloc` and `free` are available. Some systems already include libraries providing alternative memory allocator implementations. Other systems provide only the standard allocator, leaving it up to software developers to download alternatives, if desired.

Environment Variables I

- All the environment variable looks like this:

```
1 name=value
```

- The UNIX kernel do not care the meaning of the value. It's explanation depends on the application program.
- ISO C define the `getenv` function:

```
1 #include <stdlib.h>
2 char *getenv(const char *name);
```

It return a pointer to the *value* string.

- We should always access the environment variable through the `getenv` function but not the global variable `environ`.

Environment Variables II

- There are also some other functions used to create new environment variable or alert the value of an exist environment variable or delete certain environment variable.

```
1 #include <stdlib.h>
2 int putenv(char *string);
3 int setenv(const char *name, const char *value,
4             int overwrite);
5 void unsetenv(const char *name);
6 int clearenv(void);
```

Environment Variables III

- The `putenv` function adds or changes the value of environment variables. The argument *string* is of the form *name=value*. If *name* does not already exist in the environment, then string is added to the environment. If *name* does exist, then the value of *name* in the environment is changed to *value*. The string pointed to by *string* becomes part of the environment, so altering the string changes the environment.
- The `setenv` function adds the variable name to the environment with the value *value*, if *name* does not already exist. If *name* does exist in the environment, then its value is changed to *value* if *overwrite* is non-zero; if *overwrite* is zero, then the value of *name* is not changed.

Environment Variables IV

- The `unsetenv` function deletes the variable *name* from the environment.
- The `clearenv` function clears the environment of all *name-value* pairs and sets the value of the external variable `environ` to `NULL`.

setjmp and longjmp Functions I

- We can not use goto statement across the function in C language. But we can use nonlocal goto functions setjmp and longjmp instead.
- They are useful for dealing with errors and interrupts encountered in a deeply nested function call of a program.
- Example (Figure 7.9, environ/cmd1.c).

```
1 #include <setjmp.h>
2 int setjmp(jmp_buf env);
3 void longjmp(jmp_buf env, int val);
```

- longjmp restores the environment saved by the last call of setjmp with the corresponding *env* argument.

setjmp and longjmp Functions II

- After longjmp is completed, program execution continues as if the corresponding call of setjmp had just return the value *val*.
- Example (Figure 7.11, environ/cmd2.c).
- Example (Figure 7.13, environ/testjmp.c).
- Potential problem with automatic variables. The basic rule is that an automatic variable can never be referenced after the function that declared it returns. Figure 7.14 (envirno/opendata.c) shows an example.

getrlimit and setrlimit Functions I

- Every process has a group of resource limits, some of them can be queried and modified by getrlimit and setrlimit functions.

```
1 #include <sys/resource.h>
2 int getrlimit(int resource, struct rlimit *rlim);
3 int setrlimit(int resource,
4                const struct rlimit *rlim);
```

- Each resource has an associated soft and hard limit, as defined by the rlimit structure:

getrlimit and setrlimit Functions II

```
1 struct rlimit {
2     rlim_t rlim_cur;   /* Soft limit */
3     rlim_t rlim_max;   /* Hard limit (ceiling
4                           for rlim_cur) */
5 };
```

- Three rules govern the changing of the resource limits.
  1. A process can change its soft limit to a value less than or equal to its hard limit.
  2. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
  3. Only a superuser process can raise a hard limit.
- An infinite limit is specified by the constant RLIM_INFINITY.

getrlimit and setrlimit Functions III

- The *resource* indicate one of the pre-defined resource name shown in Figure 7.15.
- The resource limits affect the calling process and are inherited by any of its children.
- Example (Figure 7.16, `environ/getrlimit.c`).
- Note that we've used the ISO C string-creation operator (#) in the doit macro, to generate the string value for each resource name.

The End

The End of Chapter 7.