System Programming

Week 5: System Data Files and Process Environment

Seongjin Lee

Updated: 2018/07/06
04-sysfile_info

insight@gnu.ac.kr
http://open.gnu.ac.kr
Systems Research Lab.
Gyeongsang National University

Table of contents

1. Password, Group, and other system files

1.1 Password files

1.2 Group files

1.3 Others System Databases Files

1.4 Time and Date

1.5 Last Words

2. Process Environment

2.1 Process Creation and Termination

2.2 Environments

2.3 Memory

2.4 Misc.

Password, Group, and other system files

introduction

A UNIX system requires numerous data files for normal operations

- `/etc/passwd` - for user log-in or `ls -l`, etc
- `/etc/group`

These data files have been ASCII text files and were read with the standard I/O libaray.

- sequential scan of such files became time consuming for a large system

We want to other data format other than ASCII, but still provide an interface for any application.

Password, Group, and other system files : Password files

Password File

The UNIX System's password file <pwd.h>, called the user database by POSIX.1, contains the following fields

Description	struct passwd member	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
user name	char *pw_name	•	•	•	•	•
encrypted password	char *pw_passwd		•	•	•	•
numerical user ID	uid_t pw_uid	•	•	•	•	•
numerical group ID	gid_t pw_gid	•	•	•	•	•
comment field	char *pw_gecos		•	•	•	•
initial working directory	char *pw_dir	•	•	•	•	•
initial shell (user program)	char *pw_shell	•	•	•	•	•
user access class	char *pw_class		•		•	
next time to change password	time_t pw_change		•		•	
account expiration time	time_t pw_expire		•		•	

Figure: Fileds in `/etc/passwd` file

On your machine, you can see the contents by typing `cat /etc/passwd`

```
1 root:x:0:0:root:/root:/bin/bash
2 squid:x:23:23::/var/spool/squid:/dev/null
3 nobody:x:65534:65534:Nobody:/home:/bin/sh
4 sar:x:205:105:Stephen Rago:/home/sar:/bin/bash
```

Password File cnt'd

- if ecrypted passwd field is empty, it usually means that the user does not have passwd
- shell field contains the name of a program to be used as the login shell for the user. If this field is empty, the default is `/bin/sh`
- `/dev/null`, `/bin/false`, or `/bin/true` in the shell field prevents a particular user from logging in to a system
- nobody user name can be used to allow people to log, but with a user ID (65534) and group ID (65534)

Password File API

```
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);
// Both return: pointer if OK, NULL on error
```

`getpwuid` function is used by `ls(1)` program to map the numberical user ID contained in an i-node into a user's login name

`getpwnam` function is used by the `login(1)` program when we enter our login name

Both functions return a pointer to `passwd` structure that the functions fill in

They are only for looking up either a login name or a user ID

Password File API cnt'd

```
#include <pwd.h>
struct passwd *getpwent(void);
// Returns: pointer if OK, NULL on error or end of file
void setpwent(void);
void endpwent(void);
```

`getpwent` to return the next entry in the password file.

The function `setpwent` rewinds whatever files it uses, and `endpwent` closes these files.

When using `getpwent`, we must always be sure to close these files by calling `endpwent` when we're through.

getpwnam implementation

```
1  #include <pwd.h>
2  #include <stddef.h>
3  #include <string.h>
4
5  struct passwd *
6  getpwnam(const char *name)
7  {
8     struct passwd *ptr;
9
10    setpwent();
11    while ((ptr = getpwent()) != NULL)
12       if (strcmp(name, ptr->pw_name) == 0)
13          break; /* found a match */
14    endpwent();
15    return(ptr); /* ptr is NULL if no match found */
16 }
```

more interested reader can go on and have a look at `getspnam` and `setspent` for shadow password files

Password, Group, and other system files : Group files

Group File

The UNIX System's group file, called the group database by POSIX.1, contains the fields shown in the figure. These fields are contained in a group structure that is defined in <grp.h>.

Description	struct group member	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
group name	char *gr_name	•	•	•	•	•
encrypted password	char *gr_passwd		•	•	•	•
numerical group ID	int gr_gid	•	•	•	•	•
array of pointers to individual user names	char **gr_mem	•	•	•	•	•

Figure: Fields in /etc/group file

The field gr_mem is an array of pointers to the user names that belong to this group. This array is terminated by a null pointer.

Group File API

We can look up either a group name or a numerical group ID with the following

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
// Both return: pointer if OK, NULL on error
```

Like the password file functions, both of these functions normally return pointers to a static variable, which is overwritten on each call.

Followings are used for searching the entire group file

```
#include <grp.h>
struct group *getgrent(void); // returns next group entry in file each time it's
     called, but not in order
// Returns: pointer if OK, NULL on error or end of file
void setgrent(void); // rewinds to the beginning of entries
void endgrent(void); // closes the file
```

Password, Group, and other system files : Others System Databases Files

Other System Database Files

There are other system files and has similar routines for accessing them

Description	Data file	Header	Structure	Additional keyed lookup functions
passwords	/etc/passwd	<pwd.h>	passwd	getpwnam, getpwuid
groups	/etc/group	<grp.h>	group	getgrnam, getgrgid
shadow	/etc/shadow	<shadow.h>	spwd	getspnam
hosts	/etc/hosts	<netdb.h>	hostent	getnameinfo, getaddrinfo
networks	/etc/networks	<netdb.h>	netent	getnetbyname, getnetbyaddr
protocols	/etc/protocols	<netdb.h>	protoent	getprotobyname, getprotobynumber
services	/etc/services	<netdb.h>	servent	getservbyname, getservbyport

Figure: Similar routines for accessing system data files

- **get** function reads the next record, opening the file if necessary
- **set** function that opens the file, if not already open, and rewinds the file
- **end** entry that closes the data file. We always have to call this function when we're done.

System Identification

POSIX.1 defines the uname function to return information on the current host and operating system

```
#include <sys/utsname.h>
int uname(struct utsname *name);
// Returns: non-negative value if OK, 1 on error
```

- We pass the address of a `utsname` structure to this function, and the function then fills it in.
- The structure contains fields like sysname, nodename, release, version, and machine; but, they depend on the implementation
- This is used by the shell command `uname(1)`

```
#include <unistd.h>
int gethostname(char *name, int namelen);
// Returns: 0 if OK, 1 on error
```

Historically, BSD-derived systems provided the gethostname function to return only the name of the host

Password, Group, and other system files : Time and Date

Time and Date Routines

The basic time service provided by the UNIX kernel counts the number of seconds that have passed since the Epoch: 00:00:00 January 1, 1970, Coordinated Universal Time (UTC).

- they are represented in a `time_t` data type
- they are called *calendar times* and represent both the time and the date

The UNIX System has always differed from other operating systems in

- keeping time in UTC instead of the local time
- automatically handling conversions, such as daylight saving time,
- keeping the time and date as a single quantity.

Time and Date Routines API

The `time` function returns the current time and date

```
#include <time.h>
time_t time(time_t *calptr);
// Returns: value of time if OK, 1 on error
```

- The time value is always returned as the value of the function.
- If the argument is non- null, the time value is also stored at the location pointed to by *calptr*

Time and Date Routines cnt'd

The `clock_gettime` function can be used to get the time of the specified clock

```
#include <sys/time.h>
int clock_gettime(clockid_t clock_id, struct timespec *tsp); // get the time
int clock_getres(clockid_t clock_id, struct timespec *tsp); // resolution of a
     system clock
int clock_settime(clockid_t clock_id, const struct timespec *tsp); // set the
     time
// Returns: 0 if OK, 1 on error
```

Identifier	Option	Description
CLOCK_REALTIME		real system time
CLOCK_MONOTONIC	_POSIX_MONOTONIC_CLOCK	real system time with no negative jumps
CLOCK_PROCESS_CPUTIME_ID	_POSIX_CPUTIME	CPU time for calling process
CLOCK_THREAD_CPUTIME_ID	_POSIX_THREAD_CPUTIME	CPU time for calling thread

Figure: Clock type identifiers

Time and Date Routines cnt'd

This function is now obsolete. However, a lot of programs stil use it, because it provides greater resolution (up to a microsecond) than the `time` function

```
int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
// Returns: 0 always
```

Time and Date Routines cnt'd: formating

```
#include <time.h>
size_t strftime(char *restrict buf, size_t maxsize, const char *restrict format,
     const struct tm *restrict tmptr);
size_t strftime_l(char *restrict buf, size_t maxsize, const char *restrict format
     , const struct tm *restrict tmptr, locale_t locale);
// Both return: number of characters stored in array if room, 0 otherwise
```

The strftime and `strftime_l` functions are the same, except that the `strftime_l` function allows the caller to specify the locale as an argument.

Relationship of Time Functions

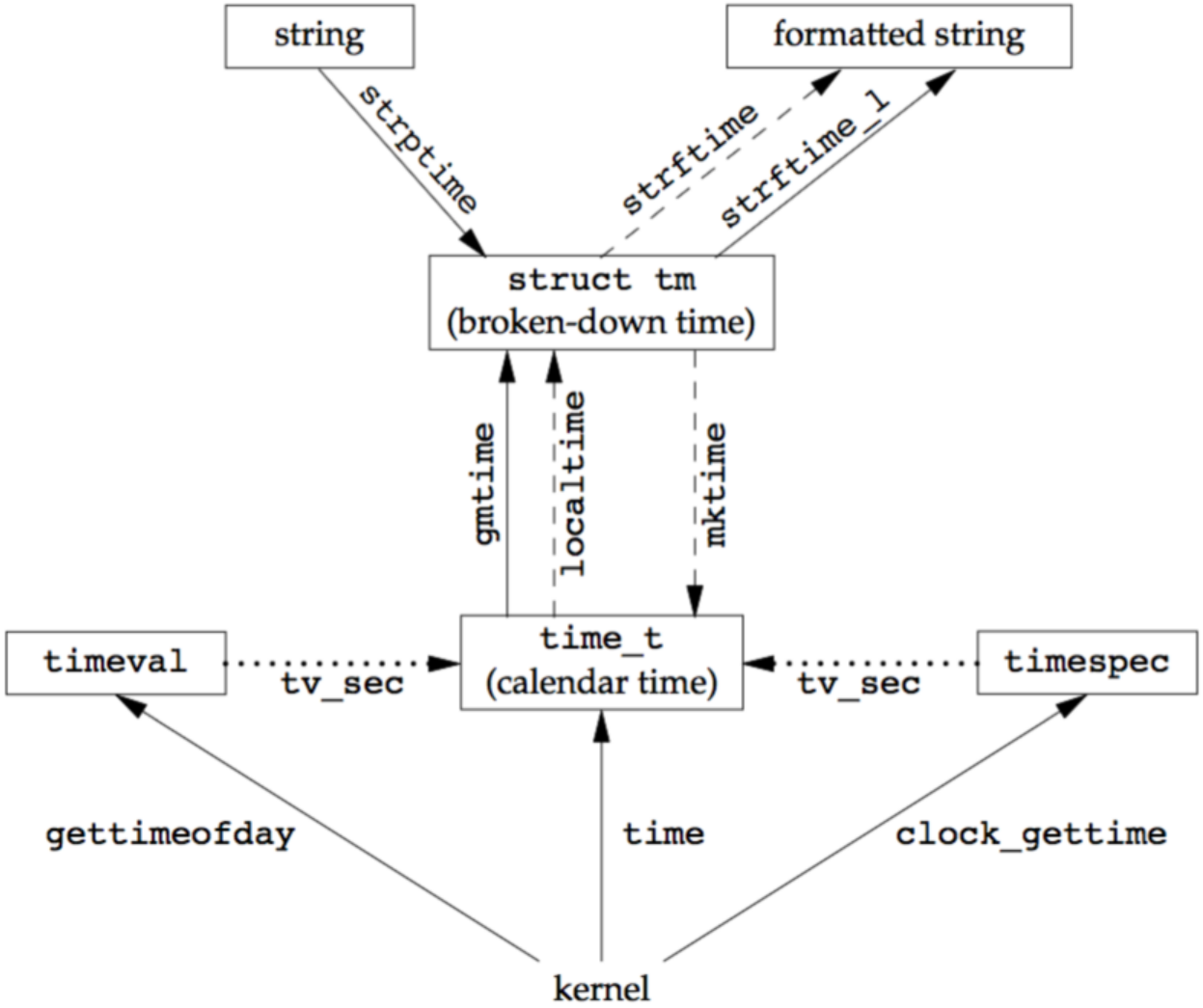

Figure: Relationship of the various time functions

Time and Date Routines cnt'd: tm structure

```
struct tm { /* a broken-down time */
     int tm_sec; /* seconds after the minute: [0 - 60] */
     int tm_min; /* minutes after the hour: [0 - 59] */
     int tm_hour; /* hours after midnight: [0 - 23] */
     int tm_mday; /* day of the month: [1 - 31] */
     int tm_mon; /* months since January: [0 - 11] */
     int tm_year; /* years since 1900 */
     int tm_wday; /* days since Sunday: [0 - 6] */
     int tm_yday; /* days since January 1: [0 - 365] */
     int tm_isdst; /* daylight saving time flag: <0, 0, >0 */
};
```

strftime example

```
make strftime; ./strftime
```

```
1     time_t t;
2     struct tm *tmp;
3     char buf1[16];
4     char buf2[64];
5
6     time(&t);
7     tmp = localtime(&t);
8     if (strftime(buf1, 16, "time and date: %r, %a %b %d, %Y", tmp) == 0)
9        printf("buffer length 16 is too small\n");
10    else
11       printf("%s\n", buf1);
12    if (strftime(buf2, 64, "time and date: %r, %a %b %d, %Y", tmp) == 0)
13       printf("buffer length 64 is too small\n");
14    else
15       printf("%s\n", buf2);
16
17    exit(0);
18 }
```

Password, Group, and other system files : Last Words

Homework

- Download Mobibench code from https://github.com/ESOS-Lab/Mobibench/tree/master/shell
- Measured time is stored in timeval which is in microseconds.
- Change the code to measure the performance in nanoseconds.
- use `diff` to create `patch` file and show your work.
- print out the `patch` file and handed in on the first class of 2016-10-12

How to create a patch file

The original file `foo.c`

```
1  #include <stdio.h>
2
3  int main() {
4  printf("Hello World\n");
5  }
```

Your modified file `foobar.c`

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[]) {
4  printf("Hello World\n");
5  return 0;
6  }
```

Create the patch file using diff command

`diff -u foo.c foobar.c > foobar.patch`

```
1  --- foo.c 2016-09-30 17:04:42.000000000 +0900
2  +++ foobar.c 2016-09-30 17:05:03.000000000 +0900
3  @@ -1,5 +1,6 @@
4  -#include <stdio.h>
5  +#include <stdio.h>
6
7  -int main() {
8  +int main(int argc, char *argv[]) {
9   printf("Hello World\n");
10 -}
11 +return 0;
12 +}
```

Process Environment

Introduction

In this chapter, we'll see

- how the `main` function is called when the program is executed
- called when the program is executed
- how command-line arguments are passed to the new program
- what the typical memory layout looks like
- how to allocate additional memory
- how the process can use environment variables
- and various ways for the process to terminate
- the `longjmp` and `setjmp` functions and their interaction with the stack

We finish the chapter by examining the resource limits of a process.

Process Environment : Process Creation and Termination

`main` Function

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

- *argc*: the number of commands-line arguments
- *argv*: an array of pointers to the arguments

The executable program file specifies this routine as the starting address for the program;

- this is set up by the link editor when it is invoked by the C compiler.
- This start-up routine takes values from the kernel—the command-line arguments and the environment—and sets things up

Process Termination

There are eight ways for a process to terminate

Normal termination

- Return from `main`
- Calling `exit`
- Calling `_exit` or `_Exit`
- Return of the last thread from its start routine
- Calling `pthread_exit` from the last thread

Abnormal termination (We'll discuss them later)

- calling `abort`
- Receipt of a signal
- Response of the last thread to a cancellation request

exit(3) Functions

Following functions terminate a program normally

```
#include <stdlib.h>
void exit(int status); // return to the kernel after some cleanup
void _Exit(int status); // return to the kernel immediately

#include <unistd.h>
void _exit(int status); // return to the kernel immediately
```

exit(3) Functions

All three exit functions expect a single integer argument, which we call the exit status. E.g., `exit(0);`

Most UNIX System shells provide a way to examine the exit status of a process.

The exit status of the process is undefined, if

- any of these functions is called without an exit status,
- main does a return without a return value, or
- the main function is not declared to return an integer,

exit(3) function example

The classic “hello, world” example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int my_main()
6  {
7     printf("hello, world\n");
8     sleep(5);
9     return 0;
10 }
11
12 void _start()
13 {
14    int ret = my_main();
15    exit(ret);
16 }
```

```
$ make hello
$ ./hello
hello, world
^C
$ echo $?
130
```

$? prints exit status of previous process

more on exit status http://tldp.org/LDP/abs/html/exitcodes.html

atexit(3) function

With ISO C, a process can register at least 32 functions that are automatically called by `exit`. These are called *exit handlers* and are registered by calling the `atexit` function.

```
#include <stdlib.h>
int atexit(void (*func)(void));
// Returns: 0 if OK, nonzero on error
```

How Process are Created and Terminated

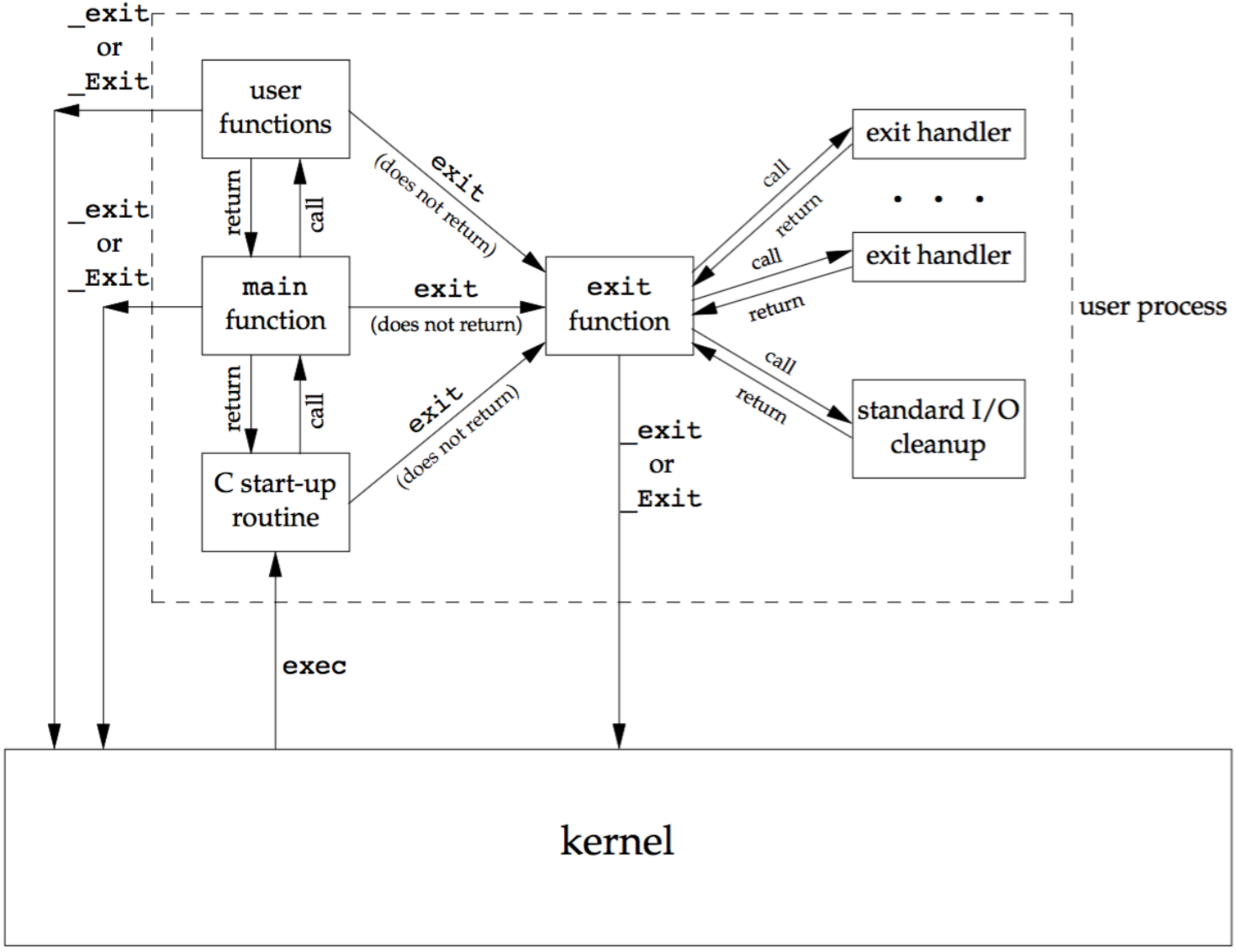

Figure: How a C program is started and how it terminates

Process Environment : Environments

Command-Line Arguments

When a program is executed, the process that does the exec can pass command-line arguments to the new program.

`cd codes; make echoall`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int
5  main(int argc, char *argv[])
6  {
7     int i;
8     for (i = 0; i < argc; i++)
9        /* echo all command-line args */
10       printf("argv[%d]: %s\n", i, argv[i]);
11    exit(0);
12 }
```

try, `./echoall arguments tests`

This you might want to test

- How many arguments can you pass
- how can you insert multiple lines of text as arguments

Environment List

Each program is also passed an environment list

- it is an array of character pointers
- 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 `extern char **environ;`

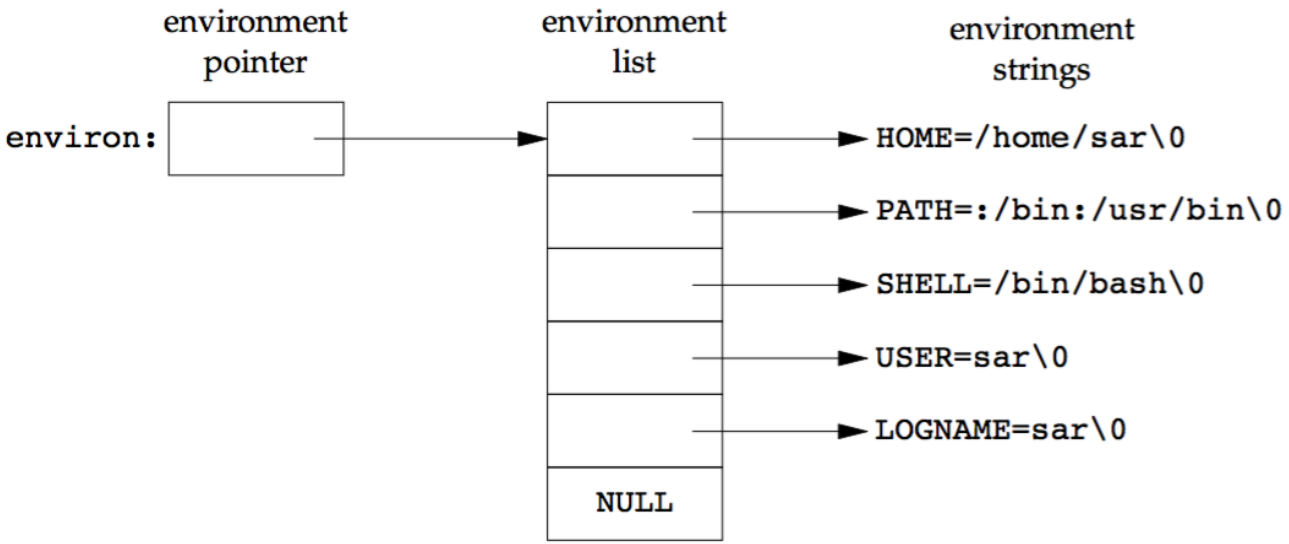

Figure: Environment consisting of five C character strings

Environment Variables

Variable	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
COLUMNS	•	•	•	•	•	terminal width
DATEMSK	XSI		•	•	•	getdate(3) template file pathname
HOME	•	•	•	•	•	home directory
LANG	•	•	•	•	•	name of locale
LC_ALL	•	•	•	•	•	name of locale
LC_COLLATE	•	•	•	•	•	name of locale for collation
LC_CTYPE	•	•	•	•	•	name of locale for character classification
LC_MESSAGES	•	•	•	•	•	name of locale for messages
LC_MONETARY	•	•	•	•	•	name of locale for monetary editing
LC_NUMERIC	•	•	•	•	•	name of locale for numeric editing
LC_TIME	•	•	•	•	•	name of locale for date/time formatting
LINES	•	•	•	•	•	terminal height
LOGNAME	•	•	•	•	•	login name
MSGVERB	XSI	•	•	•	•	fmtmsg(3) message components to process
NLSPATH	•	•	•	•	•	sequence of templates for message catalogs
PATH	•	•	•	•	•	list of path prefixes to search for executable file
PWD	•	•	•	•	•	absolute pathname of current working directory
SHELL	•	•	•	•	•	name of user's preferred shell
TERM	•	•	•	•	•	terminal type
TMPDIR	•	•	•	•	•	pathname of directory for creating temporary files
TZ	•	•	•	•	•	time zone information

Figure: Environment consisting of five C character strings

Environment List cnt'd

ISO C specifies that the main function be written with two arguments

POSIX.1 specifies that environ should be used instead of the (possible) third argument

- Access to specific environment variables is normally through the `getenv` and `putenv` functions
- But to go through the entire environment, the environ pointer must be used

Environment List APIs

```
#include <stdlib.h>
char *getenv(const char *name);
// Returns: pointer to value associated with name, NULL if not found
int putenv(char *str);
// Returns: 0 if OK, nonzero on error
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
// Both return: 0 if OK, 1 on error
```

Fuction	ISO C	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
getenv	•	•	•	•	•	•
putenv		XSI	•	•	•	•
setenv		•	•	•	•	
unsetenv		•	•	•	•	
clearenv				•		

Table: Support for various environment list functions

Process Environment : Memory

The Pieces of a C program

Text segment Machine instructions for CPU. It is sharable. It is **read-only** to prevent from accidental modification

Initialized data segment Or, data segment. Contains variables that are specifically initialized in the program

```
int maxcount = 99;
```

Uninitialized data segment Often called the "bss" segment (A.K.A "block started by symbol")

```
long sum[1000];
```

Stack Automatic variables are stored. Each time a function is called, the address of where to return to and info about the caller's environment (i.e., registers) are saved on the stack.

Heap dynamic memory allocation.

Memory Layout of a C Program

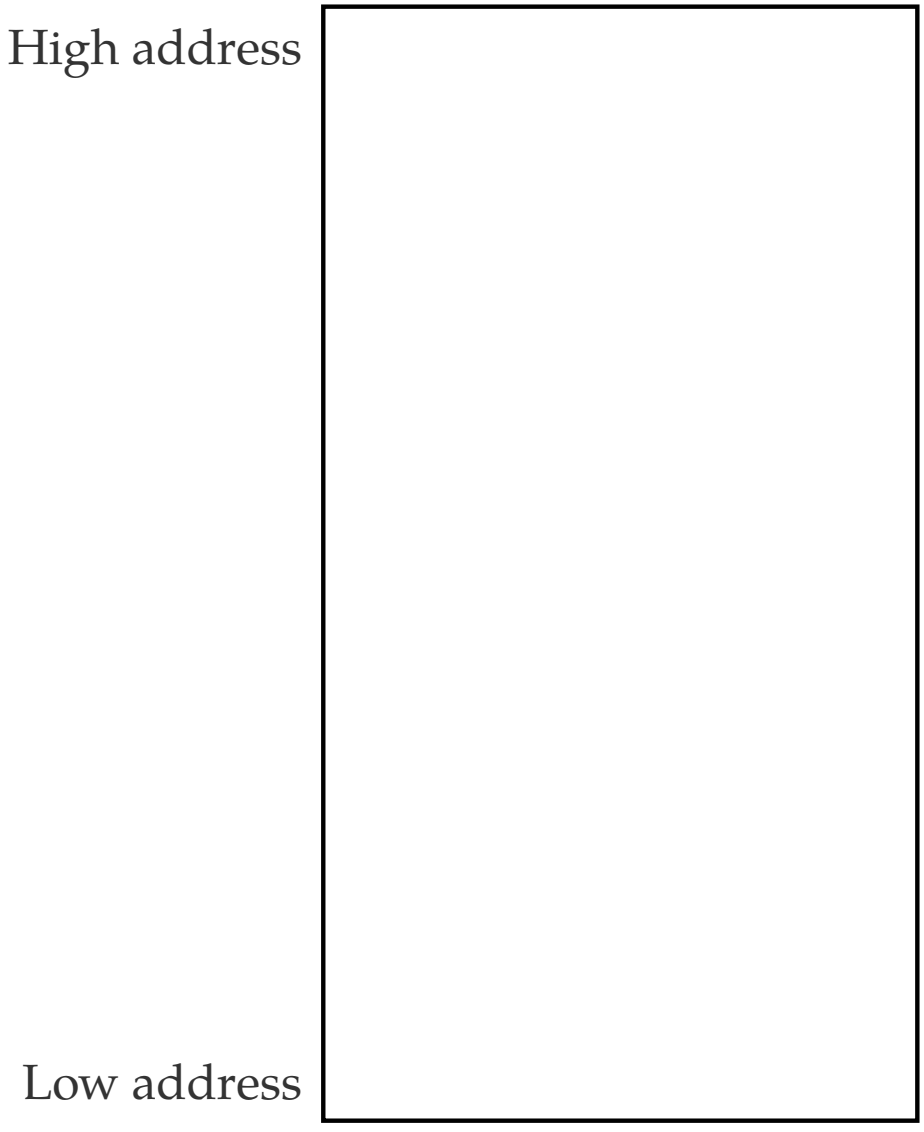

Figure: Typical Memory arrangement

Memory Layout of a C Program

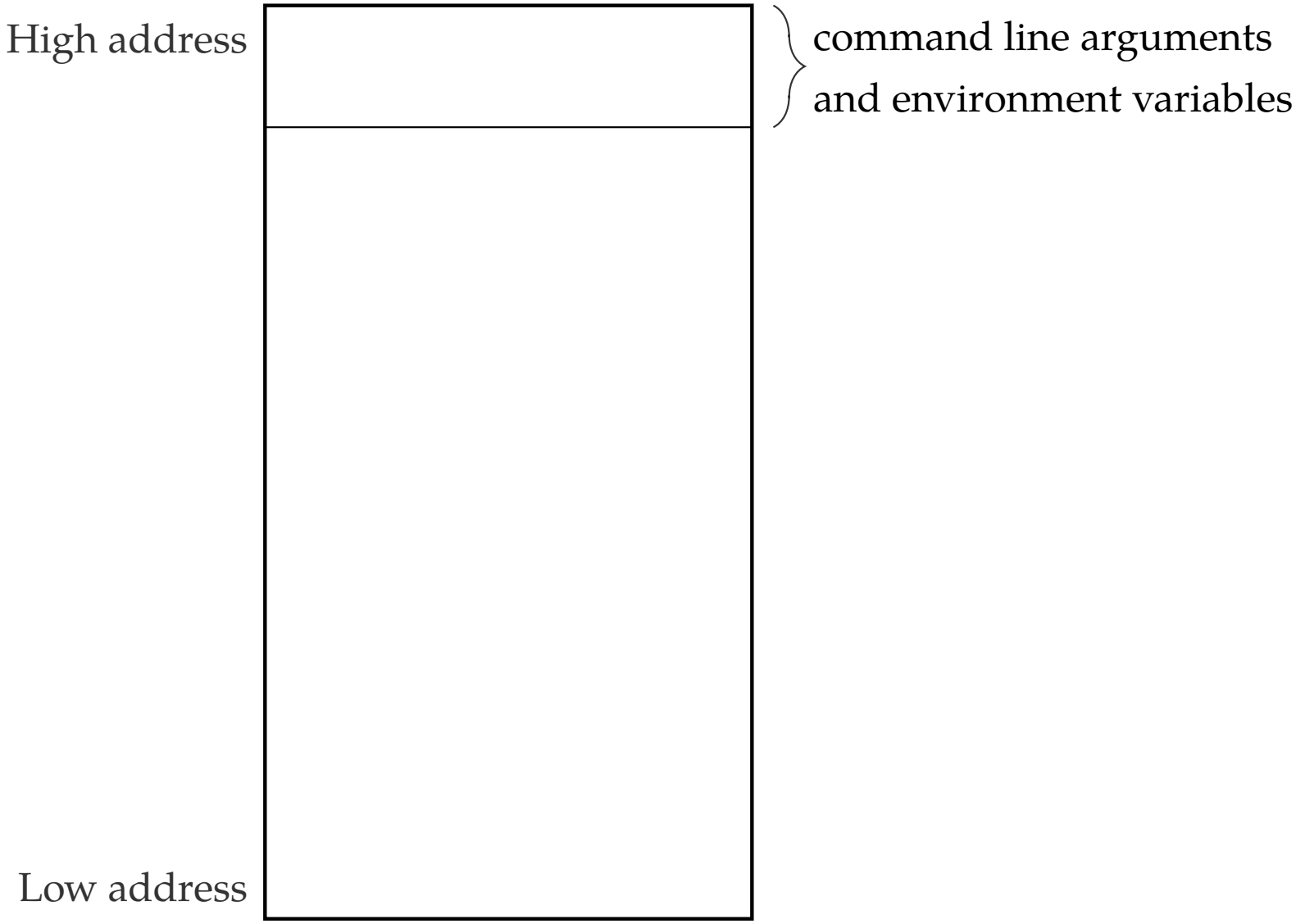

Figure: Typical Memory arrangement

Memory Layout of a C Program

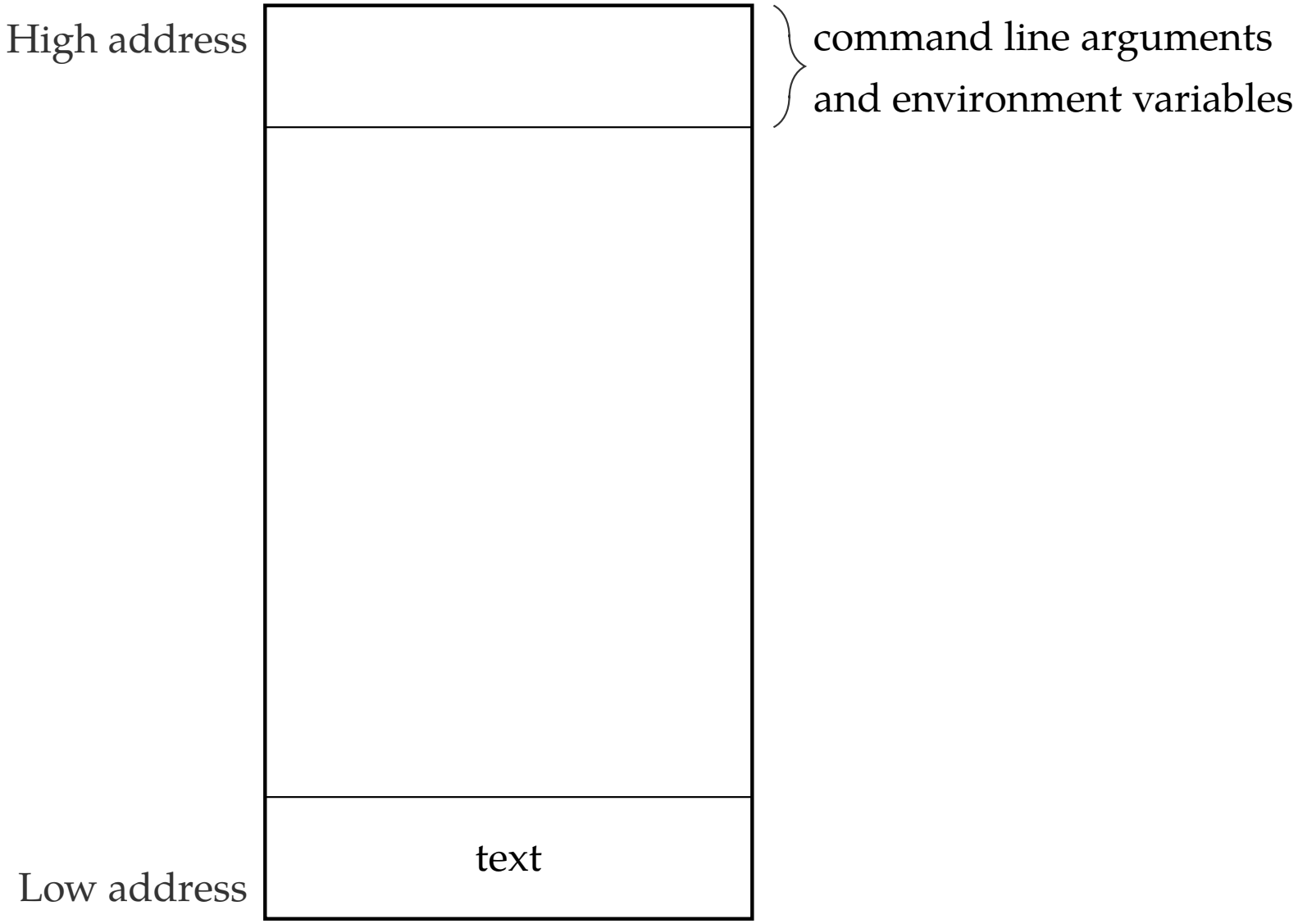

Figure: Typical Memory arrangement

Memory Layout of a C Program

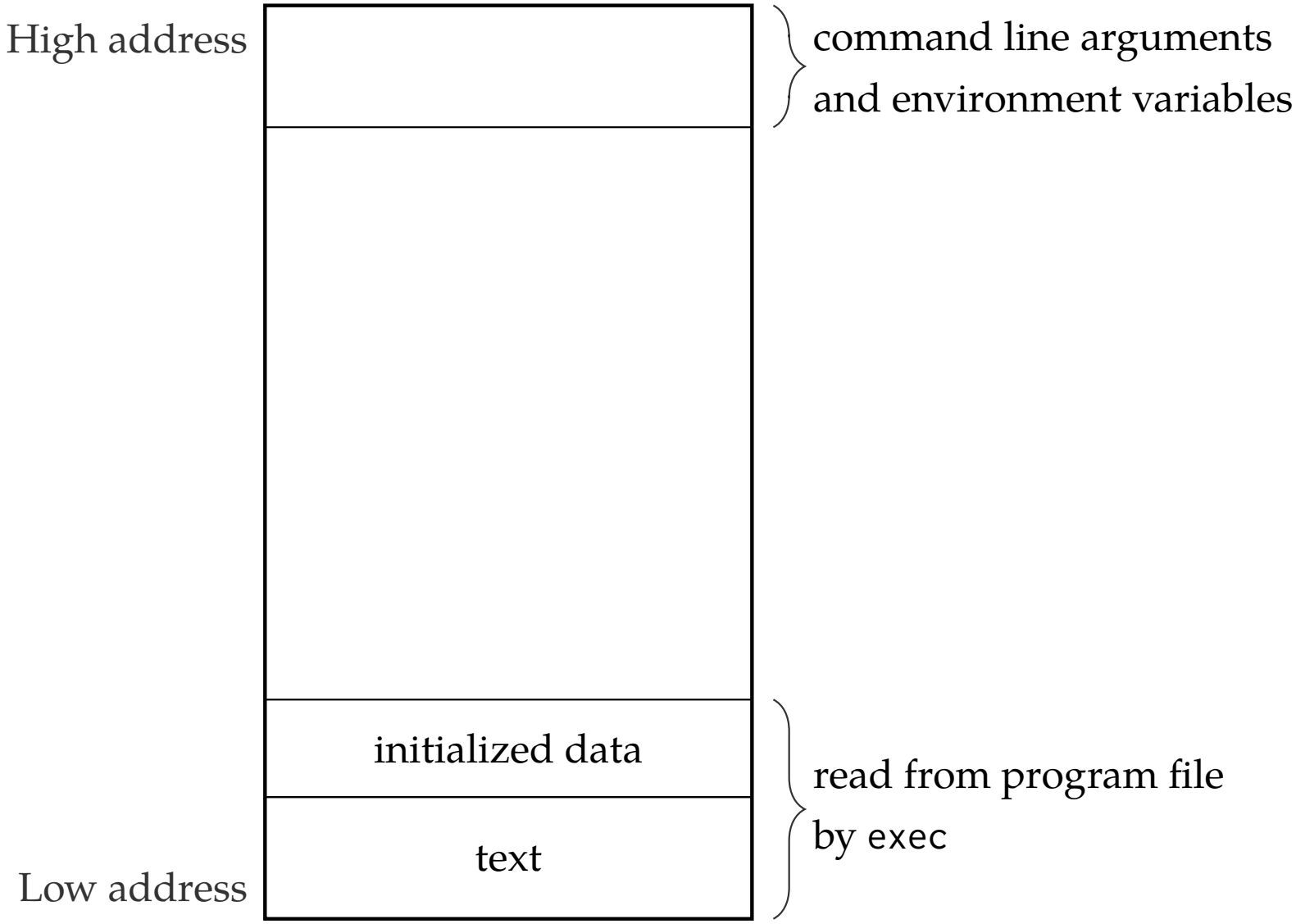

Figure: Typical Memory arrangement

Memory Layout of a C Program

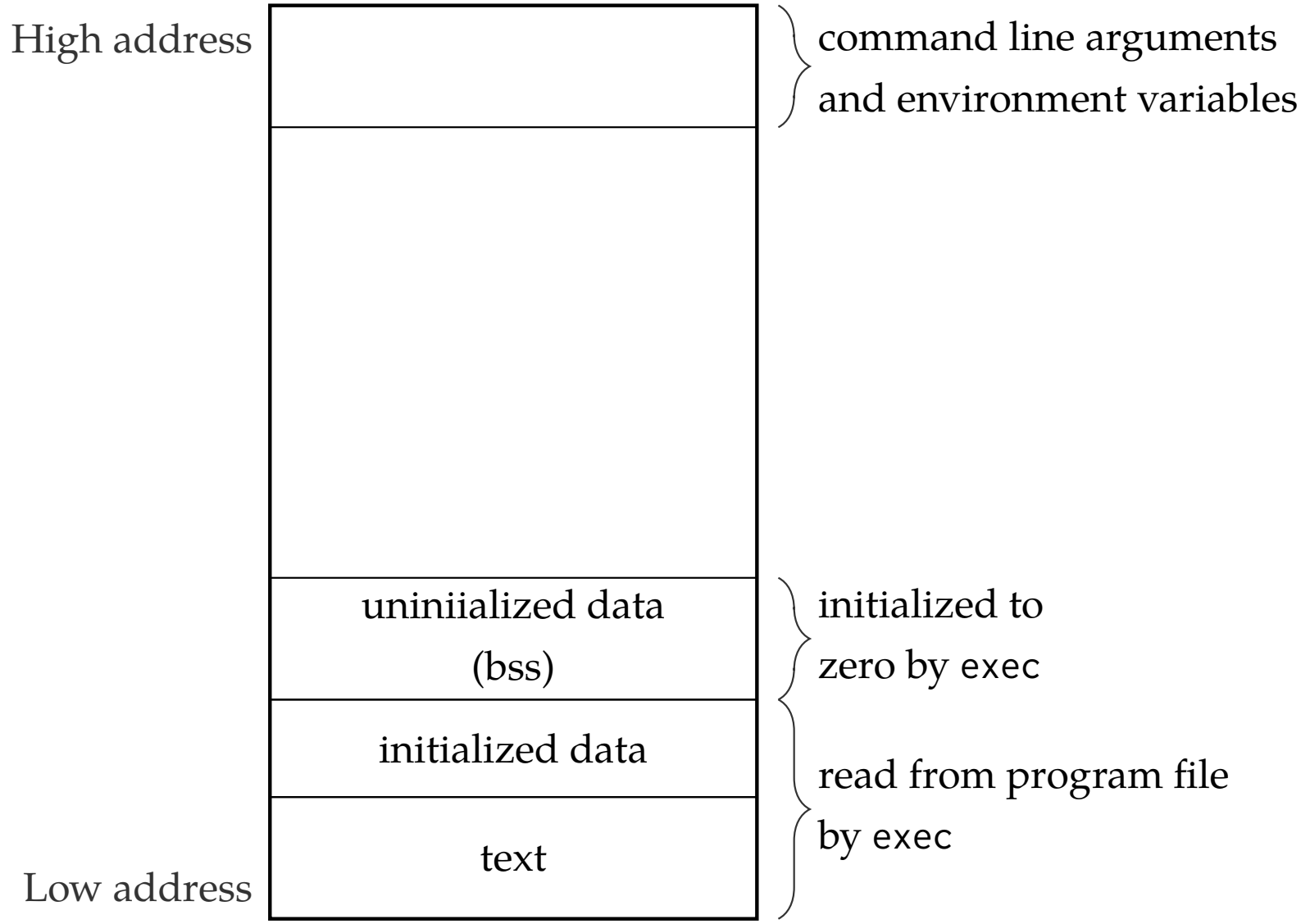

Figure: Typical Memory arrangement

Memory Layout of a C Program

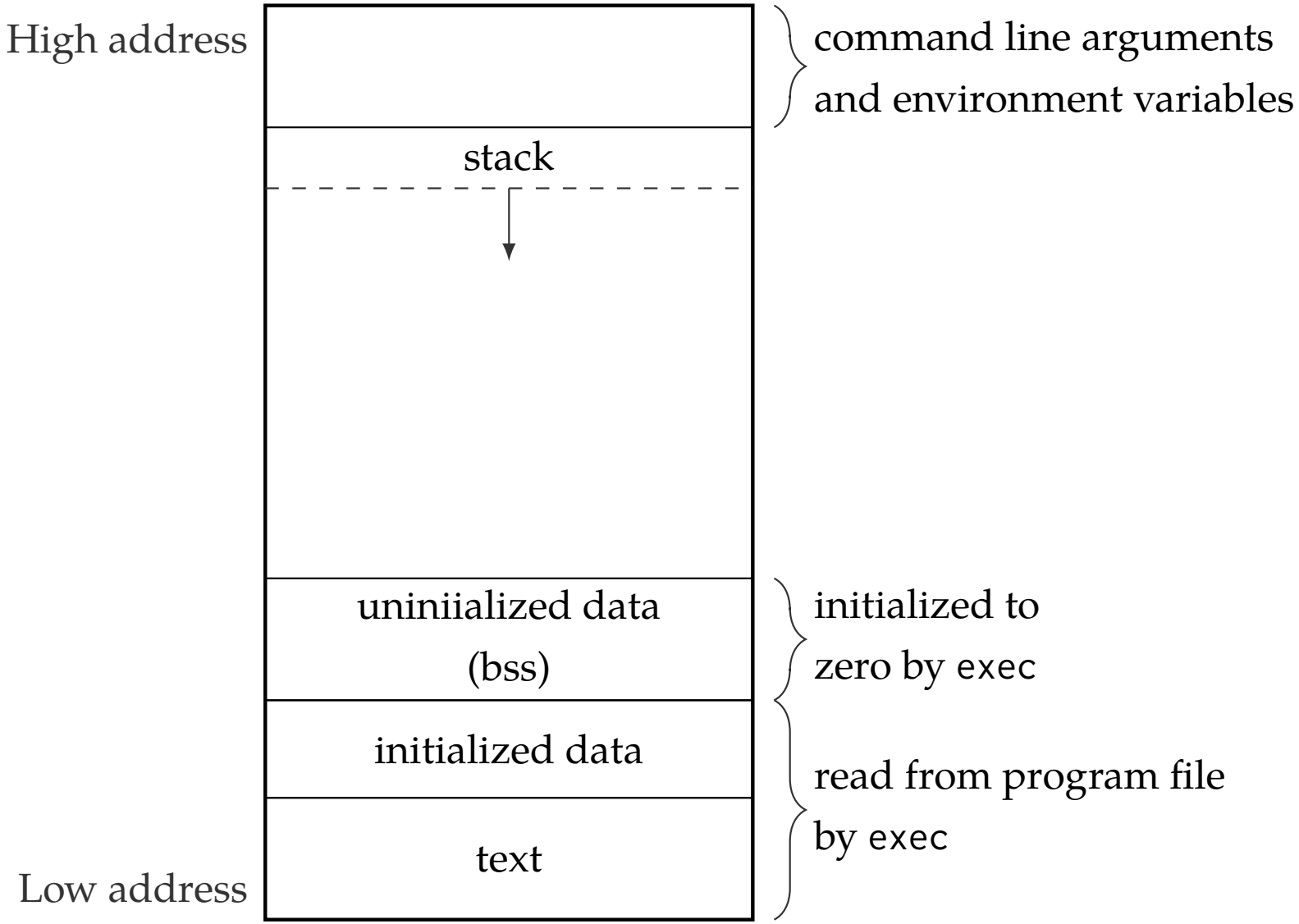

Figure: Typical Memory arrangement

Memory Layout of a C Program

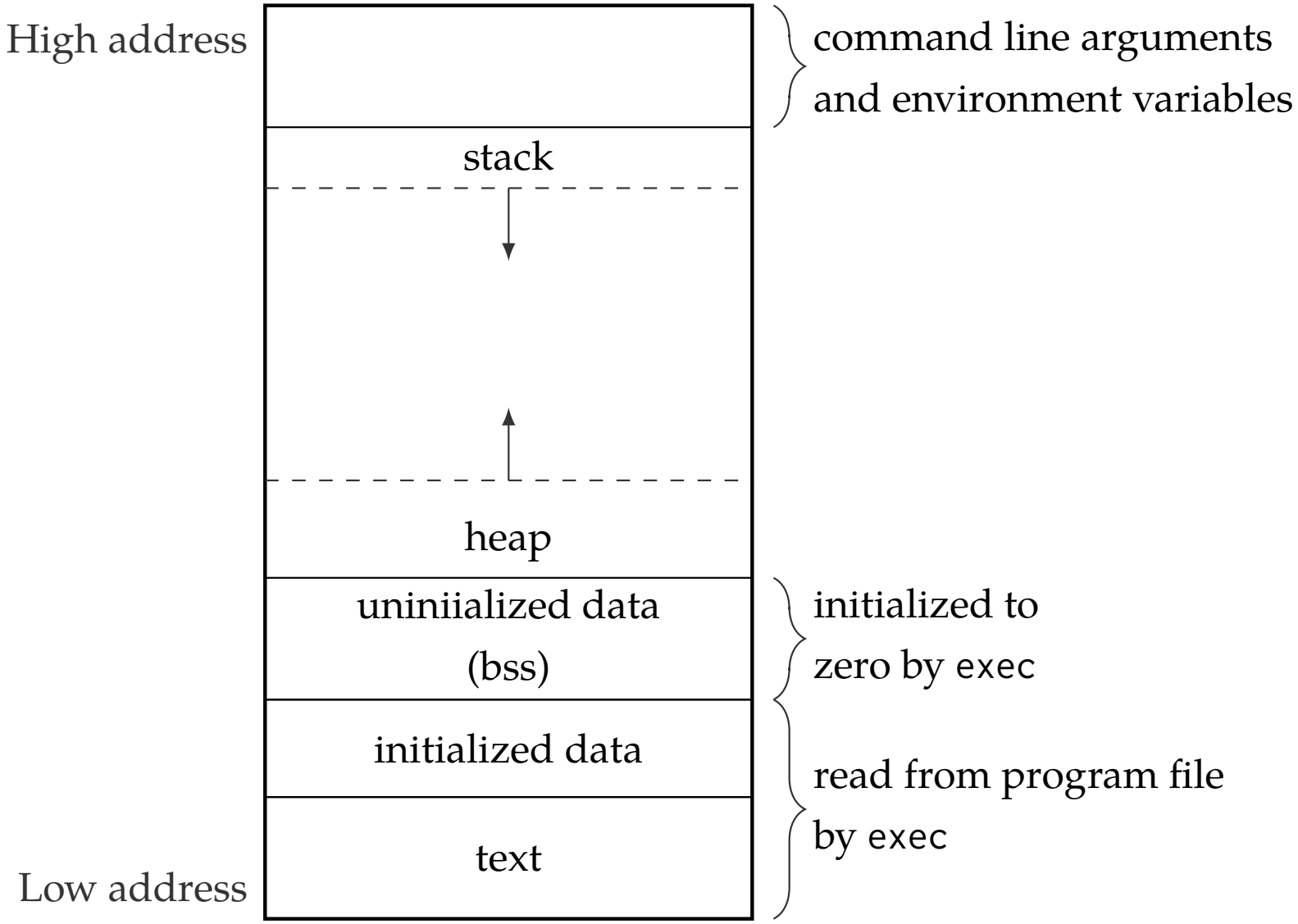

Figure: Typical Memory arrangement

Shared Libraries

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

Mac does not support static linking of a library, but on Linux supports static linking.

Compare the size of a program before and after static linking.

Dynamic linking

```
cc -o layout layout.c
file layout
ldd layout
size layout
objdump -d layout > obj_layout
wc -l obj_layout
```

Static linking

```
cc -static -o layout_st layout.c
file layout_st
ldd layout_st
size layout_st
objdump -d layout_st > obj_layout_st
wc -l obj_layout_st
```

Memory Allocation

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
// All three return: non-null pointer if OK, NULL on error

void free(void *ptr);
```

`malloc` allocates a specified number of bytes of memory. The initial value of the memory is indeterminate

`calloc` allocates a specified number of bytes of memory and initializes them to 0 bits

`realloc` increases or decreases the size of a preivously allocated area

`free` deallocates the area. Do not `free` what is already freed

Process Environment : Misc.

setjmp and longjmp Functions

In C, we can't goto a label that's in another function. Instead, we must use the setjmp and longjmp functions to perform this type of branching

```
 #include <setjmp.h>
int setjmp(jmp_buf env);
// Returns: 0 if called directly, nonzero if returning from a call to longjmp
void longjmp(jmp_buf env, int val);
```

- call setjmp from the location that we want to return to.
  - jmp_buf data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp
- when we encounter an error, we call longjmp with two arguments
  - jmp_buf env used in setjmp
  - val is nonzero value that becomes the return value from setjmp