System Programming

Prof. Chuan-Ju Wang
Dept. of Computer Science
University of Taipei

System Data Files and Information

Introduction

- A UNIX system requires numerous data files for normal operations.
  - The password file **`/etc/passwd`** and the group file **`/etc/group`** are two files that are frequently used by various programs.
    - The password file is used every time a user logs in to a UNIX system.
    - Every time someone executes an **`ls -l`** command.
- Historically, these data files have been ASCII text file.
- But for larger systems, a sequential scan through the password file becomes time consuming.

Password File

- **`/etc/passwd`**

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

- the login name
- encrypted password
- numeric user ID (UID)
- numeric group ID (GID)
- a comment field
- home directory
- shell program

Password File

- **`/etc/passwd`**

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

- There is usually an entry with the user name **`root`**.
  - This entry has a user ID of 0 (the superuser).
- The encrypted password field contains a single character where older versions of the UNIX System used to store the encrypted password.
- Some fields in a password file entry can be empty.
  - If the encrypted password field is empty, it usually means that the user does not have a password.

Password File

- **`/etc/passwd`**

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

  - The shell field contains the name of the executable program to be used as the login shell for the user.
    - The default value for an empty shell field is usually **`/bin/sh`**.
    - **`/dev/null`** as the login shell: This is a device and cannot be executed, so its use here is to prevent anyone from logging in to our system.
      - **`/bin/false`** or **`/bin/true`**

Password File

- **`/etc/passwd`**

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

- The **`nobody`** user name can be used to allow people to log in to a system, but with a user ID (65534) and group ID (65534) that provide no privileges.
  - The only files that this user ID and group ID can access are those that are readable or writable by the world.

Password File

- **`/etc/passwd`**

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

  - **`finger(1)`**: support additional information in the comment field

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] finger
The program 'finger' is currently not installed.  You can install it by typing:
sudo apt-get install finger
```

```
33 jere:x:1000:1000:Chuan-Ju Wang,G323,23113040#8936,:/home/jere:/bin/bash
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] finger -p jere
Login: jere                             Name: Chuan-Ju Wang
Directory: /home/jere                   Shell: /bin/bash
Office: G323, 23113040#8936
On since Sun Feb  5 13:44 (CST) on pts/0 from :0
No mail.
```

Password File

Figure 6.1. Fields in `/etc/passwd` file

Description	struct passwd member	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
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		•			

- Defined in **<pwd.h>**
- Encrypted password field is a one-way hash of the users password. (Always maps to 13 characters from [a-zA-Z0-9./].)

pwd.h = (/usr/include) - VIM

pwd.h

```
49 /* The passwd structure.  */
50 struct passwd
51 {
52   char *pw_name;        /* Username.  */
53   char *pw_passwd;      /* Password.  */
54   __uid_t pw_uid;       /* User ID.  */
55   __gid_t pw_gid;       /* Group ID.  */
56   char *pw_gecos;       /* Real name.  */
57   char *pw_dir;         /* Home directory.  */
58   char *pw_shell;       /* Shell program.  */
59 };
```

Password File

- This two functions allow us to look up an entry given a user's login name or numerical user ID.

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);


        Both return: pointer if OK, NULL on error
```

- The **`getpwuid`**: Used by the **`ls(1)`** program.
  - Map the numerical user ID contained in an i-node into a user's login name.
- The **`getpwnam`**: Used by the **`login(1)`** program when we enter our login name.
- This structure is usually a **`static`** variable within the function, so its contents are overwritten each time we call either of these functions.

Password File

- Some programs need to go through the entire password file.

```
#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`** returns next password entry in file each time it's called, no order
- **`setpwent`** rewinds to "beginning" of entries
- **`endpwent`** closes the file(s)

Password File

Figure 6.2. The `getpwnam` function

```
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd  *ptr;

    setpwent();
    while ((ptr = getpwent()) != NULL)
        if (strcmp(name, ptr->pw_name) == 0)
            break;       /* found a match */
    endpwent();
    return(ptr);    /*a ptr is NULL if no match found */
}
```

Shadow Passwords

- To make it more difficult to obtain the encrypted passwords, systems now store the encrypted password in another file, shadow password file.
  - **`/etc/shadow`**

```
jere@VirtualBox-MBP [/etc] sudo cat shadow | grep jere
jere:$6$6X3IYM0z$Z3qUZIVBcoIGTHA0E/hCPQ6WkATbvGgSGRTKdJZyS9LMHmo6ldzcPfohkaiFEywoSLkwqmMjDNAfDDl72j5.4/:15320:0
jere@VirtualBox-MBP [/etc]
```

Shadow Passwords

- The only two mandatory fields are the user's login name and encrypted password.
- The shadow password file should not be readable by the world.
- Only a few programs need to access encrypted passwords.
  - e.g., **`login(1)`** and **`passwd(1)`**)
  - These programs are often set-user-ID root.

Figure 6.3. Fields in `/etc/shadow` file

Description	`struct spwd` member
user login name	`char *sp_namp`
encrypted password	`char *sp_pwdp`
days since Epoch of last password change	`int sp_lstchg`
days until change allowed	`int sp_min`
days before change required	`int sp_max`
days warning for expiration	`int sp_warn`
days before account inactive	`int sp_inact`
days since Epoch when account expires	`int sp_expire.`
reserved	`unsigned int sp_flag`

Shadow Passwords

- On Linux 2.4.22 and Solaris 9, a separate set of functions is available to access the shadow password file
  - Similar to the set of functions used to access the password file.

```
#include <shadow.h>

struct spwd *getspnam(const char *name);

struct spwd *getspent(void);


        Both return: pointer if OK, NULL on error
void setspent(void);

void endspent(void);
```

Shadow Passwords

- On FreeBSD 5.2.1 and Mac OS X 10.3, there is no shadow password structure.
- The additional account information is stored in the password file **`/etc/passwd`**.

* Nikita S02, E02

Group File

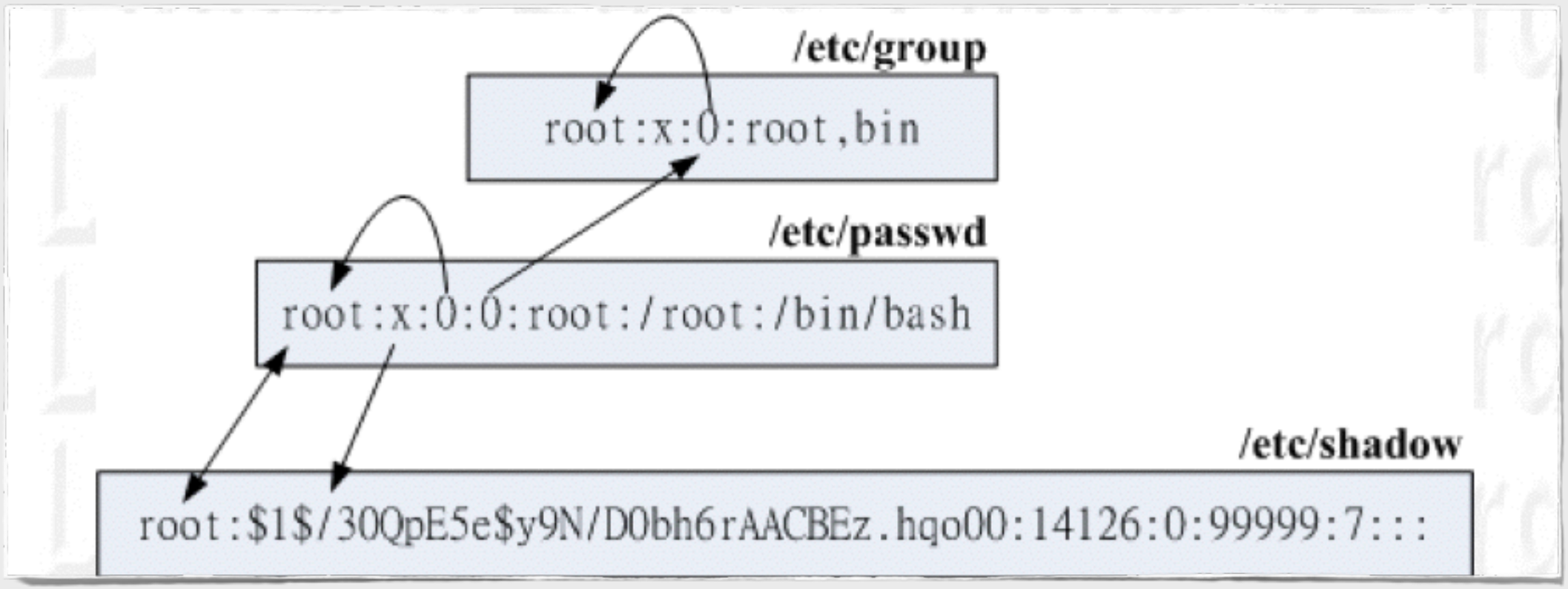
- **`/etc/group`**

```
[root@www ~]# head -n 4 /etc/group
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
```

```
jere@VirtualBox-MBP [/etc] head -4 /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
```

- the group name
- group encrypted password
- numeric group ID (GID)
- the names of the users in this group

Group File


/etc/group
root:x:0:root,bin
/etc/passwd
root:x:0:0:root:/root:/bin/bash
/etc/shadow
root:1/30QpE5e$y9N/D0bh6rAACBEz.hqo00:14126:0:99999:7:::

Group File

- **`/etc/group`**

Figure 6.4. Fields in `/etc/group` file

Description	`struct group` member	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
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`	•	•	•	•	•

Group File

- Look up either a group name or a numerical group ID with the following two functions

```
#include <grp.h>

struct group *getgrgid(gid_t gid);

struct group *getgrnam(const char *name);
```

Both return: pointer if OK, NULL on error

Group File

- Search the entire group file

```
#include <grp.h>

struct group *getgrent(void);

Returns: pointer if OK, NULL on error or end of
                                                file

void setgrent(void);

void endgrent(void);
```

- **`getgrent`** returns next group entry in file each time it's called, no order
- **`setgrent`** rewinds to "beginning" of entries
- **`endgrent`** closes the file(s)

Supplementary Group IDs

- Effective group and initial group

```
[root@www ~]# usermod -G users dmtsai    <==先設定好次要群組
[root@www ~]# grep dmtsai /etc/passwd /etc/group /etc/gshadow
/etc/passwd:dmtsai:x:503:504::/home/dmtsai:/bin/bash
/etc/group:users:x:100:dmtsai    <==次要群組的設定
/etc/group:dmtsai:x:504:          <==因為是初始群組，所以第四欄位不需要填入帳號
```

```
[dmtsai@www ~]$ groups
dmtsai users
```

```
[dmtsai@www ~]$ touch test
[dmtsai@www ~]$ ll
-rw-rw-r-- 1 dmtsai dmtsai 0 Feb 24 17:26 test
```

Effective group

```
[dmtsai@www ~]$ newgrp users
[dmtsai@www ~]$ groups
users dmtsai
[dmtsai@www ~]$ touch test2
[dmtsai@www ~]$ ll
-rw-rw-r-- 1 dmtsai dmtsai 0 Feb 24 17:26 test
-rw-r--r-- 1 dmtsai users  0 Feb 24 17:33 test2
```

Supplementary Group IDs

- With Version 7, each user belonged to a single group at any point in time.
- With 4.2BSD, the concept of supplementary group IDs was introduced.
  - Not only did we belong to the group corresponding to the group ID in our password file entry, but we also could belong to up to 16 additional groups.
  - The file access permission checks were modified so that not only was the effective group ID compared to the file's group ID, but also all the supplementary group IDs were compared to the file's group ID.

Supplementary Group IDs

- Three functions are provided to fetch and set the supplementary group IDs.

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

Returns: number of supplementary group IDs if OK, 1 on error

[View full width]

```
#include <grp.h>      /* on Linux */
#include <unistd.h>   /* on FreeBSD, Mac OS X, and
➥  Solaris */

int setgroups(int ngroups, const gid_t grouplist[]);

#include <grp.h>      /* on Linux and Solaris */
#include <unistd.h>   /* on FreeBSD and Mac OS X */

int initgroups(const char *username, gid_t basegid);
```

Both return: 0 if OK, 1 on error

Implementation Differences

Figure 6.5. Account implementation differences

Information	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Account information	/etc/passwd	/etc/passwd	netinfo	/etc/passwd
Encrypted passwords	/etc/master.passwd	/etc/shadow	netinfo	/etc/shadow
Hashed password files?	yes	no	no	no
Group information	/etc/group	/etc/group	netinfo	/etc/group

System Identification

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

```
jere@VirtualBox-MBP [/etc] uname -a
Linux jere-VirtualBox 3.0.0-12-generic #20-Ubuntu SMP Fri Oct 7 14:50:42 UTC 2011 i686 i686 i386 GNU/Linux
```

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

Returns: non-negative value if OK, 1 on error

```
struct utsname {
  char  sysname[];    /* name of the operating system */
  char  nodename[];   /* name of this node */
  char  release[];    /* current release of operating system */
  char  version[];    /* current version of this release */
  char  machine[];    /* name of hardware type */
};
```

Other Data Files

Figure 6.6. Similar routines for accessing system data files

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	gethostbyname, gethostbyaddr
networks	/etc/networks	<netdb.h>	netent	getnetbyname, getnetbyaddr
protocols	/etc/protocols	<netdb.h>	protoent	getprotobyname, getprotobynumber
services	/etc/services	<netdb.h>	servent	getservbyname, getservbyport

Time and Date

- The **`time`** function returns the current time and date.

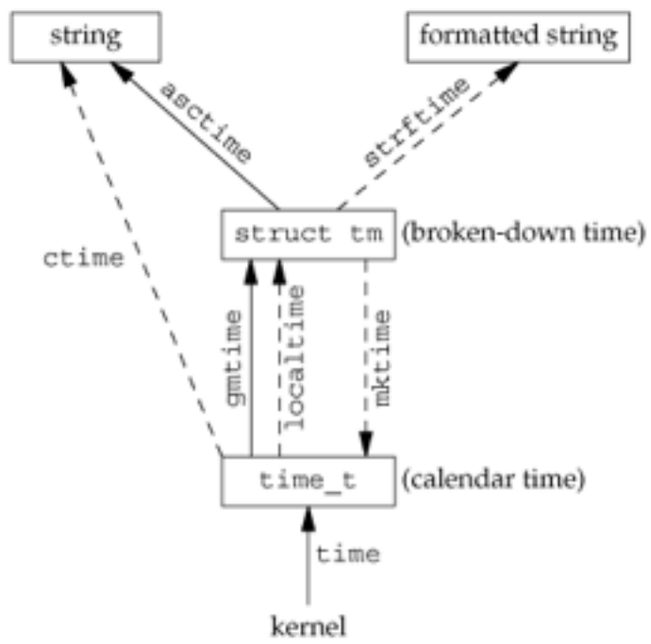
```
#include <time.h>

time_t time(time_t *calptr);

                    Returns: value of time if OK, 1 on error
```

  - Time is kept in UTC.
  - Time conversions (timezone, daylight savings time) handled "automatically."
  - Time and date kept in a single quantity (**`time_t`**).
    - Counts the number of seconds that have passed since the Epoch: 00:00:00 January 1, 1970, Coordinated Universal Time (UTC).

Time and Date

- Once we have the integer value that counts the number of seconds since the Epoch, we normally call one of the other time functions to convert it to a human-readable time and date.

Figure 6.8. Relationship of the various time functions



Affected by the environment variable **`TZ`**:

`localtime`
`mktime`
`ctime`
`strftime`

Time and Date

- The two functions **`localtime`** and **`gmtime`** convert a calendar time into a **`tm`** structure.

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);

struct tm *localtime(const time_t *calptr);
```

Both return: pointer to broken-down time

```
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 */
};
```

Time and Date

- The function **`mktime`** takes a broken-down time, expressed as a local time, and converts it into a **`time_t`** value.

```
#include <time.h>

time_t mktime(struct tm *tmptr);


      Returns: calendar time if OK, 1 on error
```

Time and Date

- The **`asctime`** and **`ctime`** functions produce the familiar 26-byte string that is similar to the default output of the **`date(1)`** command.

```
jere@VirtualBox-MBP [/etc] date
Tue Feb 28 23:59:41 CST 2012
jere@VirtualBox-MBP [/etc]
```

```
#include <time.h>

char *asctime(const struct tm *tmptr);

char *ctime(const time_t *calptr);


 Both return: pointer to null-terminated string
```

Time and Date

- The time function, **`strftime`**, is the most complicated.
- It is a printf-like function for time values.

```
#include <time.h>

size_t strftime(char *restrict buf, size_t maxsize,
                const char *restrict format,
                const struct tm *restrict tmptr);
```

Returns: number of characters stored in array if room, 0 otherwise

Assignment

- Reading (do it at home):
  - Read and try: http://linux.vbird.org/linux_basic/0410accountmanager.php
  - Manual pages for the functions covered
  - Stevens Chap. 6

Assignment

- Assignment 6

  1. Coding:

     - Given some user names, implement a C program to show show the users' information and time.

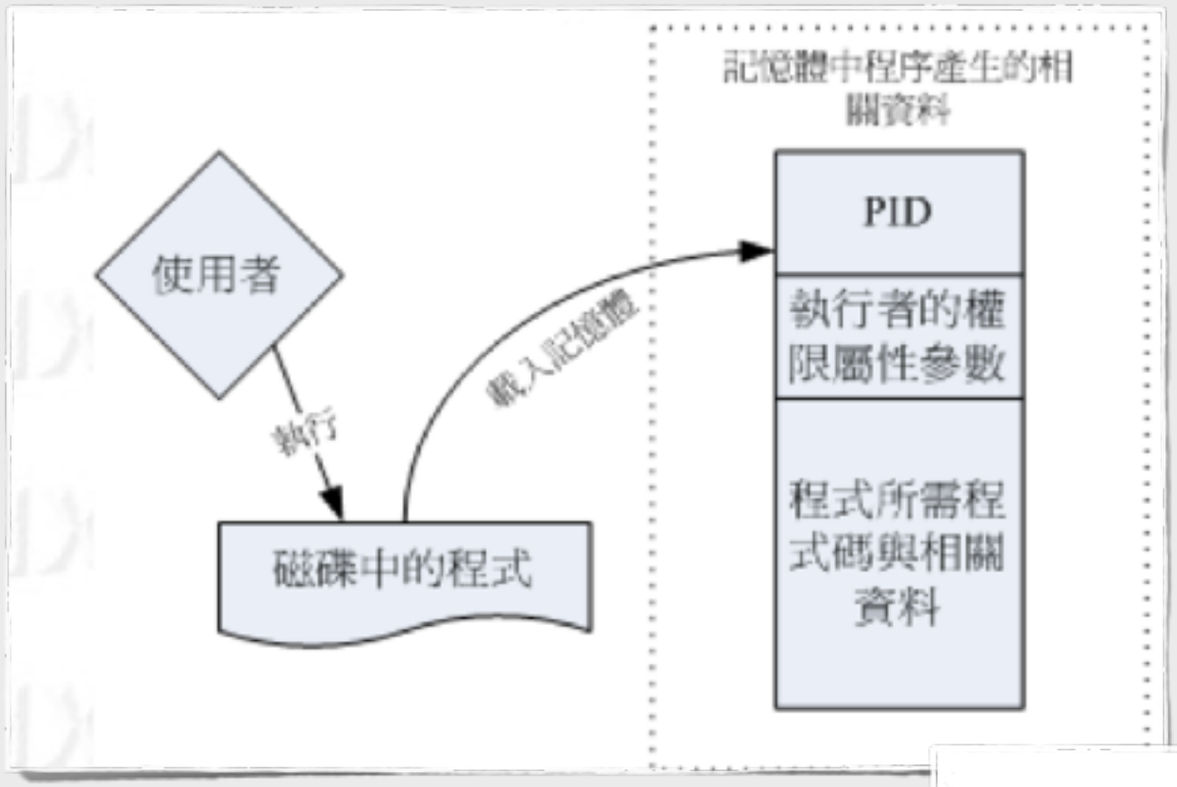
     ➡ A program with a brief report should be handed in. (submit via lms).

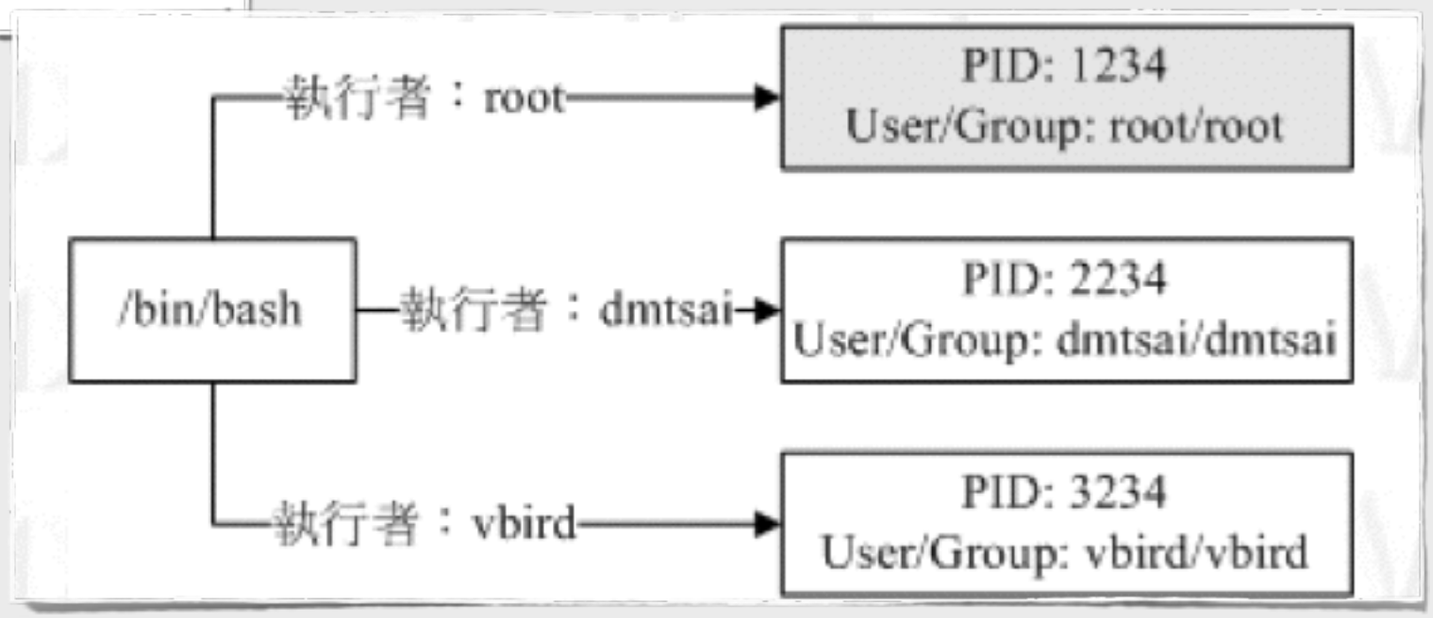
Process Environment

Introduction

- This chapter examines the environment of a single process.
- We'll see
  - how the **`main`** function is 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
  - various ways for the process to terminate
- We'll look at the **`longjmp`** and **`setjmp`** functions and their interaction with the stack.
- We'll examine the resource limits of a process.

Program and Process





* http://linux.vbird.org/linux_basic/0440processcontrol.php

Suspend, Resume, Kill

- control-C (`^C`)
  - kill process
- control-Z (`^Z`)
  - suspend process
  - If you are stuck in vim or some other interactive program that won't let you get out of it, a quick `^Z` will cause that program to go into suspended animation and let you back out to the prompt again.

Suspend, Resume, Kill

- **`kill`** *processID*
  - how to find out the processID of suspend programs
- **`ps`** or **`jobs -l`**
  - ps command shows all your processes
  - jobs command shows the suspend and background processes
- **`fg`**
  - resume (in foreground) the most recently suspend process
- **`bg`**
  - resume (in background) the most recently suspend process

Background Process

- Background process
  - use **`&`** to run a program in the background
  - use **`fg`** to bring the process to the foreground
  - **`^Z`** to suspend the process

Background Process

```
[root@www ~]# vi ~/.bashrc
# 在 vi 的一般模式下，按下 [ctrl]-z 這兩個按鍵
[1]+  Stopped                 vim ~/.bashrc
[root@www ~]#    <==順利取得了前景的操控權！
[root@www ~]# find / -print
....(輸出省略)....
# 此時螢幕會非常的忙碌！因為螢幕上會顯示所有的檔名。請按下 [ctrl]-z 暫停
[2]+  Stopped                 find / -print
```

```
[root@www ~]# jobs [-lrs]
選項與參數：
-l ：除了列出 job number 與指令串之外，同時列出 PID 的號碼；
-r ：僅列出正在背景 run 的工作；
-s ：僅列出正在背景當中暫停 (stop) 的工作。

範例一：觀察目前的 bash 當中，所有的工作，與對應的 PID
[root@www ~]# jobs -l
[1]- 10314 Stopped                 vim ~/.bashrc
[2]+ 10833 Stopped                 find / -print
```

Background Process

```
[root@www ~]# fg %jobnumber
選項與參數：
%jobnumber ：jobnumber 為工作號碼(數字)。注意，那個 % 是可有可無的！

範例一：先以 jobs 觀察工作，再將工作取出：
[root@www ~]# jobs
[1]- 10314 Stopped                    vim ~/.bashrc
[2]+ 10833 Stopped                    find / -print
[root@www ~]# fg     <==預設取出那個 + 的工作，亦即 [2]。立即按下[ctrl]-z
[root@www ~]# fg %1    <==直接規定取出的那個工作號碼！再按下[ctrl]-z
[root@www ~]# jobs
[1]+  Stopped                 vim ~/.bashrc
[2]-  Stopped                 find / -print
```

Background Process

```
範例一：一執行 find / -perm +7000 > /tmp/text.txt 後，立刻丟到背景去暫停！
[root@www ~]# find / -perm +7000 > /tmp/text.txt
# 此時，請立刻按下 [ctrl]-z 暫停！
[3]+  Stopped                 find / -perm +7000 > /tmp/text.txt

範例二：讓該工作在背景下進行，並且觀察他！！
[root@www ~]# jobs ; bg %3 ; jobs
[1]-  Stopped                 vim ~/.bashrc
[2]   Stopped                 find / -print
[3]+  Stopped                 find / -perm +7000 > /tmp/text.txt
[3]+ find / -perm +7000 > /tmp/text.txt &   <==用 bg%3 的情況！
[1]+  Stopped                 vim ~/.bashrc
[2]   Stopped                 find / -print
[3]-  Running                 find / -perm +7000 > /tmp/text.txt &
```

Kill(1)

```
[root@www ~]# kill -signal %jobnumber
[root@www ~]# kill -l
選項與參數：
-l ：這個是 L 的小寫，列出目前 kill 能夠使用的訊號 (signal) 有哪些？
signal ：代表給予後面接的那個工作什麼樣的指示囉！用 man 7 signal 可知：
 -1 ：重新讀取一次參數的設定檔 (類似 reload)；
 -2 ：代表與由鍵盤輸入 [ctrl]-c 同樣的動作；
 -9 ：立刻強制刪除一個工作；
 -15：以正常的程序方式終止一項工作。與 -9 是不一樣的。

範例一：找出目前的 bash 環境下的背景工作，並將該工作『強制刪除』。
[root@www ~]# jobs
[1]+  Stopped                 vim ~/.bashrc
[2]   Stopped                 find / -print
[root@www ~]# kill -9 %2; jobs
[1]+  Stopped                 vim ~/.bashrc
[2]   Killed                  find / -print
# 再過幾秒你再下達 jobs 一次，就會發現 2 號工作不見了！因為被移除了！

範例：找出目前的 bash 環境下的背景工作，並將該工作『正常終止』掉。
[root@www ~]# jobs
[1]+  Stopped                 vim ~/.bashrc
[root@www ~]# kill -SIGTERM %1
# -SIGTERM 與 -15 是一樣的！您可以使用 kill -l 來查閱！
```

`main` Function

- A C program starts execution with a function called **`main`**.

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

  - C program started by kernel (by one of the **`exec`** functions)
  - Special startup routine called by kernel which sets up things for **`main`** (or whatever entrypoint is defined)
  - **`argc`** is a count of the number of command line arguments (including the command itself)
  - **`argv`** is an array of pointers to the arguments
  - It is guaranteed by both ANSI C and POSIX.1 that `argv[argc]` ==`NULL`

Process Termination

- There are 8 ways for a process to terminate.
- Normal termination:
  - return from `main`
  - calling `exit`
  - calling `_exit`(or `_Exit`)
  - return of last thread from its start routine (Section 11.5)
  - calling `pthread_exit` (Section 11.5) from last thread

Process Termination

- Abnormal termination:
  - calling `abort` (Section 10.17)
  - terminated by a signal (Section 10.2)
  - response of the last thread to a cancellation request (Sections 11.5 and 12.7)

Process Termination

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

- **`_exit(2)`** and **`_Exit(2)`**
  - return to the kernel immediately
  - `_exit` required by POSIX.1
  - `_Exit` required by ISO C99
  - synonymous on Unix
- **`exit(3)`** does some cleanup and then returns
- All three functions take integer argument, called *`exit status`*.

`atexit(3)` Function

- With ISO C, a process can register up to 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
```

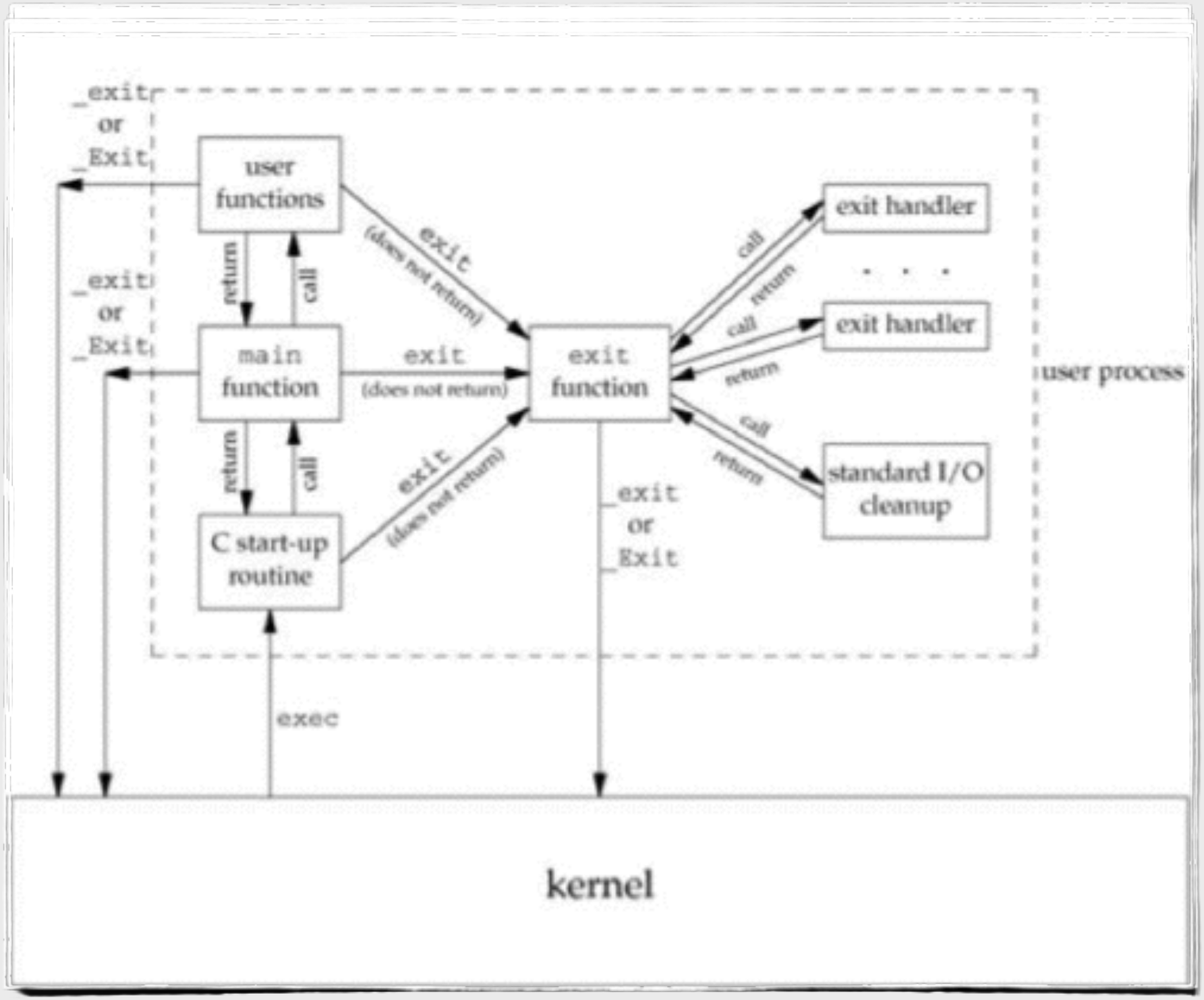
- Functions invoked in reverse order of registration.
- Same function can be registered more than once.
- Extremely useful for cleaning up open files, freeing certain resources, etc.

Figure 7.3. Example of exit handlers

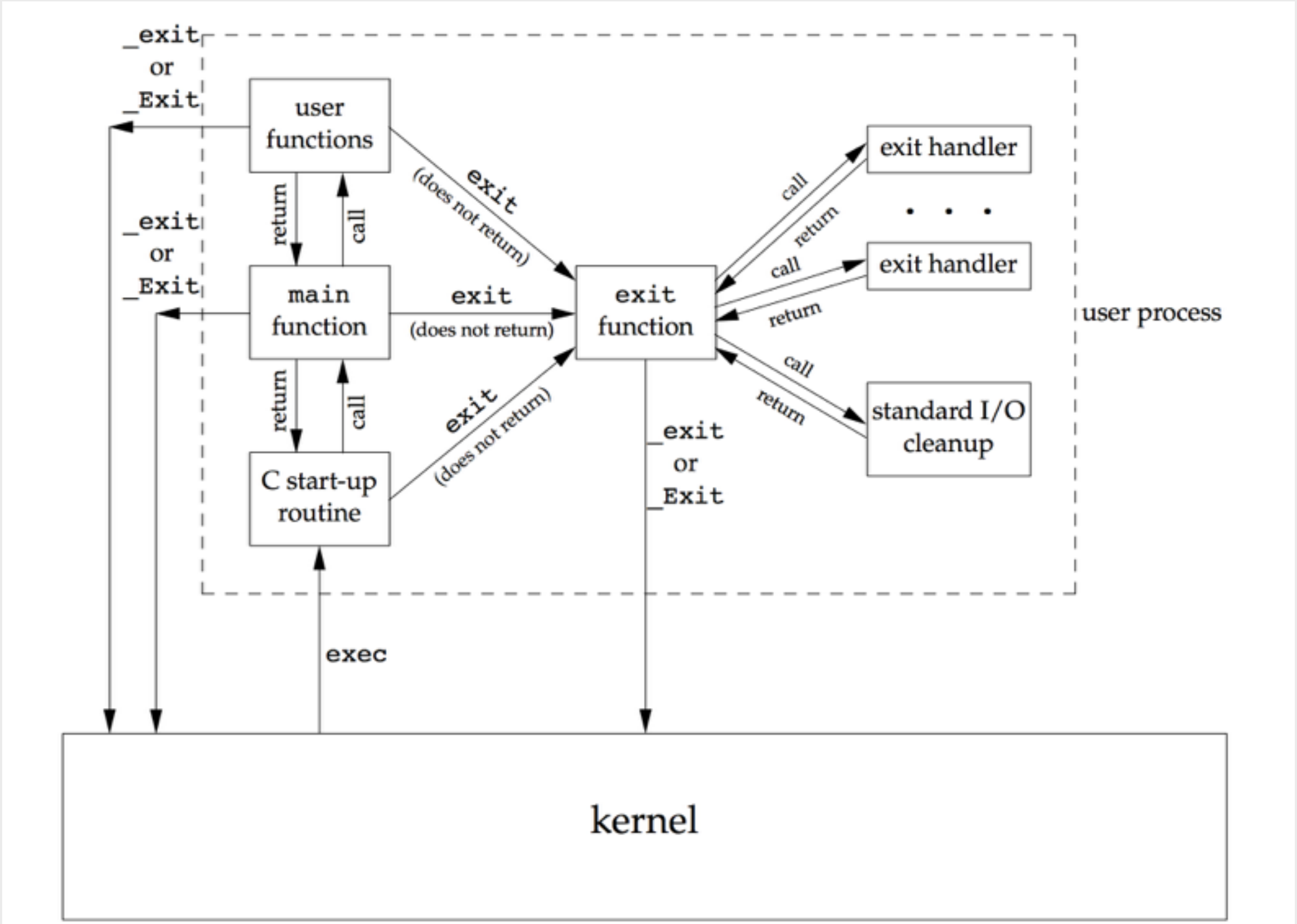
```c
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
     if (atexit(my_exit2) != 0)
         err_sys("can't register my_exit2");

     if (atexit(my_exit1) != 0)
         err_sys("can't register my_exit1");

     if (atexit(my_exit1) != 0)
         err_sys("can't register my_exit1");

     printf("main is done\n");
     return(0);
}

static void
my_exit1(void)
{
   printf("first exit handler\n");
}

static void
my_exit2(void)
{
   printf("second exit handler\n");
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] cp ../fig7.3 fig7-3.c
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig7-3
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GNU_SO
-lapue  -o fig7-3
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig7-3
main is done
first exit handler
first exit handler
second exit handler
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test]
```

Lifetime of a UNIX Process



Lifetime of a UNIX Process



Command-Line Arguments

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

Figure 7.4. Echo all command-line arguments to standard output

```
#include "apue.h"

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

    for (i = 0; i < argc;  i++)     /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] cp ../fig7.4 fig7-4.c
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig7-4
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GNU_SOURC
-lapue  -o fig7-4
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig7-4
argv[0]: ./fig7-4
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig7-4 arg1 TEST foo
argv[0]: ./fig7-4
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test]
```

Environment List

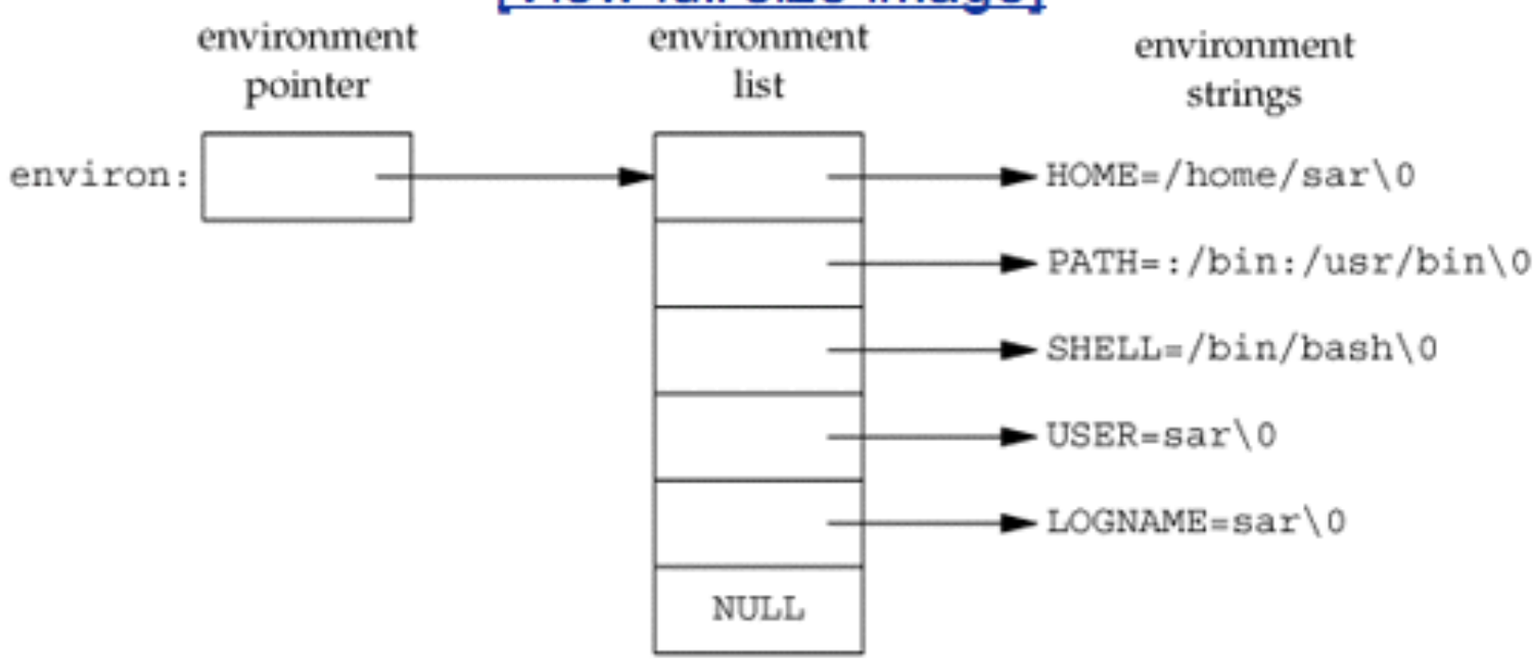
- Each program is also passed an environment list.
  - An array of character pointers
  - The address of the array of pointers is contained in the global variable **`environ`**:

```
extern char **environ;
```

Environment List

Figure 7.5. Environment consisting of five C character strings

[View full size image]



Memory Layout of a C Program

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

1. Text segment, the machine instructions that the CPU executes.
   - Often sharable and read-only
2. Initialized data segment, containing variables that are specifically initialized in the program.

```
int   maxcount = 99;
```

3. Uninitialized data segment ("bss" segment)

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

Memory Layout of a C Program

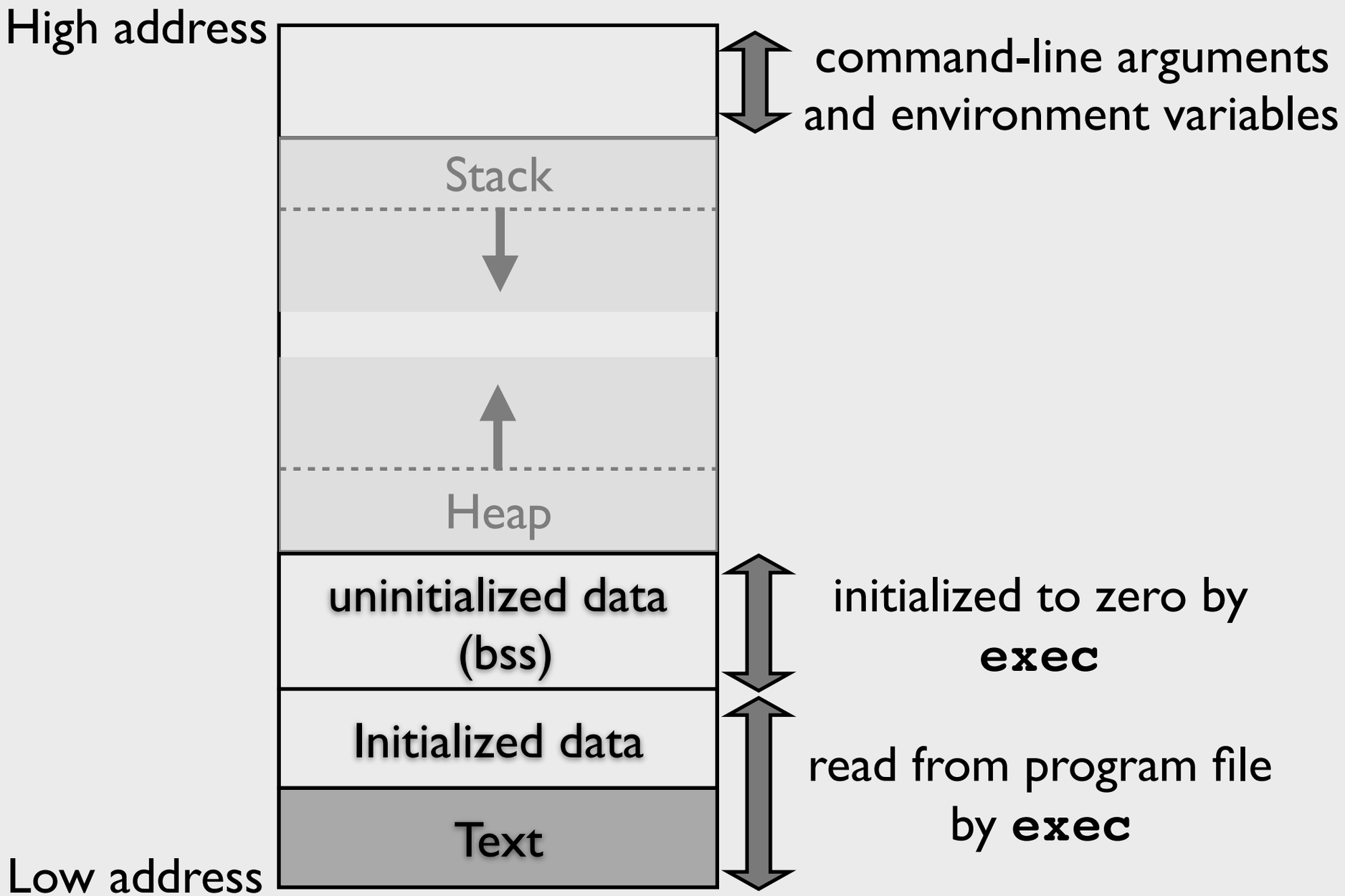
- Historically, a C program has been composed of the following pieces:

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

5. Heap, where dynamic memory allocation usually takes place.

    - Historically, the heap has been located between the uninitialized data and the stack.

Memory Layout of a C Program



Memory Layout of a C Program

Ubuntu 11.10 (Linux 3.0.0-12-generic)

```
jere@VirtualBox-MBP [~] gcc HelloWorld.c
jere@VirtualBox-MBP [~] ./a.out
Hello World
jere@VirtualBox-MBP [~] file ./a.out
./a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically l
inked (uses shared libs), for GNU/Linux 2.6.15, not stripped
jere@VirtualBox-MBP [~] size ./a.out
   text    data     bss     dec     hex filename
   1131     256       8    1395     573 ./a.out
jere@VirtualBox-MBP [~]
```

Mac OS X 10.6.8 (Darwin 10.8.0)

```
Jere@MBP [~] gcc HelloWorld.c
Jere@MBP [~] ./a.out
Hello World
Jere@MBP [~] file ./a.out
./a.out: Mach-O 64-bit executable x86_64
Jere@MBP [~] size ./a.out
__TEXT   __DATA   __OBJC  others  dec     hex
4096     4096     0       4294971392      4294979584      100003000
Jere@MBP [~]
```

Shared Libraries

- Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference.
- This reduces the size of each executable file but may add some runtime overhead,
  - either when the program is first executed or the first time each shared library function is called.
- Library functions can be replaced with new versions without having to relink edit every program that uses the library.

Shared Libraries

```
jere@VirtualBox-MBP [~] gcc -static HelloWorld.c
jere@VirtualBox-MBP [~] ls -l a.out
-rwxrwxr-x 1 jere jere 644428 2012-03-03 23:21 a.out
jere@VirtualBox-MBP [~] size a.out
   text    data     bss     dec     hex filename
 575523    2056    7048  584627    8ebb3 a.out
```

```
jere@VirtualBox-MBP [~] gcc HelloWorld.c
jere@VirtualBox-MBP [~] ls -l a.out
-rwxrwxr-x 1 jere jere 7164 2012-03-03 23:22 a.out
jere@VirtualBox-MBP [~] size a.out
   text    data     bss     dec     hex filename
   1131     256       8    1395     573 a.out
```

* Creating a shared and static library: http://www.adp-gmbh.ch/cpp/gcc/create_lib.html

Memory Allocation

- ISO C specifies three functions for memory allocation.
  - The allocation routines are usually implemented with the **`sbrk(2)`** system call.

```
#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`** – initial value is indeterminate.
- **`calloc`** – initial value set to all zeros.
- **`realloc`** – changes size of previously allocated area. Initial value of any additional space is indeterminate.
- **`free`** - causes the space pointed to by *ptr* to be deallocated.

* http://openhome.cc/Gossip/CGossip/MallocFree.html

Environment Variables

- Access to specific environment variables is normally through the **`getenv`** and **`putenv`** functions.

```
#include <stdlib.h>

char *getenv(const char *name);
```

Returns: pointer to *value* associated with *name*, `NULL` if not found

Environment Variables

- Sometimes we may want to set an environment variable.
- We may want to change the value of an existing variable or add a new variable to the environment.
  - In the next chapter, we'll see that we can affect the environment of only the current process and any child processes that we invoke. We cannot affect the environment of the parent process, which is often a shell.

Environment Variables

Figure 7.8. Support for various environment list functions

Function	ISO C	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
getenv	•	•	•	•	•	•
putenv		XSI	•	•	•	•
setenv		•	•	•	•	
unsetenv		•	•	•	•	
clearenv				•		

```
#include <stdlib.h>

int putenv(char *str);

int setenv(const char *name, const char *value,
 int rewrite);

int unsetenv(const char *name);
```

All return: 0 if OK, nonzero on error

Environment Variables

- **`putenv`**: takes a string of the form `name=value` and places it in the environment list.
  - If name already exists, its old definition is first removed.
- **`setenv`**: sets name to value.
  - If name already exists in the environment, then
    - (a) if *rewrite* is nonzero, the existing definition for name is first removed;
    - (b) if *rewrite* is 0, an existing definition for name is not removed, name is not set to the new value, and no error occurs.
- **`unsetenv`**: removes any definition of name.

Environment Variables

- How these functions must operate when modifying the environment list?

1. If we're modifying an existing name:
    - If the size of the new value is less than or equal to the size of the existing value, we can just copy the new string over the old string.
    - If the size of the new value is larger than the old one, however, we must 1) **`malloc`** to obtain room for the new string, 2) copy the new string to this area, and 3) then replace the old pointer in the environment list for name with the pointer to this allocated area.

Environment Variables

2. If we're adding a new name, call **`malloc`** to allocate room for the `name=value` string and copy the string to this area.

   - The first time we've added a new `name`:

     (1) call `malloc` to obtain room for a new list of pointers (on heap).

     (2) copy the old environment list to this new area and store a pointer to the `name=value` string at the end of this list of pointers.

     (3) Finally, set **`environ`** to point to this new list of pointers.

Environment Variables

2. If we're adding a new name, call **`malloc`** to allocate room for the `name=value` string and copy the string to this area.

   - Not the first time we've added a new name:

     (1) The list is already on the heap, so we just call **`realloc`** to allocate room for one more pointer.

     (2) The pointer to the new `name=value` string is stored at the end of the list.

`setjmp` and `longjmp` Functions

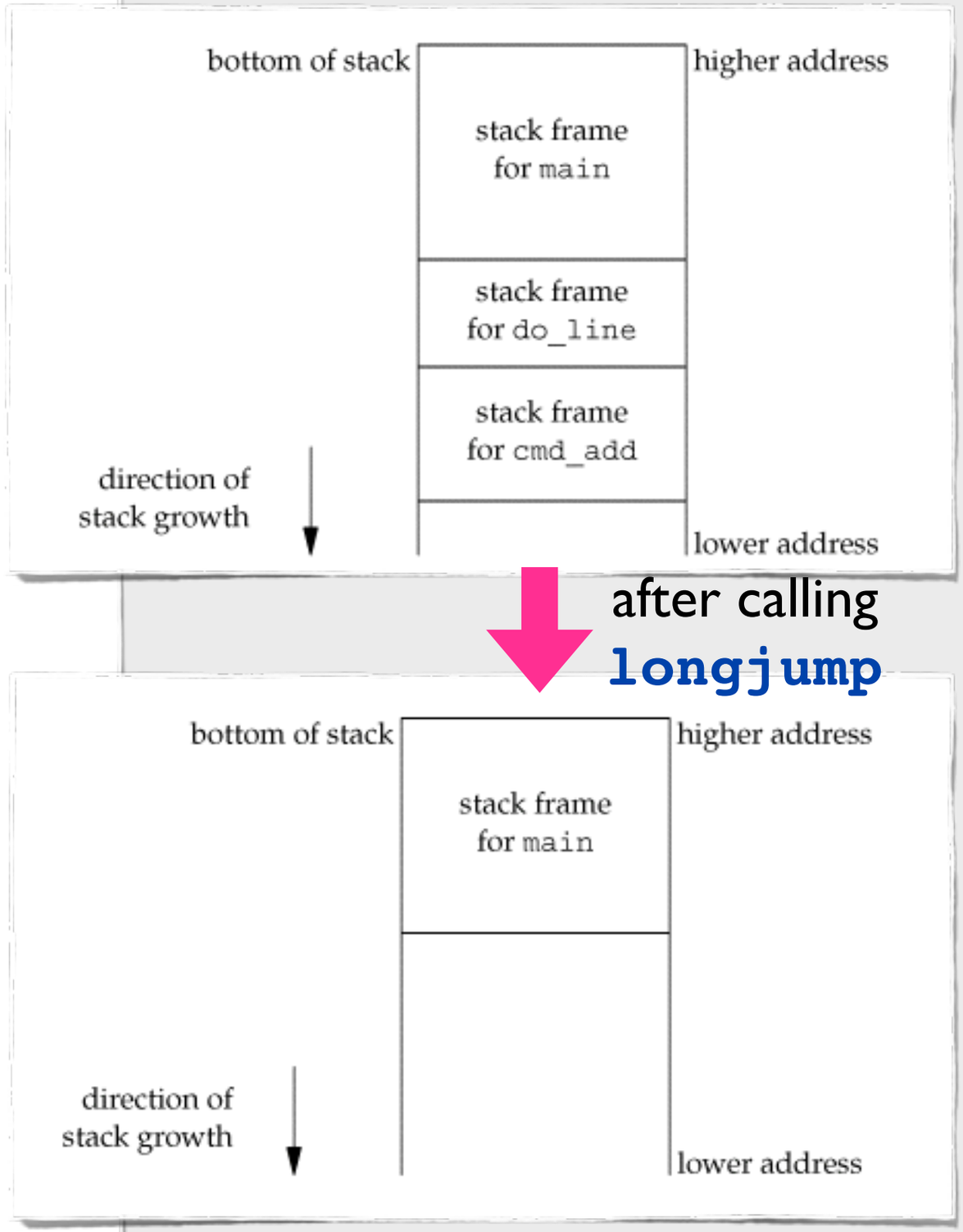
- In C, we can't goto a label that's in another function.
- Use the **`setjmp`** and **`longjmp`** functions to perform this type of branching

```
#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);
```

- Things you can (but not necessarily should) do using these functions:
  - use as "non-local goto"
  - handle error conditions in deeply nested functions
  - jump out of a signal handler back into your program
  - fake multi-threading
- Remember: `longjmp` may not be called after the routine which called `setjmp`.

Figure 7.11. Example of `setjmp` and `longjmp`

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
     char    line[MAXLINE];

     if (setjmp(jmpbuffer) != 0)
          printf("error");
     while (fgets(line, MAXLINE, stdin) != NULL)
          do_line(line);
     exit(0);
}

 ...

void
cmd_add(void)
{
     int     token;

     token = get_token();
     if (token < 0)     /* an error has occurred */
          longjmp(jmpbuffer, 1);
     /* rest of processing for this command */
}
```



Automatic, Register, and Volatile Variables

- What are the states of the automatic variables and register variables in the `main` function?
  - Unfortunately, the answer is "it depends."
  - If you have an automatic variable that you don't want rolled back, define it with the **`volatile`** attribute.
  - Variables that are declared **`global`** or **`static`** are left alone when `longjmp` is executed.

Figure 7.13. Effect of `longjmp` on various types of variables

```
#include "apue.h"
#include <setjmp.h>

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf jmpbuffer;
static int     globval;

int
main(void)
{
    int              autoval;
    register int     regival;
    volatile int     volaval;
    static int       statval;

    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }

    /*
     * Change variables after setjmp, but before longjmp.
     */
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* never returns
    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] gcc fig7-13.c
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] gcc -O fig7-13.c
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

```
static void
f1(int i, int j, int k, int l)
{
    printf("in f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
        " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}
static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

`getrlimit` and `setrlimit` Functions

- Every process has a set of resource limits, some of which can be queried and changed by the `getrlimit` and `setrlimit` functions

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit
➥ *rlptr);
```

Both return: 0 if OK, nonzero on error

```
struct rlimit {
  rlim_t  rlim_cur;   /* soft limit: current limit */
  rlim_t  rlim_max;   /* hard limit: maximum value for rlim_cur */
};
```

`getrlimit` and `setrlimit` Functions

- Changing resource limits follows these rules:
  - A soft limit can be changed by any process to a value less than or equal to its hard limit.
  - Any process can lower its hard limit greater than or equal to its soft limit.
  - Only superuser can raise hard limits.
  - Changes are per process only.