Chapter 7.
Process Environment

System Programming

http://www.cs.ccu.edu.tw/~pahsiung/courses/sp

熊博安

國立中正大學資訊工程學系

pahsiung@cs.ccu.edu.tw

(05)2720411 ext. 33119

Class: EA-104

Office: EA-512

Textbook: Advanced Programming in the UNIX Environment, 2nd Edition, W. Richard Stevens and Stephen A. Rago

Introduction

- How is main() called?
- How are arguments passed?
- Memory layout?
- Memory allocation?
- Environment variables
- Process termination

main Function

- int main(int argc, char *argv[]);
- arc = #arguments
- argv[] = arguments
- Kernel executes a special START-UP routine before main()
- Start-up routine sets things up before main() is called: stack, heap, etc.

Process Termination

8 ways

- Normal termination:
  - return from main()
  - calling exit()
  - calling _exit() or _Exit()

  Section 11.5

  - Return from the last thread from its start routine
  - Calling pthread_exit from the last thread
- Abnormal termination

  Section 10.17

  - calling abort()

  Section 10.2

  - Receipt of a signal

  Sections 11.5, 12.7

  - Response of the last thread to a cancellation request

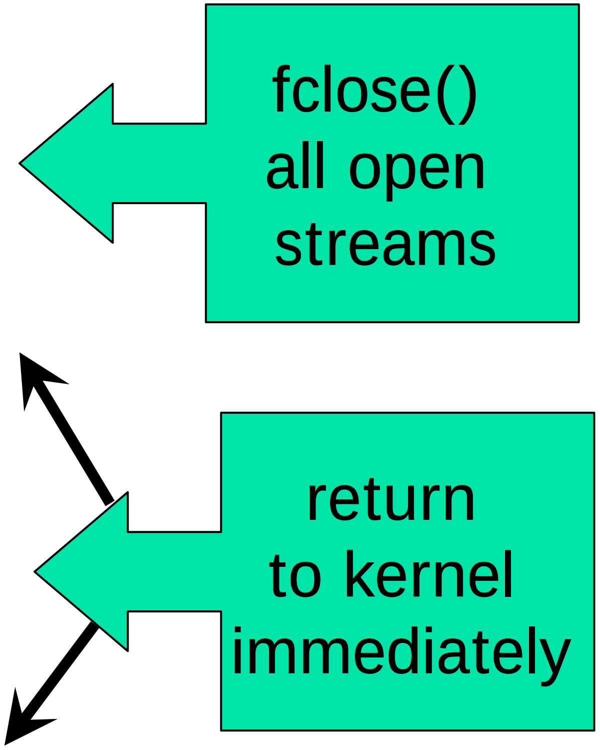
exit(), _Exit, _exit()

ISO C

- #include <stdlib.h>
- void exit(int *status*);
- void _Exit(int status);

POSIX.1

- #include <unistd.h>
- void _exit(int *status*);



fclose() all open streams

return to kernel immediately

Exit Status

- All exit functions require a single integer as exit status of the process
- Exit status is sometimes undefined if
  - Exit function called without an exit status
  - main returns without a return value
  - main function is not declared to return an integer

Exit Status Example

```
#include <stdio.h>
main() { printf(“hello, world\n”); }
```

- $ **cc hello.c**
- $ **./a.out**
- hello world
- $ **echo $?**
- 13 ← depends on stack / register contents when returning

Exit Status Example

```
$ cc –std=c99 hello.c
hello.c:4: warning: return type defaults
to ‘int’
$ ./a.out
hello, world
$ echo $?
0
```

atexit(): Exit Handler

- #include <stdlib.h>
- int atexit(void (* *func*) (void));
- Returns: 0 if OK, nonzero on error
- *func* is an exit handler, at most 32
- exit() calls these exit handler functions in the reverse order of registration
- #times called = #times registered

atexit(): Exit Handler

- ISO C and POSIX.1
  - exit first calls the exit handlers
  - then closes (via fclose) all open streams
- POSIX.1 extends ISO C standard
  - any installed exit handlers will be cleared on exec (a new process does not have the original exit handlers)

Program Start & Termination

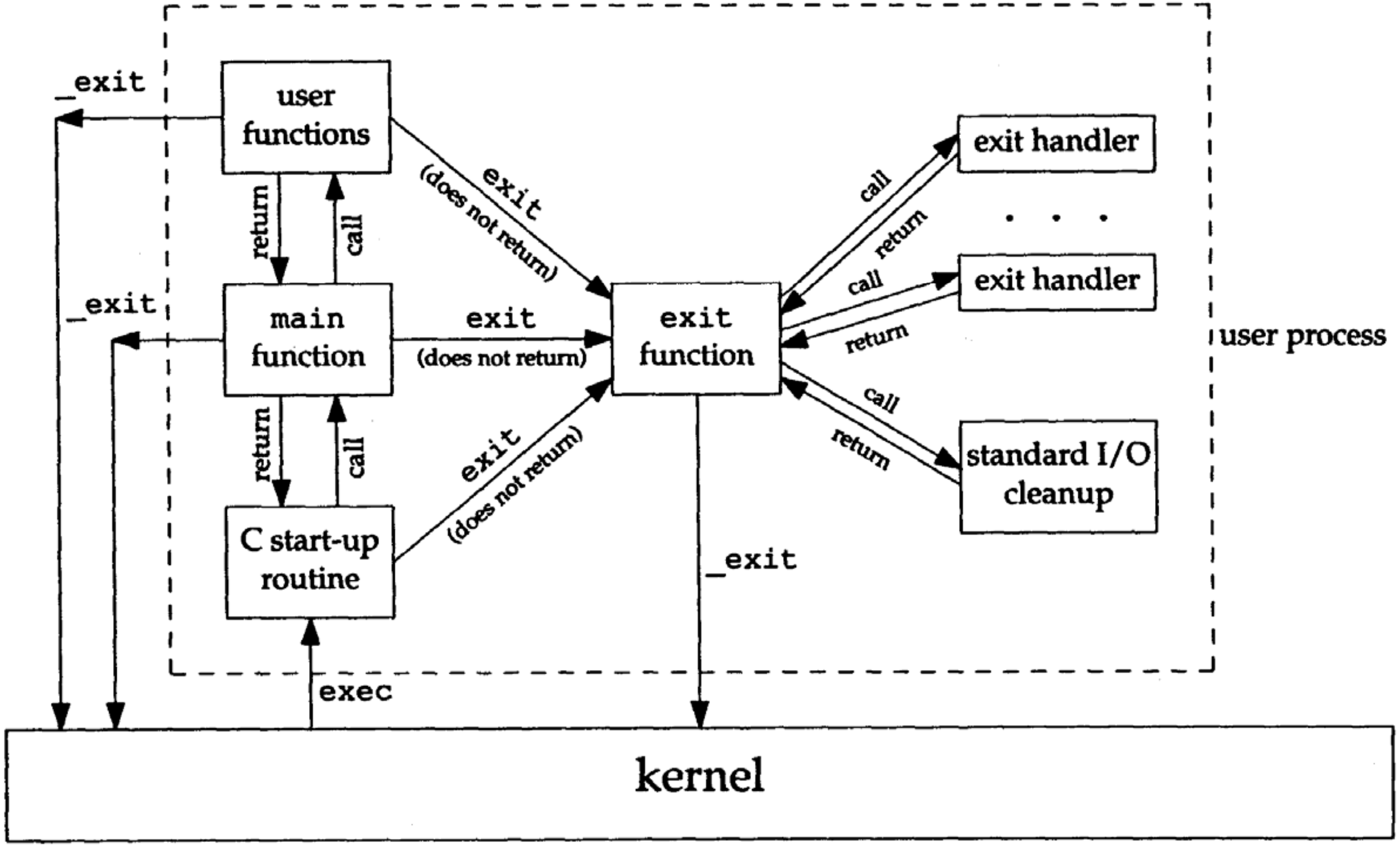


Figure 7.3: Exit Handlers

```
#include     “apue.h"
static void  my_exit1(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"); }
```

Figure 7.3: results

```
$ a.out
main is done
first exit handler
first exit handler
second exit handler
```

Command-Line Arguments

- exec() can pass command-line arguments to a new program
- Part of normal operation of Unix shells
- echo() does not echo 0th argument
- argv[argc] is NULL (ISO C, POSIX.1)
  - `for(i=0; argv[i] != NULL; i++) …`

Figure 7.4: echo()

```
#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);
}
```

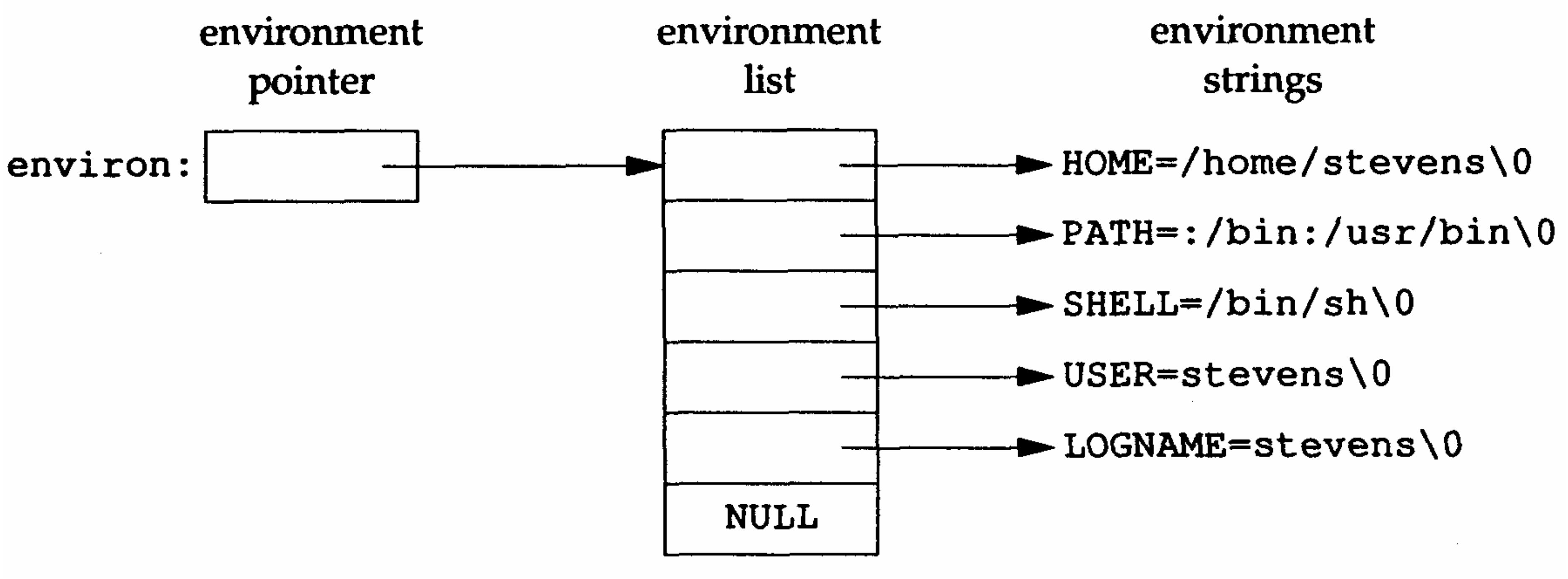
Figure 7.4: results

- $ ./echoarg arg1 TEST foo
- argv[0]: ./echoarg
- argv[1]: arg1
- argv[2]: TEST
- argv[3]: foo

Environment List

- An array of character pointers (addresses of null-terminated C strings)
- Array address is in global variable environ
  - extern char **environ;
- getenv(): get an environment string
- putenv(): set an environment string

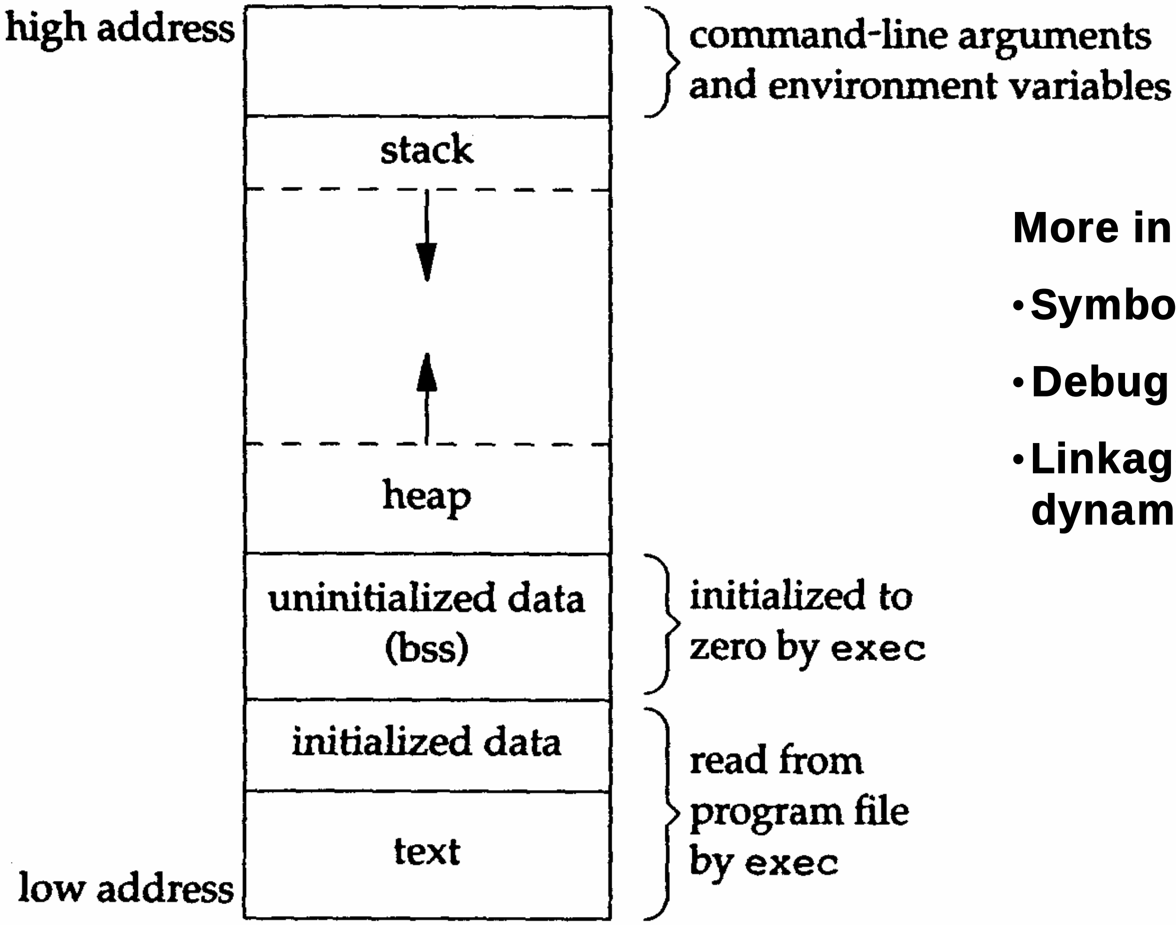
Environment List (Fig. 7.5)



Memory Layout of a C Program

- **Text segment**: Machine instructions (read-only, sharable)
- **Initialized data segment**:
  e.g. int maxcount = 99; (initialized!)
- **Uninitialized data segment**:
  (bss: block started by symbol)
  e.g. long sum[1000];
- **Stack**: automatic variables, function calling information, context-switch information, (recursive functions)
- **Heap**: dynamic memory allocation

Memory Layout (Fig. 7.6)


high address
stack
heap
uninitialized data (bss)
initialized data
text
low address
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec

More in a.out:

- **Symbol table**
- **Debug info**
- **Linkage tables for dynamic shared libs**

size

```
$ size /bin/cc /bin/sh
text  data  bss dec    hex
81920 16384 664  98968 18298 /bin/cc
90112 16384   0 106496 1a000 /bin/sh
```

Shared Libraries

- Common library routines removed from executable files
- Single copy of common library routines in memory is maintained
- No need to re-link edit every program if a library is updated or changed
- Size is smaller, some run-time overhead

Shared Libraries

Without Shared Libraries

```
$ ls –l a.out
-rwxrwxr-x 1 stevens 104859 Aug 2 14:25 a.out
$ size a.out
text   data   bss dec    hex
49152 49152 0    98304 18000
```

With Shared Libraries

```
$ ls –l a.out
-rwxrwxr-x 1 stevens 24576 Aug 2 14:26 a.out
$ size a.out
text   data   bss dec    hex
8192   8192   0    16384 4000
```

Shared Libraries

```
$ cc –static hello1.c
$ ls –l a.out
-rwxrwxr-x 1 sar 475570 Feb 18 23:17 a.out
$ size a.out
text     data   bss    dec      hex   filename
375657   3780   3220   382657   5d6c1 a.out
```

Shared Libraries

```
$ cc hello1.c
$ ls –l a.out
-rwxrwxr-x 1 sar 11410 Feb 18 23:19 a.out
$ size a.out
text  data  bss  dec   hex  filename
872   256   4    1132  46c  a.out
```

Memory Allocation

- malloc():
  - allocates specified #bytes,
  - initial value of memory is indeterminate
- calloc():
  - allocates specified #objects of specified size,
  - initialized to all 0 bits
- realloc():
  - changes size of previously allocated memory,
  - initial value of new area is indeterminate

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);
```

Return: nonnull pointer if OK,
NULL on error

```
void free(void *ptr);
```

Alternate Memory Allocators

- libmalloc
  - SVR4-based systems, such as Solaris
  - API match ISO C functions
  - mallopt: to control memory allocation operations
  - mallinfo: provide info on memory allocator

Alternate Memory Allocators

- vmalloc
  - Allows processes to allocate memory using different techniques for different regions
  - Emulations of ISO C memory allocation functions
  - Specific functions

Alternate Memory Allocators

- quick-fit
  - quick-fit memory allocation is faster than best-fit and first-fit (used by std malloc)
  - Splits memory info buffers of various sizes
  - Maintains unused buffers on different lists
  - Free implementations of malloc and free based on quick-fit available on FTP sites

Alternate Memory Allocators

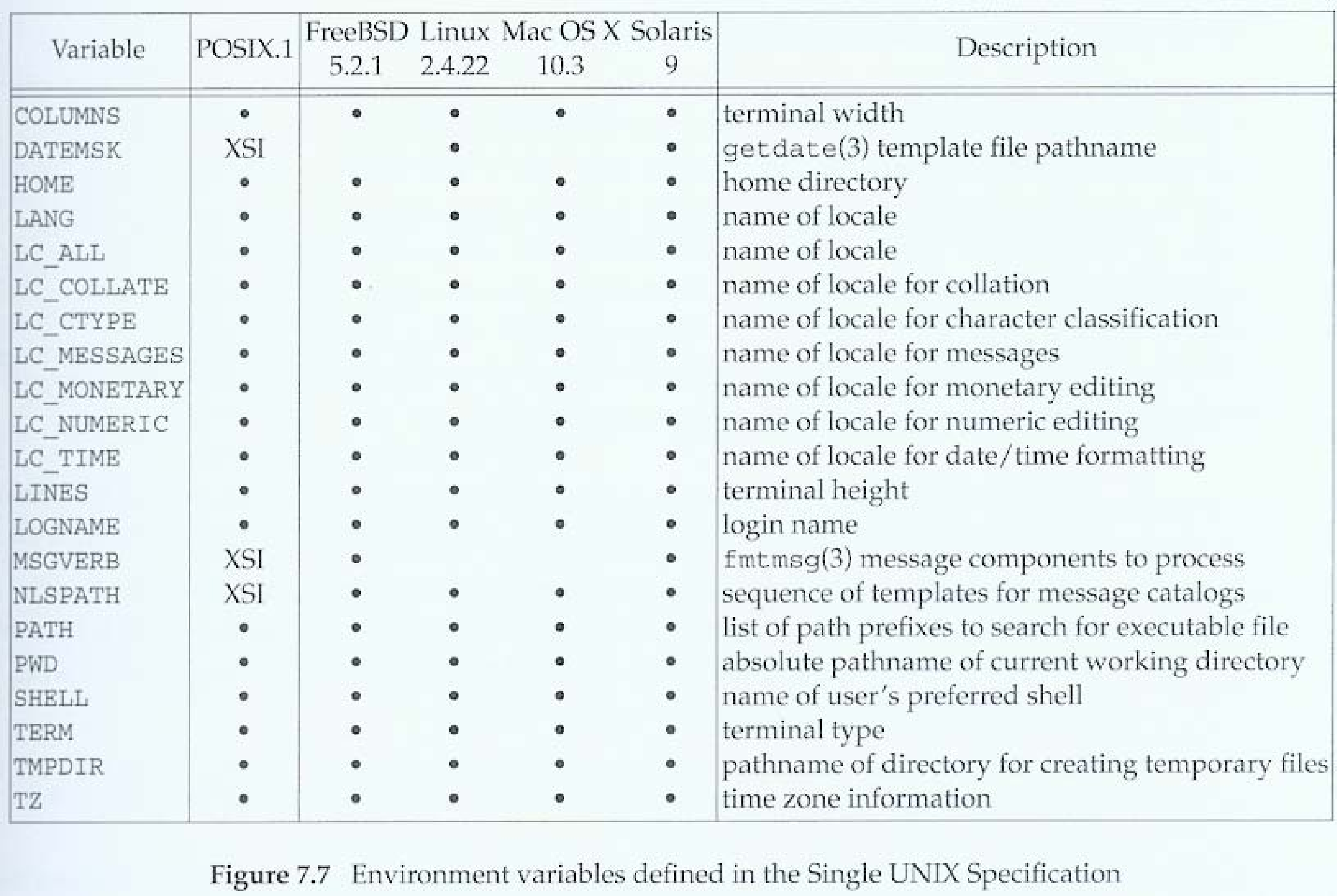
- alloca
  - Allocates memory from stack, instead of heap
  - Advantage: No need to free space, automatically freed after function returns
  - Disadvantage: Some systems do not support alloca()
  - However, all 4 platforms of textbook support it

Environment Variables

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

- Returns: pointer to value associated with name, NULL if not found
- Some environment variables are set automatically by shell upon login
- E.g.: HOME, USER, etc.

Environment Variables (Fig. 7.7)



Variable	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	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	XSI	•	•	•	•	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

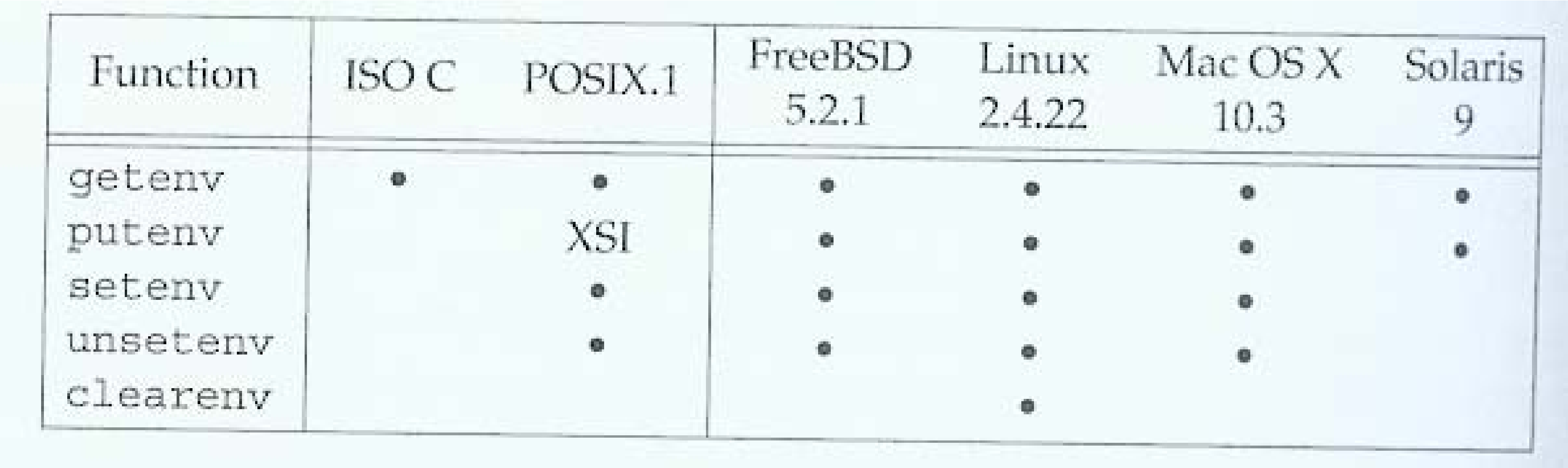
Figure 7.7 Environment variables defined in the Single UNIX Specification

Setting an environment variable

```
#include <stdlib.h>

int putenv(const char *str);

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

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

- Return: 0 if OK, nonzero on error

Environment Variables (Fig. 7.8)



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`				•		

Figure 7.8 Support for various environment list functions

setjmp(), longjmp() Functions

- In C, we cannot goto a label in another function
- setjmp() and longjmp() must be used
- See Figure 7.9 (a skeleton) for command processing
  - read lines (main),
  - interpret commands (do_line)
  - process command (cmd_add, …)

Figure 7.9 (1/4: main)

```
#include "apue.h"
#define         TOK_ADD         5

void    do_line(char *);
void    cmd_add(void);
int     get_token(void);

int main(void)
{
    charline[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
          do_line(line);
    exit(0);
}
```

Figure 7.9 (2/4: do_line)

```
char    *tok_ptr;          /* global pointer for get_token() */

void
do_line(char *ptr)         /* process one line of input */
{
    int                cmd;
tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
          switch (cmd) {   /* one case for each command */
          case TOK_ADD:
                           cmd_add();
                           break;
          }
    }
}
```

Figure 7.9 (3/4: cmd_add)

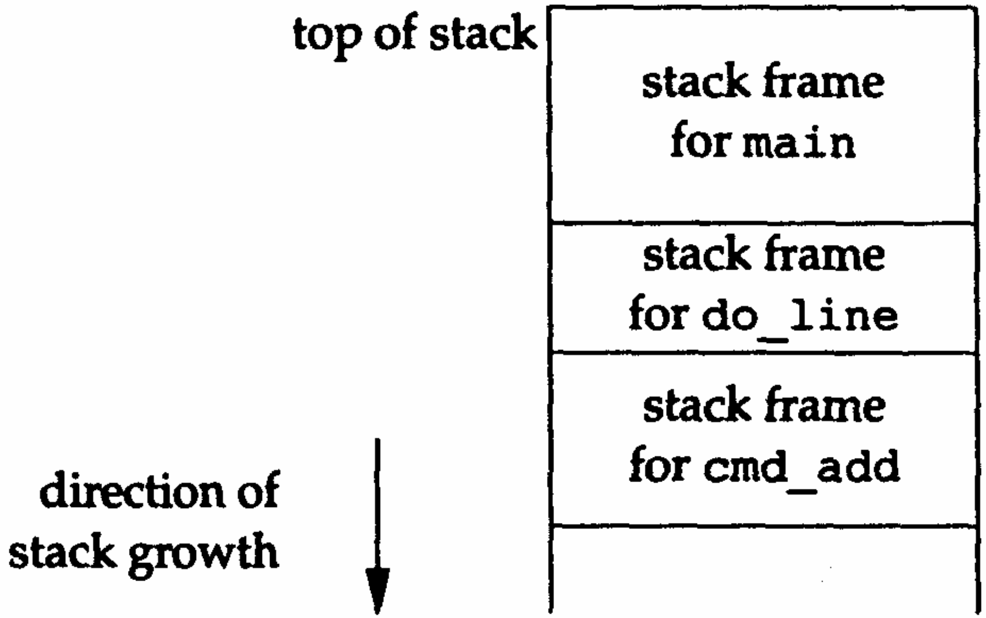
```
void
cmd_add(void)
{
   int          token;

   token = get_token();
   /* rest of processing for this command */
}
```

Figure 7.9 (4/4: get_token)

```
int
get_token(void)
{
  /* fetch next token from line pointed to
  by tok_ptr */
}
```

What if error occurs here?

After cmd_add(): stack frame



setjmp() and longjmp()

- Often we are deeply nested,
- An error occurs,
- We want to print an error, ignore rest of input, and return to main()
- Large # of levels → handle return at each level for each error
- Direct nonlocal goto: setjmp, longjmp

setjmp() and longjmp()

- #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*);

Figure 7.11

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

#defineTOK_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);
```

Figure 7.11 (cont'd)

```
    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 */
}
```

Figure 7.11

- setjmp(jmpbuffer) stores current state of main at the start of program exec
- longjmp(jmpbuffer, 1) unwounds the stacks of do_line() and cmd_add()
- and causes setjmp() to return 1

Automatic, Register, Volatile Variables

- After longjmp(), what are the values of the automatic and register variables?
  - Rolled back
  - Left alone
- Standards: indeterminate
- Volatile variables: don’t rollback values
- Global, static variables: leave alone

Program 7.5: longjmp() …

```
#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;
```

Program 7.5: longjmp() …

```
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);
}
```

Program 7.5: longjmp() …

```
    /*
     * 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);
}
```

Program 7.5: longjmp() …

```
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);
}
```

Figure 7.13: results

```
$ cc testjmp.c      // compile without any optimization
$ ./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

$ cc –O testjmp.c   // compile with full optimization
$ ./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
```

Figure 7.13: results

- setjmp(3) manual
  - Variables stores in memory will have values as of the time of the longjmp,
  - Whereas variables in the CPU and floating-point registers are restored to their values when setjmp was called.

Figure 7.13: results

- Without Optimization
  - All variables in memory
- With Optimization
  - autoval and regival go into registers
- Suggestion
  - Use “volatile” for portable code

Figure 7.14: Incorrect usage of automatic variables

```
#include      <stdio.h>
#define       DATAFILE     "datafile"

FILE *open_data(void)
{
    FILE *fp;
    char databuf[BUFSIZ];
              /* setvbuf makes this the stdio buffer */

    if ( (fp = fopen(DATAFILE, "r")) == NULL)
        return(NULL);

    if (setvbuf(fp, databuf, BUFSIZ, _IOLBF) != 0)
        return(NULL);

    return(fp);                  /* error */
}
```

local automatic variable!

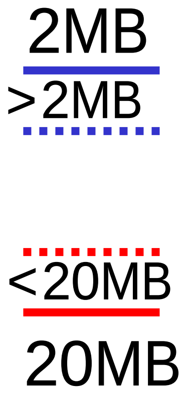
getrlimit(), setrlimit()

- Every process has resource limits

```
#include <sys/resource.h>
int getrlimit(    int resource,
                  struct rlimit *rlptr    );
int setrlimit(    int resource,
                  const struct rlimit *rlptr );
Return: 0 if OK, nonzero on error
```

Resource Limits

- struct rlimit {
    - rlim_t rlim_cur; /* soft limit: curr limit */
    - rlim_t rlim_max; /* hard limit: max */
- };



2MB

>2MB

<20MB

20MB

- Soft limit: can be changed by any process to ≤ hard limit
- Hard limit: can be lowered by any process to ≥ soft limit (irreversible!)
    - can be raised only by superuser process

Resource Limits

- Infinite Limit = RLIM_INFINITY
- RLIMIT_AS: #bytes for a process memory
- RLIMIT_CORE: #bytes in core file
- RLIMIT_CPU: #seconds of CPU time
- RLIMIT_DATA: #bytes of data seg = init data + uninit data + heap
- RLIMIT_FSIZE: #bytes of max file size
- RLIMIT_LOCKS: #file locks by a process

Resource Limits

- RLIMIT_MEMLOCK: #bytes locked by process in memory using mlock(2)
- RLIMIT_NOFILE: Max # open files
- RLIMIT_NPROC: Max # child processes
- RLIMIT_RSS: Max resident set size (bytes)
- RLIMIT_SBSIZE: #bytes of socket buffers
- RLIMIT_STACK: #bytes of stack size
- RLIMIT_VMEM: same as RLIMIT_AS

Resource Limits

- Resource limits are inherited by child processes
- For ALL processes to have same limits, shells has built-in commands:
  - ulimit (sh, bash, ksh, …)
  - limit (csh, tcsh, …)

Figure 7.16 (1/4)

```
#include "apue.h"
#if defined(BSD) || defined(MACOS)
#include <sys/time.h>
#define FMT   "%10lld "
#else
#define FMT   "%10ld "
#endif
#include <sys/resource.h>

#define         doit(name)     pr_limits(#name, name)

static void     pr_limits(char *, int);
```

ISO C string creation operator

Figure 7.16 (2/4)

```
int main(void)
{
#ifdef  RLIMIT_AS
    doit(RLIMIT_AS);
#endif
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);
#ifdef RLIMIT_LOCKS
    doit(RLIMIT_LOCKS);
#endif
#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
    doit(RLIMIT_NOFILE);
```

Figure 7.16 (3/4)

```
#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
#ifdef  RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
#endif
    doit(RLIMIT_STACK);
#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}
```

Figure 7.16 (4/4)

```
static void pr_limits(char *name, int resource)
{
    struct rlimit  limit;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite)  ");
    else
        printf(FMT, limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)");
    else
        printf(FMT, limit.rlim_max);
    putchar((int)'\n');
}
```

Figure 7.16: FreeBSD results

```
$ ./a.out
RLIMIT_CORE          (infinite)   (infinite)
RLIMIT_CPU           (infinite)   (infinite)
RLIMIT_DATA          536870912    536870912
RLIMIT_FSIZE         (infinite)   (infinite)
RLIMIT_MEMLOCK       (infinite)   (infinite)
RLIMIT_NOFILE              1735         1735
RLIMIT_NPROC                867          867
RLIMIT_RSS           (infinite)   (infinite)
RLIMIT_SBSIZE        (infinite)   (infinite)
RLIMIT_STACK          67108864     67108864
RLIMIT_VMEM          (infinite)   (infinite)
```

Figure 7.16: Solaris results

```
$ ./a.out
RLIMIT_AS         (infinite)  (infinite)
RLIMIT_CORE       (infinite)  (infinite)
RLIMIT_CPU        (infinite)  (infinite)
RLIMIT_DATA       (infinite)  (infinite)
RLIMIT_FSIZE      (infinite)  (infinite)
RLIMIT_NOFILE            256       65536
RLIMIT_STACK         8388608  (infinite)
RLIMIT_VMEM       (infinite)  (infinite)
```