Chapter 7: The Environment of a UNIX process
Introduction
The main() function

- int main(int argc, char* argv[]);
  - argc = number of command-line arguments
  - argv = array of pointers to the (string) arguments
  - main() is the first thing called in the program

- A special start-up routine is called first
  - This is what sets up the parameters to main
Process Termination

- Five ways to terminate a process

- Normal termination
  - Return from main()
  - Call exit()
  - call _exit()

- Abnormal termination
  - Call abort()
  - Terminate by a signal
exit() and _exit()

- **exit()** – Performs a clean shutdown of the standard I/O library
  - void exit(int status);

- **_exit()** – Same as exit except doesn’t call atexit()
  - void _exit(int status);
  - No cleanup, returns to kernel immediately

- Exit status undefined if not specified
Figure 7.2 How a C program is started and how it terminates
atexit() – Calls func without args when program terminates
- int atexit(void (*func)(void));
- func is a pointer to a function that takes no parameters

ANSI C: A process can register up to 32 handler functions to execute when the program exits
#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");
}

Figure 7.3  Example of exit handlers
Command-line arguments

- Programs can pass command-line parameters

```c
#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** Echo all command-line arguments to standard output
Each program is passed an environment list
- Extern char** environ;

Each environment string consists of name=value

Most names are uppercase

Usually ignored, but can be useful
- Why?

Historically environment was input to main
- int main(int argc, char* argv[], char* envp[]);
Figure 7.5  Environment consisting of five C character strings
Memory Layout of a C Program

- **Text segment**
  - The machine instructions of the program
  - Usually sharable and read-only

- **Data segment (initialized data)**
  - Global variables that are initialized in the program

- **BSS (uninitialized data)**
  - Global variables that are not initialized in the program
  - Initialized to zero or null pointers
... C program layout cont.

- Stack (automatic variables)
  - Function return information
  - Local variables

- Heap
  - Dynamic memory allocation

![Diagram of memory layout]

Figure 7.6  Typical memory arrangement
Shared Libraries

- Single shared copy of common library routines
  - Instead of each one being copied in each program

- Big space savings
  - 805,175 vs. 1,696 for hello world
Memory Allocation

- **malloc()** – Allocates the specified number of bytes
  - `void* malloc(size_t size);`
  - Uninitialized

- **calloc()** – Allocates space for the specified number of objects
  - `void* calloc(size_t nobj, size_t size);`
  - Initialized to all 0’s

- **realloc()** – Changes the size of a previously allocated area
  - `void* realloc(void* ptr, size_t newsize);`
  - May move to a new location (and copy old contents)
  - New area is uninitialized
Memory Allocation

- Memory is over-allocated
  - Additional space at the end of segment for record keeping
  - Writing past the end or before the start can overwrite the record
Freeing memory

- `free()` – Frees allocated space
  - `void* free(void* ptr);`

- Mistakes with free
  - Call free twice on same pointer
  - Call free on pointer not obtained from alloc function
  - Not calling it – causes leaks
alloca

- Allocates memory from stack
- Doesn’t have to be freed
- Doesn’t live past the return from the calling function
Environment variables

- Used by applications only
  - Not the kernel

- Name= value

- Common variables: HOME, USER, PRINTER, etc…

- getenv() – gets environment variables
  - char* getenv(const char* name);
  - Returns null if not found
<table>
<thead>
<tr>
<th>Variable</th>
<th>POSIX.1</th>
<th>FreeBSD 8.0</th>
<th>Linux 3.2.0</th>
<th>Mac OS X 10.6.8</th>
<th>Solaris 10</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMNS</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>terminal width</td>
</tr>
<tr>
<td>DATEMSK</td>
<td>XSI</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>getdate(3) template file pathname</td>
</tr>
<tr>
<td>HOME</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>home directory</td>
</tr>
<tr>
<td>LANG</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>name of locale</td>
</tr>
<tr>
<td>LC_ALL</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>name of locale</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>name of locale for collation</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>name of locale for character classification</td>
</tr>
<tr>
<td>LC_MESSAGES</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>name of locale for messages</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>name of locale for monetary editing</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>name of locale for numeric editing</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>name of locale for date/time formatting</td>
</tr>
<tr>
<td>LINES</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>terminal height</td>
</tr>
<tr>
<td>LOGNAME</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>login name</td>
</tr>
<tr>
<td>MSGVERB</td>
<td>XSI</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>fmtmsg(3) message components to process</td>
</tr>
<tr>
<td>NLS_PATH</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>sequence of templates for message catalogs</td>
</tr>
<tr>
<td>PATH</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>list of path prefixes to search for executable file</td>
</tr>
<tr>
<td>PWD</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>absolute pathname of current working directory</td>
</tr>
<tr>
<td>SHELL</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>name of user’s preferred shell</td>
</tr>
<tr>
<td>TERM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>terminal type</td>
</tr>
<tr>
<td>TMPDIR</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>pathname of directory for creating temporary files</td>
</tr>
<tr>
<td>TZ</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>time zone information</td>
</tr>
</tbody>
</table>

**Figure 7.7** Environment variables defined in the Single UNIX Specification
Other environment functions

- putenv() – creates (or overwrites) environment variable
  - int putenv(const char* str);

- setenv() - same as putenv except does nothing if rewrite = 0 and old value exists
  - int setenv(const char* name, const char* value, int rewrite);

- unsetenv() – Clears an environment variable
  - int unsetenv(const char* name);
<table>
<thead>
<tr>
<th>Function</th>
<th>ISO C</th>
<th>POSIX.1</th>
<th>FreeBSD 8.0</th>
<th>Linux 3.2.0</th>
<th>Mac OS X 10.6.8</th>
<th>Solaris 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>getenv</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>putenv</td>
<td>•</td>
<td>XSI</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>setenv</td>
<td>•</td>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>unsetenv</td>
<td>•</td>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>clearenv</td>
<td>•</td>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

**Figure 7.8** Support for various environment list functions
setjmp() and longjmp()

- Allow goto’s from lower in a call stack to higher in a call stack
- setjmp() – sets up location to jump to
  - int setjmp(jmp_buf env)
- longjmp() – jumps to location set by setjmp()
  - int longjmp(jmp_buf env, int val);
- Parameter contains the environment of the function that will be jumped to
- Spaghetti is good (delicious), but not for coding
#include "apue.h"
#define TOK_ADD 5

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

int main(void)
{
    char line[MAXLINE];
    while (fgets(line, MAXLINE, stdin) != NULL)
    {
        do_line(line);
        exit(0);
    }

    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;

            }
        }
    }

    void cmd_add(void)
    {
        int token;

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

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

    Figure 7.9 Typical program skeleton for command processing
#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 */
}

Figure 7.11  Example of setjmp and longjmp
Figure 7.10  Stack frames after `cmd_add` has been called

Figure 7.12  Stack frame after `longjmp` has been called
getrlimit() and setrlimit()

- **getrlimit()** – get resource limit
  - int getrlimit(int resource, struct rlimit* rlp); 

- **setrlimit()** – set resource limit
  - int setrlimit(int resource, const struct rlimit* rlp); 

```c
struct rlimit{
    rlim_t rlim_cur; /* Soft limit */
    rlim_t rlim_max; /* hard limit */
};
```
<table>
<thead>
<tr>
<th>Limit</th>
<th>XSI</th>
<th>FreeBSD 8.0</th>
<th>Linux 3.2.0</th>
<th>Mac OS X 10.6.8</th>
<th>Solaris 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_AS</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_CORE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>RLIMIT_CPU</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>RLIMIT_DATA</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>RLIMITFSIZE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>RLIMIT_MEMLOCK</td>
<td>•</td>
<td></td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_MSGQUEUE</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_NICE</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_NOFILE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_NPROC</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_NPTS</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_RSS</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_SBSIZE</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_SIGPENDING</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_STACK</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_SWAP</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_VMEM</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.15**  Support for resource limits