Chapter 7
Process Environment

Cheng-Hsin Hsu
National Tsing Hua University
Department of Computer Science

Parts of the course materials are courtesy of Prof. Chun-Ying Huang

Outline

- Process start and termination
- Environment variables
- Memory layout
- Shared libraries
- Memory allocation
- setjmp and longjmp
- Process resource limits

Process Start

- The main function
- Synopsis
  - `int main(int argc, char *argv[]);`
  - `int main(int argc, char *argv[], char *envp[]);`

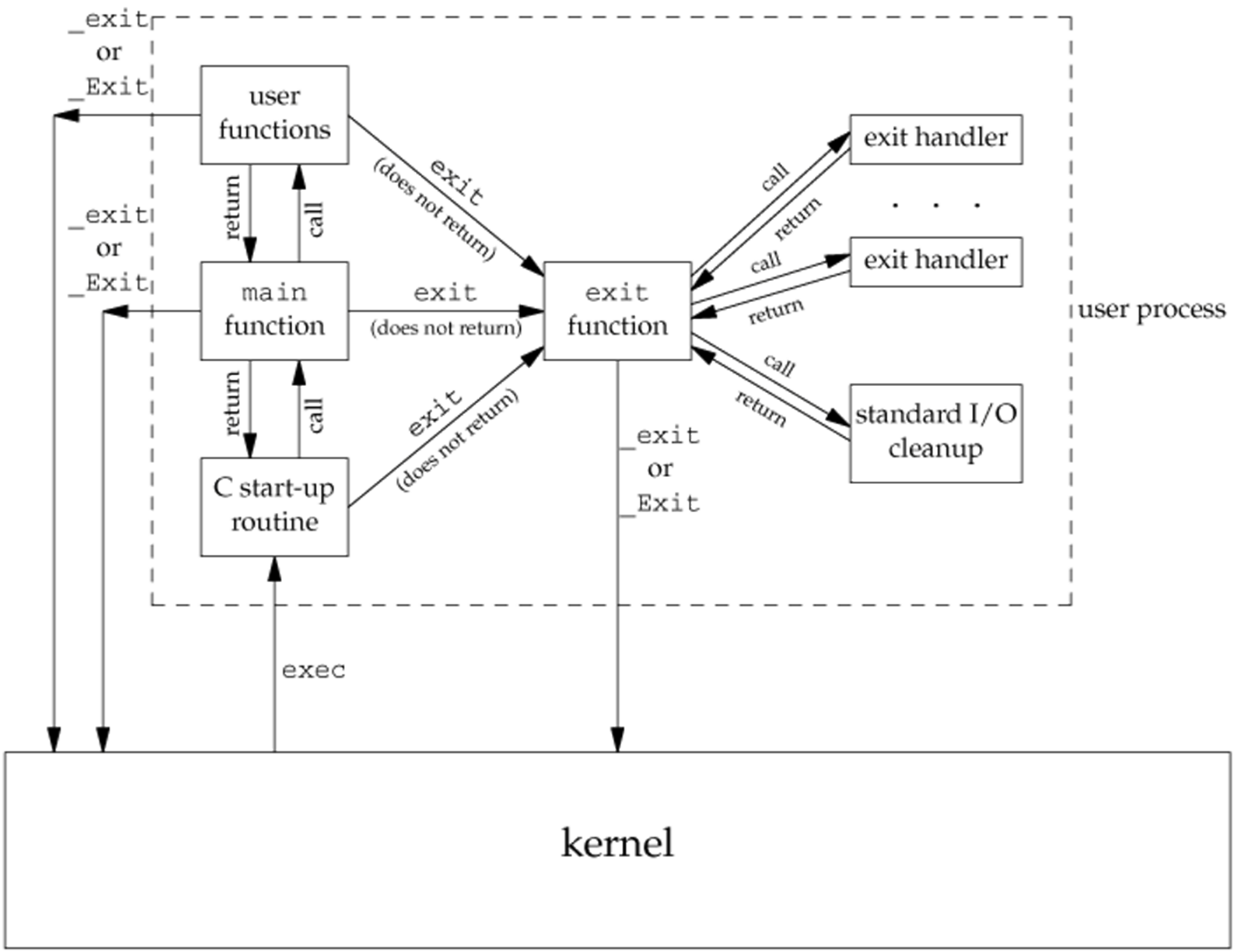
Process Termination

- Normal process termination in five ways
  - 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 process termination in three ways
  - Calling abort
  - Receipt of a signal
  - Response of the last thread to a cancellation request
- Execution of a main function looks like
  - exit(main(argc, argv));

atexit and exit Functions

- Manual cleanups on exit
  - `int atexit(void (*function)(void));`
  - Register up to 32 customized functions (textbook)
    - Linux has extended this restrictions
- Exit functions
  - exit
    - Call atexit registered functions
    - Performed a clean shutdown of the standard I/O library
    - fclose() all streams, remove tmpfile()
  - _exit and _Exit
    - Terminate immediately

Start and Termination of a C Program



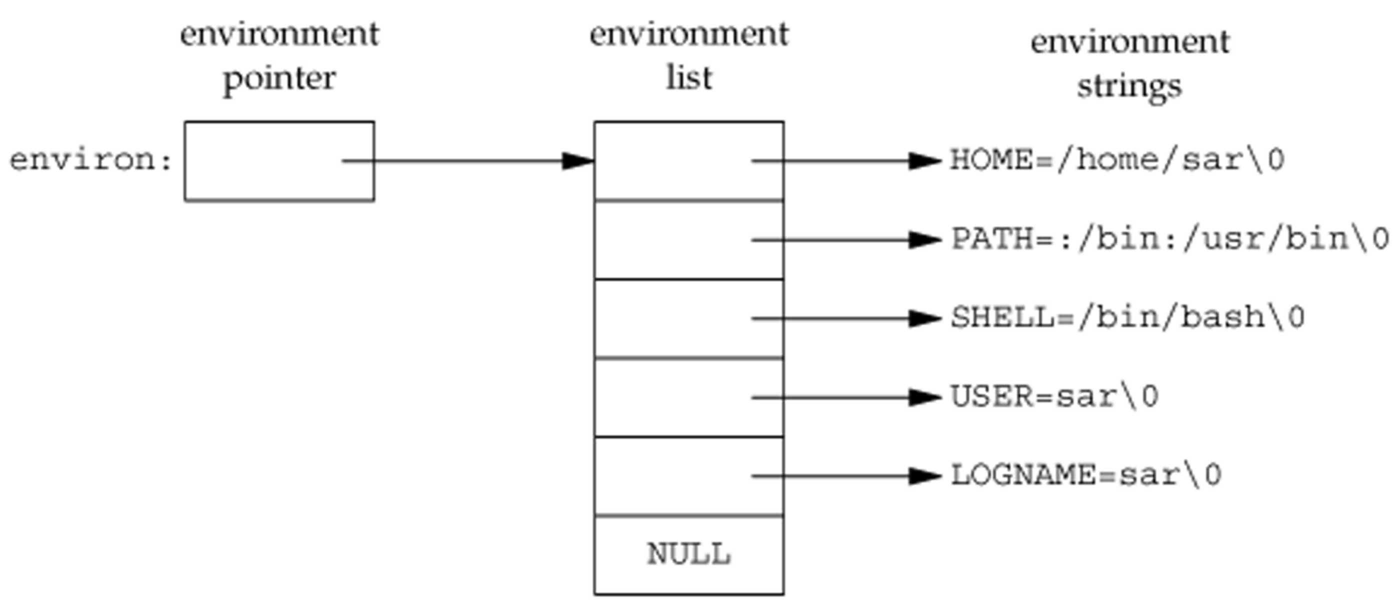
Environment Variables

- The environment variables
  - Usually in the form of: `name=value` (no spaces around =)
  - Relevant commands: env, export (bash)
  - Use $ to read a specific environment variable in a shell
- List of environment variable functions

Function	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				•		

Environment List

- Access environment variables directly
  - `int main(int argc, char *argv[], char *envp[]);`
  - `extern char **environ;`



Environment Functions

- Prototypes of functions to manipulate environment variables

```
#include <stdlib.h>

char *getenv(const char *name);
int putenv(char *string);

int setenv(const char *name, const char *value, int overwrite);
int unsetenv(const char *name);
int clearenv(void);
```

Environment List Operations

- Delete an entry
  - This is simple, just free a string and move all subsequent pointers down one
- Modify an entry
  - If new-size ≥ old-size, just overwrite the old one
  - If new-size > old-size, allocate a new space the new variable and make the pointer point to the new location
- Add an entry
  - Add for the 1st time, allocate a new space for the entire list
  - Add for non-1st time, reallocate a larger space for the entire list

Common Environment Variables (1/3)

Variable	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
COLUMNS	•	•	•	•	•	Terminal width
DATEMASK	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

Common Environment Variables (2/3)

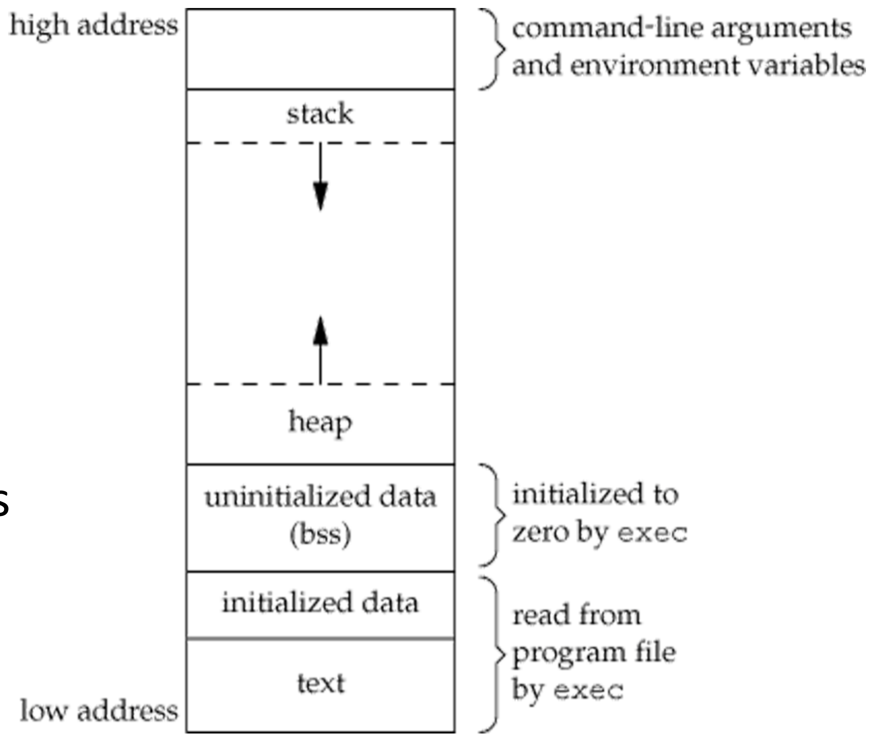
Variable	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
LC_MONETARY	•	•	•	•	•	Name of locale for monetary editing
LC_NUMERIC	•	•	•	•	•	Name of locale for numeric editing
LC_TIME	•	•	•	•	•	Name of locale for date/time formatting
LINES	•	•	•	•	•	Terminal height
LOGNAME	•	•	•	•	•	Login name
MSGVERB	XSI	•	•	•	•	fmtmsg(3) message components to process
NLSPATH	•	•	•	•	•	Sequence of templates for message catalogs

Common Environment Variables (3/3)

Variable	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
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

Memory Layout of a Program

- Text segment
  - Machine instructions
- Initialized data segment
  - int maxcount = 100;
- Uninitialized data segment (bss)
  - long sum[1000];
- Stack
  - Local variables, function call states
- Heap
  - Dynamic allocated memory



Read Sizes of an Executable Binary

- The size (1) command

```
$ size /usr/bin/gcc /bin/sh
   text    data     bss     dec     hex filename
 203913    2152    2248  208313   32db9 /usr/bin/gcc
 704028   19268   19736  743032   b5678 /bin/sh
```

Shared Libraries

- Most UNIX systems today support shared libraries
- Shared libraries remove the common library routines from the executable file
- Maintain a single copy of the library routine somewhere in memory that all processes reference
  - Reduce the size and memory requirement of each executable file
  - But It may add some runtime overhead
- Another advantage of shared libraries
  - Library functions can be replaced with new versions without having to relink every program that uses the library
  - But it might also be a security flaw

Compile Static and Dynamic Program

- A simple program that just print "Hello, World!"

```
$ gcc h1.c -o h1
$ gcc h2.c -o h2 -static
$ ls -la h1 h2
-rwxrwxr-x 1  bear  bear    9564 Mar 13 11:48 h1
-rwxrwxr-x 1  bear  bear  878192 Mar 13 11:48 h2
$ size h1 h2
   text    data     bss     dec     hex filename
    896     264       8    1168     490 h1
 499650    1928    6948  508526   7c26e h2
```

Library Injection

- Functions referenced to shared libraries can be overridden
  - The LD_PRELOAD environment variable
  - Usage:
    LD_PRELOAD=/path/to/the/injected-shared-object {program}
- Library injection does not work with suid/sgid executables

Library Injection Example

- Suppose we are going to hijack the getuid() function
  - This is commonly used in tools like fake-root
- The original program (getuid.c)

```
int main() {
        printf("UID = %d\n", getuid());
        return 0;
}
```

- The injected library (inject1.c)

```
#include <stdio.h>
#include <sys/types.h>

uid_t getuid(void) {
        fprintf(stderr, "injected getuid, always return 0\n");
        return 0;
}
```

Library Injection Example (Cont'd)

- Compile the programs and the libraries

```
$ gcc -o getuid -Wall -g getuid.c
$ gcc -o inject1.so -shared -fPIC inject1.c -ldl
```

  - The first command produces the getuid program
  - The second commands generates the inject1.so (shared) library
- Run the example

```
$ ./getuid                   # no injection
UID = 1000
$ LD_PRELOAD=./inject1.so ./getuid   # injected
injected getuid, always return 0
UID = 0
```

More on Library Injection

- But we still want the original function to work properly
- We have to locate the original function

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
char *dlerror(void);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
```

- You may have to link with `-ldl` option

Revised Library Injection Example

- We would like to know the real UID internally (inject2.c)

```
#include <dlfcn.h>
#include <stdio.h>
#include <sys/types.h>

static uid_t (*old_getuid)(void) = NULL;   /* function pointer */

uid_t getuid(void) {
        if(old_getuid == NULL) {
                void *handle = dlopen("libc.so.6", RTLD_LAZY);
                if(handle != NULL)
                        old_getuid = dlsym(handle, "getuid");
        }
        fprintf(stderr, "injected getuid, always return 0\n");
        if(old_getuid != NULL)
                fprintf(stderr, "real uid = %d\n", old_getuid());
        return 0;
}
```

Revised Library Injection Example (Cont'd)

- Compile the programs and the libraries (again)

```
$ gcc -o getuid -Wall -g getuid.c
$ gcc -o inject2.so -shared -fPIC inject2.c -ldl
```

  - The first command produces the getuid program
  - The second commands generates the inject2.so (shared) library

- Run the example

```
$ ./getuid                  # no injection
UID = 1000
$ LD_PRELOAD=./inject2.so ./getuid   # injected
injected getuid, always return 0
real uid = 1000
UID = 0
```

Determine Library Injection Possibility

- No SUID/SGID enabled
- Not a statically linked binary
- Examples of the dynamic/static linked hello-world example
  - The `file` command

```
$ file h1 h2
h1: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=e32f08cfbdda94d57273829c2bfd535d8fbe626d, not stripped
h2: ELF 64-bit LSB  executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux 2.6.24, BuildID[sha1]=2748d80822e76d183d0ef5633c0b784527727c7a, not stripped
```

  - The `ldd` command

```
$ ldd h1 h2
h1:
        linux-vdso.so.1 =>  (0x00007ffe7d3d5000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1bc2150000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f1bc2515000)
h2:
        not a dynamic executable
```

Determine Library Injection Possibility (Cont'd)

- Use symbols from a shared library
- The nm command
- Example: static VS dynamic linked symbols

```
$ gcc -o getuid -Wall -g getuid.c              # dynamically linked
$ gcc -o getuid_s -Wall -g getuid.c -static    # statically linked
$ nm getuid | grep getuid
                 U getuid@@GLIBC_2.2.5         # getuid is unknown
$ nm getuid_s | grep getuid
0000000000433590 W getuid                      # getuid is known (but weak)
0000000000433590 T __getuid                    # the getuid implementation
```

- Symbols can be stripped using the `strip` command

Memory Allocation

- ISO C memory allocation functions
- void *malloc(size_t size);
  - Allocates a specified number of bytes of memory
  - The initial value of the memory is indeterminate
- void *calloc(size_t nobj, size_t size);
  - Allocates space for a specified number of objects of a specified size
  - The space is initialized to all 0 bits
- void *realloc(void *ptr, size_t newsize);
  - Increases or decreases the size of a previously allocated area
  - It may involve moving the previously allocated area somewhere else, to provide the additional room at the end
  - The initial value of increased memory is indeterminate

Memory Allocation (Cont'd)

- Allocated memory can be released by free()
- The allocation routines are usually implemented with the sbrk(2) system call
- This system call expands (or contracts) the heap of the process
  - However, most versions of malloc and free never decrease their memory size
  - The space that we free is available for a later allocation
  - The freed space is usually kept in the malloc pool, not returned to the kernel

The alloca Function

- A special memory allocation function – alloca

```
#include <alloca.h>
void *alloca(size_t size);
```

- alloca() allocate memories in stack frames of the current function call
- So you don't have to free() the memory – it is released automatically after the execution of the current function returns
- May be not supported by your system, but modern UNIXes supports the function (Linux, FreeBSD, Mac OS X, Solaris)
- Pros: might be faster (than malloc), no need to free, easier to work with setjmp/longjmp
- Cons: Portability

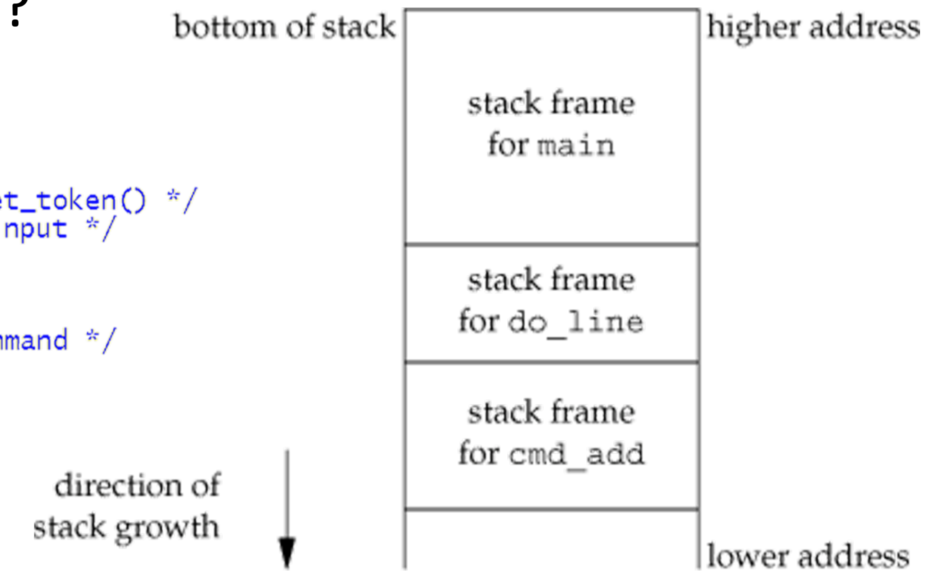
setjmp and longjmp Function

- The reserved keyword "goto" can be used only in the same function
- We cannot goto a label that is in another function
- Instead, we must use the setjmp and longjmp functions to perform this type of branching

Typical Program Skeleton for Command Processing

```
#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 */
}
```

- What if we encounter an error in cmd_add and would like to jump back to the main function for processing the next line?



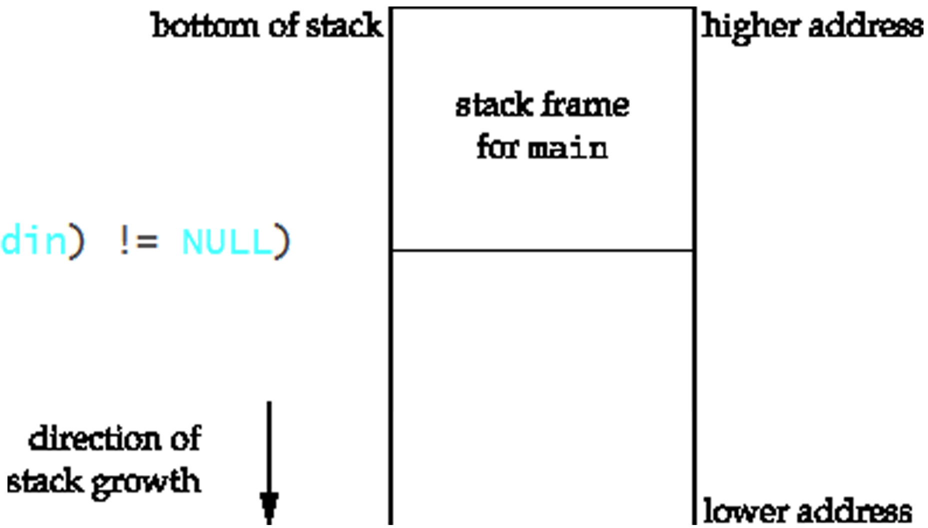
The Solution for Jumping Across Functions

- Set the jump back position
  - int setjmp(jmp_buf env);
  - env is usually a global variable – has to be accessed from both the setjmp side and the longjmp side
  - Returns: 0 if called directly, or nonzero if returning from a call to longjmp
- Jump back
  - void longjmp(jmp_buf env, int val);
    - The 'val' will be returned from setjmp
    - If val is 0, it will be replaced by 1

Using 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 */
}
```

- Stack after jumped back



Restoration of Variables (1/4)

- Type of variables
  - Automatic, e.g., [auto] int autoVal;, the default
  - Register, e.g., register int regVal;, store in register if possible
  - Volatile, e.g., volatile int volVal;, store in memory
- What are the values of variables after jumped back?
  - It depends
  - Most implementations do not try to roll back these automatic variables and register variables
  - The standards say only that their values are indeterminate
  - If you have an automatic variable that you do not want to be rolled back, define it with the volatile attribute
  - Variables that are declared global or static are left alone when longjmp is executed
  - In short: variables in register – restored; variables in memory – kept

Restoration of Variables (2/4)

```
#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);
}

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); }
```

Restoration of Variables (3/4)

- Rules for variable restoration
  - Variables stored in memory will have values as of the time of calling longjmp
  - Variables in the CPU and floating-point registers are restored to their values when setjmp was called
- Hence,
  - auto variables may be indeterminate, it depends on compiler implementations
  - register variables are restored to the value of “before calling **setjmp**”
  - volatile variable are restored to the value of “before calling **longjmp**”

Restoration of Variables (4/4)

- Set 1,2,3,4,5 → setjmp → Set 95,96,97,98,99 → longjmp → ?
  - No optimization: gcc places everything in memory
  - Full optimization: auto/register variables are placed in registers

```
$ gcc fig7.13-testjmp.c -I../include -o t1       compile without any optimization
$ gcc fig7.13-testjmp.c -I../include -o t2 -O    compile with full optimization
$ ./t1
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
$ ./t2
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

Process Resource Limits

- Every process has a set of resource limits
- Resource limits are usually initialized by a parent process and inherited by its child processes
- The getrlimit and setrlimit functions

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

- The rlimit structure

```
struct rlimit {
    rlim_t rlim_cur;  /* Soft limit */
    rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};
```

Partial List of Process Resources

Limit	XSI	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
RLIMIT_AS	•	•	•		•
RLIMIT_CORE	•	•	•	•	•
RLIMIT_CPU	•	•	•	•	•
RLIMIT_DATA	•	•	•	•	•
RLIMIT_FSIZE	•	•	•	•	•
RLIMIT_MEMLOCK		•	•	•	
RLIMIT_NOFILE	•	•	•	•	•
RLIMIT_NPROC		•	•	•	
RLIMIT_RSS		•	•	•	
RLIMIT_SBSIZE		•			
RLIMIT_STACK	•	•	•	•	•
RLIMIT_VMEM					•

Example to Dump Resource Limits

- See code fig7.16-getrlimit.c

```
$ ./fig7.16-getrlimit
RLIMIT_AS        (infinite)  (infinite)
RLIMIT_CORE      1024000000  (infinite)
RLIMIT_CPU       (infinite)  (infinite)
RLIMIT_DATA      (infinite)  (infinite)
RLIMIT_FSIZE     (infinite)  (infinite)
RLIMIT_LOCKS     (infinite)  (infinite)
RLIMIT_MEMLOCK        65536       65536
RLIMIT_NOFILE          1024        4096
RLIMIT_NPROC          96120       96120
RLIMIT_RSS       (infinite)  (infinite)
RLIMIT_STACK        8388608  (infinite)
```

Example to Dump Resource Limits

Limits	Description
RLIMIT_CORE	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
RLIMIT_MEMLOCK	The maximum amount of memory in bytes that a process can lock into memory using mlock(2).
RLIMIT_NOFILE	The maximum number of open files per process.
RLIMIT_NPROC	The maximum number of child processes per real user ID.
RLIMIT_STACK	The maximum size in bytes of the stack.

Assignment #5 (5%)

- Different from the prior assignments, this is a paper-based one. Please write the 10 exercise questions (each worth 0.5 point) of Chapter 7 of the textbook (third edition)
- You are encouraged to discuss. However, copying others' solutions is prohibited and you may suffer from penalty
- Due date: Nov. 14th, please turn in when we have the class on Monday
- No lab/demo session on Nov. 15th

7.1 On an Intel x86 system under Linux, if we execute the program that prints "hello, world" and do not call `exit` or `return`, the termination status of the program—which we can examine with the shell—is 13. Why?

7.2 When is the output from the `printf`s in Figure 7.3 actually output?

7.3 Is there any way for a function that is called by `main` to examine the command-line arguments without (a) passing `argc` and `argv` as arguments from `main` to the function or (b) having `main` copy `argc` and `argv` into global variables?

7.4 Some UNIX system implementations purposely arrange that, when a program is executed, location 0 in the data segment is not accessible. Why?

7.5 Use the `typedef` facility of C to define a new data type `Exitfunc` for an exit handler. Redo the prototype for `atexit` using this data type.

7.6 If we allocate an array of `long`s using `calloc`, is the array initialized to 0? If we allocate an array of pointers using `calloc`, is the array initialized to null pointers?

7.7 In the output from the `size` command at the end of Section 7.6, why aren't any sizes given for the heap and the stack?

7.8 In Section 7.7, the two file sizes (879443 and 8378) don't equal the sums of their respective text and data sizes. Why?

7.9 In Section 7.7, why does the size of the executable file differ so dramatically when we use shared libraries for such a trivial program?

7.10 At the end of Section 7.10, we showed how a function can't return a pointer to an automatic variable. Is the following code correct?

```
int
f1(int val)
{
    int     num = 0;
    int     *ptr = &num;

    if (val == 0) {
        int     val;

        val = 5;
        ptr = &val;
    }
    return(*ptr + 1);
}
```