系統程式設計
Systems Programming

鄭卜壬教授
臺灣大學資訊工程系

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Contents

1. OS Concept & Intro. to UNIX
2. UNIX History, Standardization & Implementation
3. File I/O
4. Standard I/O Library
5. Files and Directories
6. System Data Files and Information
7. Environment of a Unix Process
8. Process Control
9. Signals
10. Inter-process Communication
11. Thread Programming
12. Networking

The Environment of a Unix Process

Objectives:

- How a process is executed and terminates?
- How the command line arguments are passed to the new process?
- What the file format of a program is?
- What the typical memory layout is?
- How the process can use environment variables?
- Related functions and resource limits.

Process

- **A *process* is an instance of a running program.**
  - Not the same as “program” or “processor”
- **Process provides each program with two key abstractions:**
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - Private address space
    - Each program seems to have exclusive use of main memory.
- **How are these illusions maintained?**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system

main Function

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

- The starting point of a C program. Programs written in different languages have a different name.
  - e.g., mainCRTStartup, WinMainCRTStartup
- A special start-up routine is called to set things up first before call main()
  - Set up by the link editor (invoked by the compiler)

C Run-time Startup

Startup Routines in UNIX

hello.c

```
#include <stdio.h>

int main(void)
{
        printf( "Hello world!\n");
        return 0;
}
```

no main function

```
pjcheng@stego:~> gcc myhello.c -o myhello
/usr/lib/crt1.o(.text+0x81): In function `_start':
: undefined reference to `main'
pjcheng@stego:~> cat myhello.c
#include <stdio.h>

int mymain(void)
{
        printf( "Hello world!\n");
        return 0;
}
```

nm
- list symbols from object files

```
pjcheng@stego:~> nm -g hello
08049644 A _DYNAMIC
080496f0 A _GLOBAL_OFFSET_TABLE_
         w _Jv_RegisterClasses
0804970c A __bss_start
         w __deregister_frame_info
08049638 D __dso_handle
08049634 D __progname
         w __register_frame_info
0804970c A _edata
0804972c A _end
08048584 T _fini
08048368 T _init
         U _init_tls
080483d0 T _start
         U atexit
08049728 B environ
         U exit
08048524 T main
         U printf
```

(-g: only external symbols)

readelf
- display info. about ELF files

```
pjcheng@stego:~> readelf -a hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 09 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - FreeBSD
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x80483d0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          2260 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         5
  Size of section headers:           40 (bytes)
  Number of section headers:         23
  Section header string table index: 20
```

objdump – display info. about obj. files

objdump –d hello

(-d: disassemble)

```
080483d0 <_start>:
 80483d0:       55                      push   %ebp
 80483d1:       89 e5                   mov    %esp,%ebp
 80483d3:       57                      push   %edi
 80483d4:       56                      push   %esi
 80483d5:       53                      push   %ebx
 80483d6:       83 ec 0c                sub    $0xc,%esp
 80483d9:       83 e4 f0                and    $0xfffffff0,%esp
 80483dc:       8b 5d 04                mov    0x4(%ebp),%ebx
 80483df:       89 d7                   mov    %edx,%edi


 8048420:       b8 44 96 04 08          mov    $0x8049644,%eax
 8048425:       85 c0                   test   %eax,%eax
 8048427:       74 41                   je     804846a <_start+0x9a>
 8048429:       89 3c 24                mov    %edi,(%esp)
 804842c:       e8 7b ff ff ff          call   80483ac <_init+0x44>
 8048431:       c7 04 24 84 85 04 08    movl   $0x8048584,(%esp)
 8048438:       e8 6f ff ff ff          call   80483ac <_init+0x44>
 804843d:       e8 26 ff ff ff          call   8048368 <_init>
 8048442:       89 74 24 08             mov    %esi,0x8(%esp)
 8048446:       8d 45 08                lea    0x8(%ebp),%eax
 8048449:       89 44 24 04             mov    %eax,0x4(%esp)
 804844d:       89 1c 24                mov    %ebx,(%esp)
 8048450:       e8 cf 00 00 00          call   8048524 <main>
 8048455:       89 04 24                mov    %eax,(%esp)
 8048458:       e8 5f ff ff ff          call   80483bc <_init+0x54>
 804845d:       8d 76 00                lea    0x0(%esi),%esi
 8048460:       89 15 34 96 04 08       mov    %edx,0x8049634
 8048466:       89 d1                   mov    %edx,%ecx
 8048468:       eb af                   jmp    8048419 <_start+0x49>
 804846a:       e8 2d ff ff ff          call   804839c <_init+0x34>
 804846f:       eb c0                   jmp    8048431 <_start+0x61>
```

Process Termination

- **Eight ways to terminate:**
  - Normal termination
    - Return from main()
      - exit(main(argc, argv));
    - Call `exit()`
    - Call `_exit()` or `_Exit()`
    - Return of the last thread from its start routine
    - Calling `pthread_exit` from the last thread.
  - Abnormal termination (Chapter 10)
    - Call abort()
    - Be terminated by a signal
    - Response of the last thread to a cancellation request.

How a C program is started and terminated.

Process Termination

#include <stdlib.h>

void exit(int status);

- Perform cleanup processing
  - Call exit handlers
  - Close and flush I/O streams (fclose)
  - Called once, never returns
  - Puts the process into “zombie” status

void _Exit(int status);

- ANSI C

#include <unistd.h>

void _exit(int status);

- POSIX.1

- **Exit status:**
  - Undefined exit status:
    - Exit/return without status.
    - main() is not declared to be an integer.
  - 0 as the status:
    - main() is declared to be an integer and main() falls off the end.
    - return(0) and exit(0).

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

Example of Exit Status

```
$ cc hello.c
$ ./a.out
hello, world
$ echo $?                    print the exit status
13
```

```
$ cc -std=c99 hello.c        enable gcc's 1999 ISO C extensions
hello.c:4: warning: return type defaults to 'int'
$ ./a.out
hello, world
$ echo $?
0
```

Process Termination

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

- At least 32 functions called by *exit* – ANSI C, supported by SVR4&4.3+BSD
- The same exit functions can be registered for several times.
- Exit functions/handlers will be called in reverse order of their registration.
- Exit handlers registered will be cleared if exec() is called.

Example of atexit()

```
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");
}
```

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

Discussion

- atexit should not call exit(); use _exit() if necessary.
- There is no way to un-register an exit handler in some systems.
- How to design a function to un-register an exit handler?

Command-Line Arguments & Environment Variables

- **Command-Line Arguments**
  - argv[argc] is NULL under POSIX.1 & ANSI C
- **Environment Variables**
  - int main(int agrc, char **argv, char **envp);

- **getenv/putenv**

Example of Command-line Arguments

```
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);
}
```

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

Alternative code:

```
for (i = 0; argv[i] != NULL; i++)
```

File Format/Layout of a Program

- **ELF (Executable and Linkable Format)**
  - Standard library format for object files
  - Derived from AT&T System V Unix
    - Later adopted by BSD Unix variants and Linux
  - One unified format for
    - Relocatable object files (.o)
    - Executable object files
    - Shared object files (.so)
  - Generic name: ELF binaries
  - Better support for shared libraries than old **a.out** formats.

ELF Object File Format

- ELF header
  - Magic number(\177ELF), type (.o, exec, .so), machine, byte ordering, entry point, size, etc
- Program header table
  - Page size, virtual addresses memory segments (sections), segment sizes
- **.text** section
  - Code
- **.data** section
  - Initialized (static) data
- **.bss** section
  - Uninitialized (static) data
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space

ELF header
Program header table (required for executables)
`.text` section
`.data` section
`.bss` section
`.symtab`
`.rel.txt`
`.rel.data`
`.debug`
Section header table (required for relocatables)

ELF Object File Format (cont.)

- **.symtab** section
  - Symbol table
  - Procedures and static variable names
  - Section names and locations
- **.rel.text** section
  - Relocation info for **.text** section
  - Addresses of instructions that will need to be modified in the executable.
  - Instructions for modifying.
- **.rel.data** section
  - Relocation info for **.data** section
  - Addresses of pointer data that will need to be modified in the merged executable.
- **.debug** section
  - Info for symbolic debugging (gcc –g)

ELF header
Program header table (required for executables)
`.text` section
`.data` section
`.bss` section
`.symtab`
`.rel.txt`
`.rel.data`
`.debug`
Section header table (required for relocatables)

Linking Example

Code consists of symbol definitions and references
Each symbol definition has memory address
References can be either local or external

m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

Def of local symbol e

Ref to external symbol exit (defined in libc.so)

Ref to external symbol a

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

Def of local symbol ep

Def of local symbol a

Ref to external symbol e

Defs of local symbols x and y

Relocating Object Files

- Relocate and merge into an executable object file

Flows of compiling, loading, and running a program

Executable code file - hello

when execute the program

Loader (ld.so)

libraries (*.so) /run time objects/modules

system, argc, argv, env

stack segment

lib*.so

heap segment

data segment

malloc.o (lib*.a)
printf.o (lib*.a)

main.o

crt0.o/crt1.o (startup routine)

Memory Layout of a Process

- **Pieces of a process**
  - Text Segment
    - Read-only usually, sharable
  - Initialized Data Segment
    - int maxcount = 10;
  - Uninitialized Data Segment – bss (Block Started by Symbol)
    - Initialized to 0 by *exec*
    - long sum[1000];
  - Stack – return addr, automatic var, etc.
  - Heap – dynamic memory allocation (malloc)

Read from program file by *exec*

Loading Executable Binaries

Typical Memory Arrangement

High address

command-line arguments and environment variables.

Initialized to 0 by exec.

Read from program file by exec.

Low address

Virtual Memory vs. Physical Memory

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2006
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Virtual Memory

- Address translation: Hardware converts virtual addresses to physical addresses via OS-managed lookup table (page table)

Virtual Address Spaces

- **Two processes share physical pages by mapping in page table**
- **Separate Virtual Address Spaces (separate page table)**
  - Virtual and physical address spaces divided into equal-sized blocks
    - Blocks are called “pages” (both virtual and physical)
  - Each process has its own virtual address space
    - OS controls how virtual pages as assigned to physical memory

Virtual memory view
Page Table
Physical memory view
1111 1111
stack
1110 0000
1100 0000
heap
1000 0000
data
0100 0000
code
0000 0000
page # offset
11111 11101
11110 11100
11101 10111
11100 10110
11011 null
11010 null
11001 null
11000 null
10111 null
10110 null
10101 null
10100 null
10011 null
10010 10000
10001 01111
10000 01110
01111 null
01110 null
01101 null
01100 null
01011 01101
01010 01100
01001 01011
01000 01010
00111 null
00110 null
00101 null
00100 null
00011 00101
00010 00100
stack
1110 0000
stack
data
0101 000
code
0001 0000
0000 0000
Allocate new pages where room!

Address Translation via Page Table♪

Virtual Memory

- **The translation from virtual memory address to physical address has to be *LIGHT* fast.**
  - Not possible to be done by operating systems.
  - Done with hardware support: Memory Management Unit (MMU)
- **'Most' modern processors support MMU: Pentium, PowerPC, etc.**
- **Processors for embedded systems may not support MMU.**

An example

```
#include <stdio.h>
int main(void)
{
        printf("hello world\n");
        return 0;
}
```

- Compile it and check the size of each segment.

```
$gcc -c HelloWorld.c
$size HelloWorld.o
text     data     bss     dec     hex filename
64        0        0      64      40 HelloWorld.o
```

- Try to add global variables and see what's the difference.
- objdump can show you more detail information for your program.

Stack Frame

- **When one function is called, one area of memory, called *frame* or *stack frame*, is set aside for the function. Each frame contains:**
  - Return address
  - Passed parameter(s)
  - Saved registers
  - Local variable(s)
- **The stack frame is reclaimed by the system when a function returns.**

```
1.    #include <stdio.h>
2.    void first_function(void);
3.    void second_function(int);
4.
5.
6.    int main(void)
7.    {
8.       printf("hello world\n");
9.       first_function();
10.      printf("goodbye goodbye\n");
11.
12.      return 0;
13.   }
14.
15.
16.   void first_function(void)
17.   {
18.      int imidate = 3;
19.      char broiled = 'c';
20.      void *where_prohibited = NULL;
21.
22.      second_function(imidate);
23.      imidate = 10;
24.   }
25.
26.
27.   void second_function(int a)
28.   {
29.      int b = a;
30.   }
```

Potential Problem of Local Variables

- **Never be referenced after their stack frames are released.**

```
#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, _IOLBF, BUFSIZ) != 0)
        return(NULL);
    return(fp);     /* error */
}
```

See Ch. 5

Potential Problem of Stack Frames

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

int
main(void)
{
    f1();
    f2();
    _exit(0);
}

static void
f1(void)
{
    pid_t   pid;

    if ((pid = vfork()) < 0)
        err_sys("vfork error");
    /* child and parent both return */
}

static void
f2(void)
{
    char    buf[1000];      /* automatic variables */
    int     i;

    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}
```

Shared Library vs. Static Library

- **A program usually calls standard C/C++ library, POSIX functions, etc.**
- **The binary codes of the pre-compiled function call have to be *linked* with the program.**
- **Static library:**
  - the library and program are complied into one file.
  - In Linux, static library ends with .a.
- **Problems:**
  - When the library is updated, the program has to be re-linked.
  - The file size will be large. Potential for duplicating lots of common code in the executable files on a filesystem.

Shared Library

- ***Shared libraries*** **(dynamic link libraries, DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.**
- **Why a shared library?**
  - Remove common library routines from executable files.
  - Have an easy way for upgrading
- **Problems**
  - More overheads: dynamic linking
- **gcc:**
  - Searches for shared libraries first.
  - Static linking can be forced with the -static option to gcc to avoid the use of shared libraries.
- **Supported by many Unix systems**

```
$ cc -static hello1.c                    prevent gcc from using shared libraries
$ 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

$ cc hello1.c                            gcc defaults to use shared libraries
$ 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

- **Three ANSI C Functions:**
  - malloc – allocate a specified number of bytes of memory. Initial values are indeterminate.
  - calloc – allocate a specified number of objects of a specified size. The space is initialized to all 0 bits.
  - realloc – change the size of a previously allocated area. The initial value of increased space 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);
```

- Suitable alignments for any data object
- Generic void * pointer
- free(void *ptr) to release space to a pool.
- Leaking problem
  - allocated memory that program no longer references.
- Free already-freed blocks or blocks not from alloc().
- mallopt, mallinfo

```
#include <stdio.h>
#include <stdlib.h>
main() {
char *ptr;

    ptr = malloc(100);
    free(ptr);
    free(ptr);
    ptr[0] = 'a'; ptr[1]=0;
    printf("%s - Done\n", ptr);
}
```

Memory Allocation

- **Remark**
  - realloc() could trigger moving of data → avoid pointers to that area!
    - *prt* == NULL → malloc()
  - sbrk() is used to expand or contract the heap of a process – a malloc pool
  - Record-keeping info is also reserved for memory allocation – do not move data inside.
  - alloca() allocates space from the stack frame of the current function!
    - No needs for free with potential portability problems

Record-keeping Info.

- **Knowing how much to free**
  - Keep the length of a block in the word preceding the block.

- **Keeping track of free blocks**
  - Explicit list among the free blocks using pointers within the free blocks

Environment Variables

- **Name=value**
  - Interpretation is up to applications.
    - Setup automatically or manually
    - E.g., HOME, USER, MAILPATH, etc.

      setenv FONTPATH $X11R6HOME/lib/X11/fonts

#include <stdlib.h>

char *getenv(const char *name);

- ANSI C function, but no ANSI C environment variable.

Environment Variables

#include <stdlib.h>

int putenv(const char *name-value);

- Remove old definitions if they exist.

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

- rewrite = 0 → no change on existing values.
- Rewrite > 0 → overwriting existing values.

int unsetenv(const char *name);

- Remove any def of the *name*
- No error msg if no such def exists.

Environment Variables

- **Adding/Deleting/Modifying of Existing Strings**
  - Modifying
    - The size of a new value <= the size of the existing value → overwriting
    - Otherwise; malloc() & redirect the ptr
  - Adding
    - The first time we add a new name → malloc of pointer-list's room & update *envron*
    - Otherwise; copying. realloc() if needed.
  - The heap segment could be involved.

Nonlocal Jumps: setjmp & longjmp

- **Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location**
- **Objective:**
  - Goto to escape from a deeply nested function call!
    - break the procedure call/return discipline
  - Useful for error recovery and signal handling (Ch. 10)

- What if cmd_add() suffers a fatal error?
- How to return to main() to get the next line?

```
#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 */
}
```

Note: Automatic variables are allocated within the stack frames!

setjmp and longjmp

#include <setjmp.h>

int setjmp(jmp_buf env);

- Return 0 if called directly; otherwise, it could return a value *val* from longjmp().
- *env* tends to be a global variable.

int longjmp(jmp_buf env, int val);

- longjmp() unwinds the stack and affect some variables.

- Parameter: jmp_buf

  Definition is machine-dependent.

  Includes CPU registers, stack pointers and return address (Program Counter).

Example of setjmp() and longjmp()

```
#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 */
}
```

Effects on Variables

```
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);
}
```

```
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);
}
```

setjmp and longjmp

- Variables stored in memory have their values unchanged – no optimization…

```
[cshih@oris environ]$ ./testjmp
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
[cshih@oris environ]$ ./testjmp.opt
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

Effects on Automatic, Register, and Volatile Variables

- Compiler optimization
  - Automatic and register variables

    could be in CPU (for reducing access time) or memory
  - Volatile variables

    can be changed by something beyond the control of the program in which it appears, e.g., threads, signal handlers
- Values are often indeterminate
  - Normally no roll back on automatic and register variables
  - Variables in momory (e.g., global, static & volatile variables) are left alone when longjmp is executed.
- Portability Issues!

Potential Problems of Nonlocal Jumps

- **Works within stack discipline**
  - Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

P1()
{
  if (setjmp(env)) {
    /* Long Jump to here */
  }
  P2();
}

P2()
{  . . . P2(); . . . P3(); }

P3()
{
  longjmp(env, 1);
}
```

Potential Problems of Nonlocal Jumps

- **Works within stack discipline**
  - Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

P1()
{
  P2(); P3();
}

P2()
{
   if (setjmp(env)) {
    /* Long Jump to here */
  }
}

P3()
{
  longjmp(env, 1);
}
```

Memory-mapped I/O (Ch.14)

- mmap() maps a file on disk into a buffer in memory
- When we fetch bytes from the buffer, the corresponding bytes of the file are read.
- When we store data in the buffer, the corresponding bytes are written
- This lets us perform I/O without using read or write.

high address
stack
memory-mapped
portion of file
len
start addr
heap
uninitialized data
(bss)
initialized data
text
low address
file:
memory-mapped
portion of file
off
len

```
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

DESCRIPTION

The **mmap**() system call causes the pages starting at addr and continuing for at most len bytes to be mapped from the object described by fd, starting at byte offset offset. If len is not a multiple of the page-size, the mapped region may extend past the specified range. Any such extension beyond the end of the mapped object will be zero-filled.

If addr is non-zero, it is used as a hint to the system. (As a convenience to the system, the actual address of the region may differ from the address supplied.) If addr is zero, an address will be selected by the system. The actual starting address of the region is returned. A successful mmap deletes any previous mapping in the allocated address range.

The protections (region accessibility) are specified in the prot argument by or'ing the following values:

```
PROT_NONE   Pages may not be accessed.
PROT_READ   Pages may be read.
PROT_WRITE  Pages may be written.
PROT_EXEC   Pages may be executed.
```

The flags argument specifies the type of the mapped object, mapping options and whether modifications made to the mapped copy of the page are private to the process or are to be shared with other references. Sharing, mapping type and options are specified in the flags argument by

Copy a file using memory-mapped I/O

```
int
main(int argc, char *argv[])
{
    int         fdin, fdout;
    void        *src, *dst;
    struct stat statbuf;

    if (argc != 3)
        err_quit("usage: %s <fromfile> <tofile>", argv[0]);

    if ((fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);

    if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) < 0)
        err_sys("can't creat %s for writing", argv[2]);

    if (fstat(fdin, &statbuf) < 0)    /* need size of input file */
        err_sys("fstat error");

    /* set size of output file */
    if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
        err_sys("lseek error");
    if (write(fdout, "", 1) != 1)
        err_sys("write error");

    if ((src = mmap(0, statbuf.st_size, PROT_READ, MAP_SHARED,
      fdin, 0)) == MAP_FAILED)
        err_sys("mmap error for input");

    if ((dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
      MAP_SHARED, fdout, 0)) == MAP_FAILED)
        err_sys("mmap error for output");

    memcpy(dst, src, statbuf.st_size); /* does the file copy */
    exit(0);
}
```

What You Should Know

- How command line arguments are passed to a new process?
  - main()
- How a process is executed and terminates?
  - exit, _exit, _Exit, atexit()
- What the typical memory layout is?
  - Virtual Memory
  - Physical Memory
  - Volatile, register, and automatic variables
- Memory allocation
  - malloc, calloc, realloc, leakage problem
- How the process can use environment variables?
- What is nonlocal jump?
- Memory-mapped I/O