Introduction to UNIX System Programming

Page number :

Introduction to UNIX System Programming

Chapter 1 : Introduction

1.1. Man command

- an" finds and displays reference manual pages, including

  - utilities (or shell commands) : usually in section 1;
  - system calls : usually in section 2
  - and library calls : usually in section 3.

  - Example :

    - Use " man -s 2 write" to get the manual for system call write(2) /* note : In man page, (2) → section 2 */
    - But an write" return the 1st section write() function, which is write(1);

  - Note : some manuals are available in other section number such as ctime(3c).

- man -k keyword : list one-line summaries of all function calls that contain the given key word in the manuals (this may return many entries)

  - Example → libra% man -k fork

```
fork                        fork (2)        - create a new process
fork1                       fork (2)        - create a new process
pthread_atfork  pthread_atfork (3t)     - register fork handlers
vfork                       vfork (2)       - spawn new process in a
                                              virtual memory efficiently
```

- man page allows user to determine the specific of calling the function. Some important information are :

  - NAME /* all functions in this man page */
  - SYNOPSIS /* include files + function prototypes */

- DESCRIPTION /* describe the functions in details, including the usage of parameters */
- RETURN VALUES /* return values of function call */
- ERRORS /* error values and conditions */
- SEE ALSO /* related function calls */

- Example

    - libra% man -s 2 time

      SYNOPSIS

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

       time_t time(time_t *tloc);
      ```

    - To use this functions, you must

        - include the ".h" file in your program
        - understand the format of any structures or parameters by looking at the header file. All header files are in directory "/usr/include". If the header file is <sys/XXX>, then it is in directory "/usr/include/sys/"

    - example : in file time.h, time_t is defined as

      ```
      typedef  long    time_t;
      ```

1.2. System Calls and Library calls

- Function calls that can be invoked in programs.

- Library call : a function in a standard C library
    - Example : string functions, standard file I/O functions

- System call : a function call into kernel resident code

- The standard UNIX C library provides a C interface to each system call.
- Flow of control during a system call :

<u>User Mode</u>

1. Executes a system call open()
2. Library code for open. Executes a trap(n) instruction (code for open). This causes software interrupt → to kernel mode, start at specify kernel location

<u>Kernel mode</u>

3. Get the location of   pen” code in kernel, goto to the location of   pen”
4. kernel open code : it will access and modify kernel data structure. When done, switch back to user mode

<u>User mode</u>

5. Continue step 2 /* get the address from stack */
6. Return to the statement after open() calls at step 1

<u>1.3. Using system calls and library functions</u>

- In a system call, if there is parameter pointing to a data object, <u>you must allocate space</u> for the object and pass its address in the call, i.e. using &

  - Example : time_t time(time_t *tloc);
    To use the time(2) system call,

```
time_t t;
time(&t)              /* correct */

time_t *t_ptr;
time(t_ptr);          /* incorrect */
```

- In a library call, if there is parameter pointing to a data object, you may or may not need to allocate space for the object. Read the man page carefully. There are three possibilities :

  1. It does not allocate space
     → you need to allocate space, e.g. ctime_r(3)

  2. It allocates space statically
     → calls to the same function overwrite the previous value, e.g. ctime(3). You need to copy the data to the local object before making the same call again.

  3. It allocate space dynamically
     → every call return pointer to new object /* multithreaded safe function calls */

  - Example : char *ctime(const time_t *clock);

    Correct solution

```
time_t        t;
char timeStr[30], *now;

time(&t);
now = ctime(&t);
strcpy(timeStr, now);
        :
        :
time(&t)
now = ctime(&t)    /* note : now points to new time string */
printf("%s:%s",timeStr,now);
```

    Incorrect solution

```
time_t          t;
char *old, *now;

time(&t);
old = ctime(&t);
        :
        :
time(&t)
now = ctime(&t)    /* now and old point to the same string */
printf("%s:%s",timeStr,now);
```

1.4. Error Handling

- errono

  - Most system calls return a value -1 if an error occurs, or a value greater than or equal to zero if all is OK.

  - Every system call error sets the global (external) variable "errno" to a particular value. Note that "errno" contains valid data only if a system call actually returns -1.

- Perror(3)

  - The perror() allows you to display an informative error message in the event of an error.

  - perror(msg) : print   sg: XYZ" to the standard error channel, where XYZ is a description of the last system error.

  - Example :

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>

main()
```

```
{
        int fd;
        fd = open("bad.file",O_RDONLY);
        if (fd == -1)
                perror(    rror in read");
        else
        {       close(fd);
                exit(0);
        }
}
```

```
Sample run :
Error in read : No such file or directory
```

- strerror(3)

    - It returns a pointer to error string corresponding to its argument errno.

    - Example :

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
main() {
        int fd;
        fd = open("bad.file",O_RDONLY);
        if (fd == -1)
                fprintf(stderr,    rror : %s : read: line %d : %s\n",
                        __FILE__, __LINE__,strerror(errno));
        else { close(fd);
                exit(0);
        }
}
```

```
libra% a.out
Error : try.c : read: line 11 : No such file or directory
```

- Note : __FILE__, __LINE__, __DATE__, __TIME__ are the predefined macros including in all programs.

- Find out the cause of error

    - Sometime, you may want to know what cause the error.
      The RRORS" part of a specific system call man page
      lists all error numbers (names) with explanations

    - Look at <sys/errno.h> for the list of all errnos and corresponding representations.

    - Example : ERRORS in command an open"

      ERRORS
      The open() function fails if:

      EACCES Search permission is denied on a component of
      the path prefix, or the file exists and the
      permissions specified by oflag are denied, or
      …
      EDQUOT …

      EEXIST …
      EINTR …

- What do you do when you detect an error?

    - Use a default value (if you cannot get the value from function call)
    - Require user intervention
    - Write an error message (use perror(3) or strerror(3))
    - Write an error message and exit(2) with non-zero status
    - Combination of above depending of errorno

Chapter 2. Common C Library Functions (all in section 3)

2.1. String Functions

- #include <string.h>
- assume one or more characters (non null byte) terminated by a null byte
- note : it may not work on binary data
- here is the list of functions :

    - char * strcpy(char *dest, const char *src);
    - char * strcat(char *dest, const char *src);
    - size_t strlen(const char *s);

    - strcmp(); strdup(); strncpy(); strncat(); strncmp();

    - strpbrk(); strstr(); strchr();   /* parsing functions */

        - check for existing of specific character(s) or substring in the given string

        - example :
          strchr(ello",   ') → pointer to   " in string   ello"
          strpbrk(ello",   eo") → pointer to   " in string
          ello"
          strstr(ello",   eo") → null, no match string
          strsts(ello",   l") → pointer to   " in straing   ello"

    - strtok(3) :

        - parse a string into string tokens
        - initial call : input original string and a set of separator characters, return 1st argument
        - subsequent call : input NULL and a set of separator characters, return next argument (if no more argument, return NULL pointer)

- prototype :

```
#include <string.h>
char *strtok(char *s, const char *set);
```

- example :

```
char *pt, strbuffer[] =      am happy”;
cp = strtok(strbuffer, “ “);         /* cp →     \0”       */
cp = strtok(NULL, “ “);              /* cp →     m\0”      */
cp = strtok(NULL, “ “);              /* cp →     appy\0” */
cp = strtok(NULL, “ “);              /* cp == NULL      */
```

2.2. Memory Functions

- operate on chunks of memory
- more flexible than string functions
- able to handle binary data (including   0” character)
- may work on whole structure, i.e. copy a structure to another
- easy way to set a chunk of memory to all   0”
- void * memcpy(void *dest, const void *src, size_t n);
- void *memset(void *s, int c, size_t n);
- memmove(), memcmp(), memset()

2.3. Some string conversion routines

- #include <stlib.h>
- int atoi(const char *str);
- long atol(const char *str);
- double atof(const char *str);

2.4. Standard I/O

- here are some standard I/O functions operating on file:

  - #include <stdio.h>
  - fopen(); fclose();   /* open/close   */
  - fgetc(), fputc();   /* char-based I/O   */
  - fprintf(); fscanf();   /* formatted I/O   */
  - fgets(); fputs();   /* line-based I/O   */
  - fread(); fwrite();   /* buffer-based I/O*/
  - fseek(); ftell(); rewind();   /* repositioning file pointer */
  - ferror(); feof();   /* status functions */

  - Note : file pointer can be converted to file descriptor and vice versa, i.e. fileno() & fdopen()

2.5. Processing input arguments

- Run : program_name input1 input2 ... input_n

  In your main program : main (int argc, char **argv)

  ```
  argc = n+1
  argv[0] = "program_name"
  argv[1] = "input1"
          :
          :
  argv[n]= "input_n"
  ```

- Program option : each option is preceded by a - and is followed by additional character. An argument may be followed by option with a white space in between, e.g. cat -n filename

- Use getopt (3c) to process input arguments

- Prototype :

```
#include <stdlib.h>
int getopt(int argc, char * const *argv,  const  char  *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

argc, argv : from main()

optstring :

- contain the option letters the command using getopt() will recognize;
- if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which may be separated from it by white space.

optarg is set to point to the start of the option argument on return from getopt().

getopt() places in optind the argv index of the next argument to be processed.

ERROR in parsing :

- getopt() prints an error message on the standard error and returns a ``?" (question mark) when it encounters an option letter not included in optstring or no argument after an option that expects one.
- This error message may be disabled by setting opterr to 0.
- The value of the character that caused the error is in optopt.

When all options have been processed (that is, up to the first non-option argument), getopt() returns EOF.

- Example : Assume the usage of the following program
  "cmd [-a] [-b] [-o <file>] files..."

```
#include <stdlib.h>
```

```
#include <stdio.h>
main (int argc, char **argv)
{
        int c;
        extern char *optarg;
        extern int optind, opterr, optopt;
        int count = 0;
        char *ofile = NULL;

        opterr = 0;
        while ((c = getopt(argc, argv, "abo:")) != EOF)
        {
                count++;
                switch (c) {
                case 'a':
                        printf("get -a in position %d\n", count);
                        break;
                case 'b':
                        printf("get -b in position %d\n", count);
                        break;
                case 'o':
                        ofile = optarg;
                        printf("get -o in position %d and ofile = %s\n",
                        count, ofile);
                        count++;
                        break;
                case '?':
                        fprintf(stderr, "usage: cmd [-a] [-b] [-o <file>]
                        files...\n");
                        fprintf(stderr, "The error option symbol is -
                        %c,",optopt);
                        printf(" it is in position %d\n", count);
                        exit (2);
                }
        }
        for ( ; optind < argc; optind++)
                printf("%s\n", argv[optind]);
        return 0;
}
```

- Sample run 1:

```
libra% cmd -a -o fileXYZ -b -a File1 File2
get -a in position 1
get -o in position 2 and ofile = fileXYZ
get -b in position 4
get -a in position 5
File1
File2
```

- Sample run 2 :

```
libra% cmd -a -o fileXYZ -c -a File1 File2
get -a in position 1
get -o in position 2 and ofile = fileXYZ
usage: cmd [-a] [-b] [-o <file>] files...
The error option symbol is -c, it is in position 4
```

- Sample run 3 : set opterr = 1;

```
libra% cmd -a -o fileXYZ -c -a File1 File2
get -a in position 1
get -o in position 2 and ofile = fileXYZ
a.out: illegal option -- c
usage: cmd [-a] [-b] [-o <file>] files...
The error option symbol is -c, it is in position 4
```

Chapter 3. System Information Function Calls

3.1. Time

- time(2) : return number of seconds since 0:00:00 1/1/1970
- gettimeofday(3) : same as above + microseconds
- ctime(3) : change number of seconds to readable string

- times(2) : get process   user and system CPU times

- Example :

```
#include <sys/time.h>
int gettimeofday(struct timeval *tp, void * );

struct timeval {
      time_t  tv_sec;        /* seconds */
      long    tv_usec;       /* and microseconds */
};

---------------------------------------------------

#include <time.h>
char *ctime(const time_t *clock);
```

- Sample program :

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

main()
{
        struct timeval rt;
        gettimeofday(&rt, NULL);
        printf(   urrent time: %d secs %d usecs\n”,rt.tv_sec,
        rt.tv_usec);
        printf(   eadable time in secs : %s\n”, ctime(&rt.tv_sec);
}
```

sample run :

```
current time: 917395957 secs 235576 usecs
readable time in secs : Tue Jan 26 16:12:37 1999
```

- look at manual for other time related functions.

3.2. Machine information

- limit(1) : to obtain system resource limits, including CPU time, descriptors etc
- getrlimit(2) : system call interface of limit(1)
- setrlimit(2) : set system resource limits
- uname(2) : to obtain the OS name, hostname, OS version, etc.
- sysconf(2) : to obtain some system limits, e.g. number of CPU, page size, clock ticks per second
- pathconf(2): to obtain file related limits, e.g. max. length of file name, pipe buffer size
- getuid(2) : to get user ID
- getpwnam(3) : to get password entry by user login
- getpwuid(3) : to get password entry by UID
- password entry includes UID, GID, login name, user name, home directory & etc.

- Example :

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

main()
{
        struct utsname host;
        struct passwd *pptr;

        uname(&host);      /* note : should have error checking */
        printf(    ostname is : %s\n”, host.nodename);

        printf(    age size : %d bytes\n”,sysconf(_SC_PAGESIZE));

        pptr = getpwnam(    wong”);
        printf(    our UID is : %d\n”, pptr->pw_uid);
        printf("Your home directory is : %s\n", pptr->pw_dir);
}
```

sample run

```
hostname is : libra
Page size : 4096 bytes
Your UID is : 319
Your home directory is : /afs/sfsu.edu/f1/jwong
```

Chapter 4. File I/O and File/Directory Information

4.1. Introduction

- Regular files (ordinary files) contain a stream of bytes. It can be text, executable programs, or other data.
- To access a file, you must first obtain a file descriptor, which is a small integer used to identify a file that has been opened for I/O.
- Most systems provide more than 20 file descriptors per process (0, 1,2, .., max-number -1).
- UNIX programs associate file descriptors 0, 1, and 2 with the standard input, standard output and standard error respectively.
- File descriptors are assigned by the kernel when the following system calls are successful, including open(2), creat(2), dup(2), pipe(2) and fcntl(2).
- Every file on the disk has a special fixed-size housekeeping structure called an inode (index node) associated with it.
- An inode contains the following information such as file owner, file type, permission flags, last modified date, last accessed date, file size and locations of file (in blocks).
- Directory files contain the names and locations of other files (only accessible via system calls)
- Hard link is a directory entry that contains the filename and its inode number.
- A single file in UNIX may have more than one name. You may use the UNIX Command ln(1) or link(2) to create a hard link to a file.
- Symbolic Link is a special file that contains a path name as its data. Use n -s" or symlink(2) to make symbolic links.

- Note : you cannot create a hard link to another logical filesystem, but you can create symbolic link across the filesystem boundaries. Also, you cannot create a hard link to a directory

4.2. File Access Permission

- Every process has a unique PID. (in the range 0 -> 30000)
- use getpid(2) system call to get PID.
- PID = 0 (the swapper or scheduler), PID = 1 (init) and PID = 2 (page daemon)
- Every process has a parent process (except 0), and a corresponding PPID. Use getppid(2) system call to get PPID.

- Each user is assigned a positive integer Real User ID (UID) and a positive integer Real Group ID (GID)
- Each file in the system is associated with a UID and a GID
- The file /etc/passwd maintains the mapping between login names and UIDs
- The file /etc/group maintains the mapping between group names and GIDs

- Each process also has a UID and a GID (from the program file)
- Get the UID and GID of the process by calling the getuid(2) and getgid(2) system call

- Each process is also assigned a positive integer Effective User ID (EUID) and a positive integer Effective Group ID (EGID). Use system call geteuid(2)and getegid(2) to get EUID.
- Normally, EUID and EGID values are the same as UID and GUID.
- (However, if a process execute the program with setuid bit on, the process EUID and EGID becomes UID and GID of file owner)

- chown(2) changes the owner and group of the indicated file to the specified owner and group.

- Setuid using chmod(1) :

```
chmod 4755 a.out          /* set uid bit */
-rwsr-xr-x   1 cswong   diagnost    7744 Jan 29 19:58 a.out

chmod 2755 a.out          /* set gid bit */
-rwxr-sr-x   1 cswong   diagnost    7744 Jan 29 19:58 a.out
```

- To determine if a process can access a file :

  1. If the EUID = 0 (super user), then OK
  2. If the EUID = UID (owner) then OK (check 3 permission bits for User)
  3. If the EGID = GID , then OK (check 3 permission bits for Group)
  4. If not EUID=UID and not EGID=GID, check 3 permission bits for Other.

- Note : So a process will have PID, PGID, EUID, EGID, UID and PGID and a file will have UID and GID

- Example :

  - when you run passwd command, you want to change an entry in passwd file.
  - The owner of the password file is the root and only the root may write (change) the password file.
  - Now in order to change the password file, the EUID of a process must be root.
  - This can be done by setting the set-user-id bit in passwd program (the file) using setuid.
  - The process (initially EUID = your UID) executes passwd program, the EUID becomes root, so it may change the passwd file.

<u>4.3. File I/O system calls</u>

- stat(2) : retrieve information from inode
- lstat(2) : same as stat(2) but does not follow symbolic links, i.e. it return information of the link itself.
- fstat(2) : work on file descriptor

- Prototypes :

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

int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
```

- The contents of the structure pointed to by buf include the following members:

```
mode_t     st_mode;    /* File mode */
ino_t      st_ino;     /* Inode number */
nlink_t    st_nlink;   /* Number of links */
uid_t      st_uid;     /* User ID of the file's owner */
gid_t      st_gid;     /* Group ID of the file's group */
off_t      st_size;    /* File size in bytes */
time_t     st_atime;   /* Time of last access */
time_t     st_mtime;   /* Time of last data modification */
```

- You can use macros to extract fields from the st_mode field. <sys/stat.h> provides many macros.

- Example : Check file type

```
struct stat ibuf;
stat(    yfile”, &ibuf);
if (S_ISDIR(ibuf.st_mode)) printf(    t is a directory\n”);
```

- Example : Change file mode chmod(2)

```
struct stat ibuf;
stat(    yfile”, &ibuf);

ibuf.st_mode |= S_IWUSR | S_IWGRP;
if (chmod(    yfile”, ibuf.st_mode) == -1)
{ perror(    hmod”); }
```

- Useful macros in <sys/stat.h>

```
#define S_ISUID 0x800          /* set user id on execution */
#define S_ISGID   0x400        /* set group id on execution */

#define S_IRWXU 00700          /* read, write, execute: owner */
#define S_IRUSR 00400          /* read permission: owner */
#define S_IWUSR 00200          /* write permission: owner */
#define S_IXUSR 00100          /* execute permission: owner */

#define S_IRWXG 00070          /* read, write, execute: group */
#define S_IRGRP 00040          /* read permission: group */
#define S_IWGRP 00020          /* write permission: group */
#define S_IXGRP 00010          /* execute permission: group */

#define S_IRWXO 00007          /* read, write, execute: other */
#define S_IROTH 00004          /* read permission: other */
#define S_IWOTH 00002          /* write permission: other */
#define S_IXOTH 00001          /* execute permission: other */

#define S_ISFIFO(mode)  (((mode)&0xF000) == 0x1000)
#define S_ISDIR(mode)   (((mode)&0xF000) == 0x4000)
#define S_ISREG(mode)   (((mode)&0xF000) == 0x8000)
#define S_ISLNK(mode)   (((mode)&0xF000) == 0xa000)
#define S_ISSOCK(mode)  (((mode)&0xF000) == 0xc000)
```

- open(2) : Opens the file for reading and/or writing. Returns a file descriptor.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, /* mode_t mode */ ...);
```

Note : define mode if O_CREAT in oflag

- Useful macro in <fcntl.h>

```
#define O_CREAT         /* open with file create (uses third arg) */
#define O_TRUNC         /* open with truncation */
#define O_EXCL          /* exclusive open */

#define O_RDONLY
#define O_WRONLY
#define O_RDWR

#define O_NDELAY        /* non-blocking I/O */
#define O_APPEND        /* append (writes guaranteed at the end) */
#define O_NONBLOCK /* non-blocking I/O (POSIX) */
```

- read(2) : reads characters
- write(2) : writes characters

- lseek(2) : moves the current position of any location
- close(2) : closes a file descriptor.

- unlink(2) : unlink a file.
- umask(2) : set the file creation mask → this affect the creating mode of files

- dup(2) : duplicate the file descriptor (OS find the lowest available file descriptor, copy the file descriptor, i.e. pointing to same location)
- dup2(2) : allow user to specify the new file descriptor ID

- fcntl(2) : supports a set of miscellaneous operations for open files, e.g. duplicates file descriptor, returns the file status flags, set file mode or flags eand etc

- fsync(2) :   flushes all dirty buffers onto disks.
- tmpnam(3) : allocate temporary file

- Example : To write standard output to a file.

```
int fd;

fd=open(    yfile", O_CREAT | OWONLY | O_TRUNC, 0660);

if (fd == -1) { perror(    pen"; exit(1); }

close(1); /* here close the standard output to screen */

if (dup(fd) != 1)  /* if OK, fd and 1 point to myfile */
{
        printf(    up failed to return 1\n"); exit(1);
}

close (fd);
printf(    ello world goto myfile!!\n");
```

4.4. Directory

- getcwd(3) : get the current working directory. See also pwd(1)
- chdir(2) : change the current directory. See also cd(1)

- mknod(2) : makes a file of any type. (e.g. (e.g. device files.)
- mount(2) : mounts the file system onto the specified directory.
- umount(2) : unmount the file system

- mkdir(2) : create a directory
- rmdir(2) : remove a directory
- rename(2) : rename file/directory

- symlink(2) : create symbolic link
- link(2) : create hard link

- Note : open(2) and read(2) cannot read the symbolic link (but read the file). To read symbolic link (i.e. the filename), use readlink(2) → use lstat to check the file and to get the size of the name in symbolic link

- To access entries in a directory :

```
#include <direct.h>

typedef struct dirent {
        char               d_name[1];  /* name of file */

        /*  optional fields */
        ino_t              d_ino;        /* "inode number" of entry */
        off_t              d_off;      /* offset of disk directory entry */
        unsigned short     d_reclen;    /* length of this record */
} dirent_t;
```

  Note : dname[] is variable size, you can use strlen(3) to find out the size.

- opendir(3C); readdir(3C); closedir(3C);
- telldir(3C); seekdir(3C); rewindir(3C)

Chapter 5. Process Information

5.1 Introduction

- A program is an executable file
- The only way a program can be executed is by issuing the exec() system call
- A process is a program in a state of execution.
- The only way a new process can be created is issuing the fork(2) system call

- A process has
  - text (or executable code - usually read only)
  - data
  - stack
  - process management information, including PID, open files, etc.

- Example : Program vs Process virtual address space

```
int a;

main()
{
        int b;
        static int c;

        b = a * a;
        c = hello(b);
}

int hello( int d)
{
        int e;

        e = d + 2;
        return(e);
}
```

- Process Virtual Address Space

```
                |                              |
stack           |      main()                  |
                |      b = ?                   |
                |______________________________|
                |                              |
                |      hello()                 |
                |      d = ?  e = ?            |
                |------------------------------|
                |      :            :          |
                |      :            :          |
                |------------------------------|
                |      Heap : malloc()         |
                |------------------------------|
data            |      a = ?  c = ?            |
                |------------------------------|
                |      main()  { ... }         |
text            |      hello()  { … }          |
                |______________________________|
```

- A process may

  duplicate (fork(2))
  replace itself with another program (exec())
  terminate (exit(3C))
  execute other system calls (open(2), close(2), etc)
  communicate with other processes (signal, pipe, IPC etc)

- system calls

getpid(2)	: get process ID
getppid(2)	: get parent process ID
getenv(3)	: get process environment information, e.g. $HOME, $PATH
putenv(3)	: set or add environment variables
env(1)	: get environment information

<u>5.2. Process Control</u>

- Process Creation

- easy method → system(3)
- create sh process to execute the command line as given in “(“ “)”.
- system(3) returns when the sh has completed.

- Example :

```
int ind;

if ((ind = system(    ho > tempFile”)) == -1)
{
        printf(    rror in system\n”);
}

/* you can manipulate tempFile here !! */
```

- fork(2) : fork a new child process (with different PID)
- fork1(2) : same as fork(2), but duplicate calling thread only
- vfork(2) : same as fork(2), but do not duplicate the whole virtual memory, expect to exec() soon.

- Example :

```
int ind;
ind = fork();
if (ind == -1) {
        perror(    rror, fork”);
} else if (ind == 0) {
        printf(    hild process ID : %d\n”, getpid());
        printf(    y parent  PID : %d\n”, getppid());
        /* usually, exec() here */
        exit(0);
} else {
        printf(    arent ID : %d\n”, getpid());
        exit(0);
}
```

- Sample run :

```
parent ID : 4829
child process ID : 4830
```

my parent PID : 4829

- Process Transmutation (executing a new program)

  - A process may replace itself with another program and retain the same PID by using exec() system call

  - There are several exec() system calls available : execl(2), execv(2), execle(2), execve(2), execlp(2), execvp(2)

    - Some take list of strings as parameters
    - Some take an array which stored arguments
    - Some search for PATH to find program
    - etc

  - Example : Prototype

    ```
    #include <unistd.h>
    int execlp (const char *file, const char *arg0, ..., const char
    *argn, char * /*NULL*/);
    ```

  - From above example :

    ```
                :       :
            } else if (ind == 0) {

                x = execlp(    yProg”,     yProg”, “-c”,     tr1”,
                                                   “-F”, (char *) 0);
                if (x == -1)
                {
                        perror(    rror : execlp”);
                }
                exit(0);
            } else {
                :       :
    ```

  - The file descriptors of the original process are kept during the exec()

- Example : Program simulates "date > temp"

```
main()
{
        int fd;
        fd = open("temp"....);
        dup2(fd,1) /* copy stdout to fd */
        close(fd);
        execl("date"...);
}
```

- Process Termination

  - The process exits by calling exit(3C), i.e. exit(intcode);

  - intcode is an integer that is returned to the parent for examination

  - if the parent is a C-shell, this value is available in $status (do cho $status") ; otherwise, the value is available in parent process, i.e use wait(2).

  - The return code is stored in the 2nd rightmost byte of the parameter.

  - Note : usually, use exit(0) to indicate successful termination; look at <stdlib.h>

    ```
    #define EXIT_FAILURE    1
    #define EXIT_SUCCESS    0
    ```

  - atexit(3c) : allow user to register a clean-up function. When exit(3C) is executed, the clean-up function will be automatically called.

- Parent cleanup

- After the process executes exit(), it becomes a "zombie", waiting for its parent to accept the return code via wait(2) or waitpid(2). When the return status is accepted the process finally goes away.

- If the parent does not wait for its child, the child will be adopted by the init process (PID = 1)

- Example :

```
main()
{
        int i, PID, retcode;

        for (i=1; i<=2; i++)
        {
                if (fork()==0)
                {
                  printf(" I am a child with PID %d and will return
        %d\n",
                                        getpid(), i);
                  exit(i);
                 }
        }

        PID = wait(&retcode);
        printf("A child (PID %d) exited with %d\n",PID, retcode >> 8);
        PID = wait(&retcode);
        printf("A child (PID %d) exited with %d\n",PID, retcode >> 8);
}
```

- Sample Run :

```
I am a child with PID 5862 and will return 1
I am a child with PID 5863 and will return 2
A child (PID 5863) exited with 2
A child (PID 5862) exited with 1
```

- <wait.h> defines some useful macros (look at wstat(5))

WIFEXITED(stat)	: True if child process exit(3C).
WEXITSTATUS(stat)	: return exit(3C) value
WIFSIGNALED(stat)	: True, if child process is terminated by a signal
WTERMSIG(stat)	: return signal number
WIFSTOPPED(stat)	: True, if child process is stopped by a signal
WSTOPSIG(stat)	: return signal number.

- Note : you can replace few lines from above sample program by

```
#include <wait.h>

if (WIFEXITED(retcode))
{
      printf("A child (PID %d) exited with %d\n",PID,
                        WEXITSTATUS(retcode));
}
```

Chapter 6. Introduction UNIX IPC

6.1. Introduction

- In single process programming, different modules within the single process can communicate with each other using global variables, function calls, etc.

- For multiprogramming operating systems, two processes may want to synchronize and communicate with each other.

- The operating system must provide some facilities for the Interprocess Communication (IPC).

- We consider several different methods of IPC in UNIX :

<u>work within the same host</u>	
exit status	(in previous chapter)
file and record locking	
pipes	
FIFOs (named pipes)	
mmap	
signals	(in another chapter)
message queues	(in another chapter)
semaphores	(in another chapter)
shared memory	(in another chapter)
<u>work within the same host and network</u>	
sockets	(in another chapter)
TLI	(in another chapter)

6.2. File and Record Locking

- Multiple processes want to share some resource
- Some form of Mutual Exclusion must be provided so that only one process at a time can access the resource.

- There are several system calls may be used to enforce the mutual exclusion : link(2), creat(2), open(2) /* slow ways */

- Advisory lock : other processes may still write to the file, i.e. other processes may still open() the file and modify the data.
- Mandatory lock : other processes may not read from or write to the file.

- File locking : locks an entire file
- Record (range) locking : locks the specified range of the file

- lockf(3C) : mandatory file and record locking
- flock(3B) : advisory file locking

6.3. Anonymous Pipes

- Allow transfer of data between processes in a FIFO manner

- One way flow of data (usually, about 4k bytes)
  Note : now, UNIX allows bi-directional flow of data

- It is created by the pipe(2) system call.
  Prototype : int pipe (int *fd)

  fd is an array of 2 file descriptors (in fd table), fd[0] is for reading, fd[1] is for writing
  Note : now, both fd[0] and fd[1] can be used for both reading and writing

- Use system calls read(2) and write(2) to access the pipe.

- Read :
  if the pipe is empty, the process is blocked (unless the writing fd is closed → read 0 byte); otherwise, read up to the number of specified bytes.

- Write :

    - If the pipe is full, the process is blocked until enough spaces are available to write the entire data. If no reading fd attached, return a signal IGPIPE”

    - If the pipe is not full and the size of write is less than 4k, then the write is atomic. If the size is > 4k, you should separate into several writes.

- You may use fcntl(2) to set O_NDELAY flag on fd, so that the process will not block in read or write.

- Example : /* should have error checking */

```
#include <stdio.h>

main()
{
        int fd[2];
        char ch[11];

        pipe(fd); /* creates a pipe */

        if (fork() == 0) {
                read(fd[0],ch,11);
                printf("Child got %s\n",ch);
                read(fd[0],ch,11);
                printf("Child got %s\n",ch);
                exit(0);
        } else {
                write(fd[1],"hello world",11);
                sleep(10);
                write(fd[1],"next word",11);
                wait(0);
                close(fd[0]);
                close(fd[1]);
        }
}
```

Example 2 : simulate date | cat

```
main()
{
        int fd[2];
        pipe(fd);
        if (fork() == 0)
        {
                close(fd[0]);
                dup2(fd[1],1) /* duplicate write end of pipe to stdout */
                close(fd[1]);
                execl ("/bin/date","date",0);
        }
        else
        {
                close(fd[1]);
                dup2(fd[0],0) /* duplicate write end of pipe to stdin */
                close(fd[0]);
                execl ("/bin/cat","cat",0);
        }
}
```

<u>6.4. FIFOs</u>

- Named pipe has a name in the file system

- Allows independent processes to communicate.

- Create a named pipe by using
    - mknod(2)
    - mknod(1)
    - mkfifo(1)

    - Example : mknod("NamedPipe1",S_IFIFO | 0666,0);
    - Note : S_IFIFO (make FIFO)

- After this, you have the NamedPipe1 in your directory, any number of processes may use open(2) system call to access the NamedPipe1

- Use read(2) and write(2) to access the data
- Use rm(1) or unlink(2) to remove the NamePipe1.

- For pipes and FIFOs, the data is a stream of bytes
  - Reads and writes do not examine the data at all
  - When the data consists of variable-length messages, we may want to know where the message boundaries are so that a single message may be read.
  - you may read or write a structure.

6.5. Mmap

- mmap(2) : allows user to map a file into process address space. After mapping, the process may access the data using pointer instead of read(2) and write(2).

- To use the mmap(2), you need to open(2) the file, then use the file descriptor of the opened file to map the file. Once the file is mapped, you can close(2) the file.

- munmap : to unmap the file

- Example :

```
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <sys/mman.h>
    #include <stdlib.h>
    #include <fcntl.h>
    #include <stdio.h>

    int main()
    {
            int fd;
            struct stat st;
            caddr_t base, ptr;

            /* open the file */
```

```
        if ((fd = open("myFile", O_RDONLY, 0)) < 0) {
                perror("cannot open my file");
                exit(1);
        }

        /* determine the size of the file */
        fstat(fd, &st);
        printf("File size: %d\n",st.st_size);

        /* map the entire file into memory      */
        /* 0 : any memory location              */
        /* st.st_size : size of a file                  */
        /* PROT_READ : "Read" access map pages  */
        /* MAP_SHARED : may shared among other processes */
        /* fd : opened file descriptor          */
        /* 0 : starting position of mapping     */

        base = mmap(0, st.st_size, PROT_READ, MAP_SHARED, fd, 0);

        if (base == MAP_FAILED) {
                perror("map failed");
                close(fd);
                exit(1);
        }

        /* no longer need the fd */
        close(fd);

        /* print the file */
        printf("data : |%s|\n", base);

        /* unmap the file */
        munmap(base, st.st_size);

        exit(0);
}
```

- Sample run :

```
libra% cat myFile
Hello World

libra% a.out
File size: 12
data : |Hello World
|
libra%
```

- You need to specify the size of mapping,
  if it is more than the file size you will not get an error.
  You will get SIGBUS when you are trying to access the locations > file size.

- Solution : You can use truncate(3C) or ftruncate(3C) to extend the file size before mapping (must be opened for writing).

- Example : from previous example, you can extend myFile to 4K size as below :

```
/* open the file */
if ((fd = open("myFile", O_RDWR, 0)) < 0) {
        perror("cannot open my file");
        exit(1);
}

/* determine the size of the file before ftruncate(3C)*/
fstat(fd, &st);
printf("File size: %d\n",st.st_size);

if (ftruncate(fd, (off_t) 4096) == -1) {
        perror("ftruncate error!");
        exit(1);
}


/* determine the size of the file after ftruncated*/
fstat(fd, &st);
printf("File size: %d\n",st.st_size);
```

- Sample run :

```
libra% a.out
File size: 12
File size: 4096
data : |Hello World
|
```

- use msync(2) to flush data into disk.

Chapter 7. System V IPC and Sockets

7.1. Introduction

- System V IPC :

message queues
shared memory
semaphores

- ipcs(1) : list all current IPC objects
- ipcrm(1) : use to remove IPC object

- key_t key : use key value to specific IPC object during create time

  key = IPC_PRIVATE
  key = some integer number →
  all processes know the number can access the same IPC object
  key = ftok(3) →
  input a pathname to ftok(3), kernel will return same key for
  all processes using the same pathname

- id : IPC object returns the identifier to refer to the specific IPC object

- Summary of System V IPC (all system calls are from section 2)

	Message Queue	Semaphore	Shared Memory
include files	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
create or open	msgget	semget	shmget
control operations	msgctl	semctl	shmctl
IPC operations	msgsnd msgrcv	semop	shmat smhdt

- The get system calls create or open an IPC channel, all take a key value and return an integer identifier.

- The get system calls also take a flag argument.

- The rules for whether a new IPC channel is created or whether an existing one is referenced are :

  #include <sys/ipc.h>

  - key = IPC_PRIVATE (= 0) → creates a new IPC channel

  - key != 0 and flag = IPC_CREAT | 0640 → creates a new IPC channel if the key does not already exist. If an existing entry is found, that entry is returned (if the access permission is OK.)

  - key != 0 and flag = IPC_CREAT | IPC_EXCL | 0640 → creates a new IPC channel if the key does not already exist. If an existing entry is found, an error occurs.

7.2. Message Queues

- Message queues can be viewed as pipes with message-based (instead of stream based)

- The kernel maintains a structure for every message queue in the system. It includes the following information (use msgctl(2) to access the info.)

  - operation permission,
  - pointers to first and last message
  - number of bytes and number of messages on queue
  - maximum number of bytes allowed
  - pids of last sender and last receiver
  - times of last sender and last receiver
  - time of last msgctl

- Each message on a queue has the following attributes :

  #include <sys/msg.h>

  struct msgbuf {

```
        long    mtype;          /* message type */
        char    mtext[1];       /* message text : can be variable length*/
    };
```

- System calls

```
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>

    int msgget(key_t key, int getFlag);

    /* send a message into queue */
    int msgsnd(int id, const void *msgptr, size_t msgsz, int msgflag);

        id       : id of message queue (returning from get)
        msgptr   : pointer to msgbuf
        msgsz    : size of mtext (not including mtype)
        msgflag  : 0 - blocks (when full) or IPC_NOWAIT

    /* get a message from the queue */
    int msgrcv(int id, void *msgptr, size_t msgsz,
                                    long msgtype, int msgflag);

        msgtype  : type of message
        msgflag  : 0 - blocks when empty or specific type
                       message is not available
```

- Every message is stamped with a "type" (long), and receiving processes can restrict incoming messages to those of a specified type.

- When a process wants to receive a message from the queue, it must specify the "msgtype"

    - If "msgtype" = 0, first message on the queue is returned (oldest message)

    - If "msgtype" > 0, the first message with a type equal to "msgtype" is returned.

    - If "msgtype" < 0, the first message with the lowest type that is less than or equal to the absolute value of "msgtype" is returned.

- msgctl(2) is used to remove the object and get and set the control information.

- IMPORTANT : Once created, a message queue exists until it is explicitly removed by creator or superuser. (use magctl(2) or ipcrm(1))

Example :

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

```
typedef struct {
        long mesg_type;
        char mesg_data[1000];
} Mesg;

main()          /* writer */
{
        int msgid;
        int len;
        Mesg mesg;

        msgid = msgget(1000, 0600|IPC_CREAT);
        mesg.mesg_type = 100;
        strcpy(mesg.mesg_data, "message number 1111\0");
        len = strlen(mesg.mesg_data);
        msgsnd(msgid, (char *) &(mesg), len, 0);

        mesg.mesg_type = 300;
        strcpy(mesg.mesg_data, "message number 3333\0");
        len = strlen(mesg.mesg_data);
        msgsnd(msgid, (char *) &(mesg), len, 0);

        mesg.mesg_type = 200;
        strcpy(mesg.mesg_data, "message number 2222\0");
        len = strlen(mesg.mesg_data);
        msgsnd(msgid, (char *) &(mesg), len, 0);
}
```

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

typedef struct {
        long mesg_type;
        char mesg_data[1000];
} Mesg;

main()        /* Reader */
{
        int msgid;
        int len;
        Mesg mesg;

        msgid = msgget(1000, 0600|IPC_CREAT);
        msgrcv(msgid, (char *) &(mesg), 100, 0, 0);
        printf("Get a message type : %d  = %s\n",mesg.mesg_type,
        mesg.mesg_data);

        msgrcv(msgid, (char *) &(mesg), 100, 300, 0);
        printf("Get a message type : %d  = %s\n",mesg.mesg_type,
        mesg.mesg_data);

        msgrcv(msgid, (char *) &(mesg), 100, -300, 0);
        printf("Get a message type : %d  = %s\n",mesg.mesg_type,
        mesg.mesg_data);

        msgctl(msgid,IPC_RMID,0);
}
```

Sample run :

```
libra% ipcs

IPC status from <running system> as of Wed Jan 27 21:05:00 1999
Message Queue facility not in system.
T       ID     KEY       MODE       OWNER    GROUP
```

```
Shared Memory:
m        0   0x50000d9f --rw-r--r--     root     root
Semaphores:

libra% reader &

libra% ipcs

IPC status from <running system> as of Wed Jan 27 21:05:13 1999
T        ID      KEY        MODE        OWNER    GROUP
Message Queues:
q        0   0x000003e8 -Rrw-------    jwong       f1
Shared Memory:
m        0   0x50000d9f --rw-r--r--     root     root
Semaphores:

Note :          Key  = 1000 = 0x000003e8 (base 16)
                ID = 0;

libra% writer &
[2] 24454

← note messages from reader →
Get a message type : 100  = message number 1111
Get a message type : 300  = message number 3333
Get a message type : 200  = message number 2222
```

7.3. Semaphores

- Semaphores are synchronization primitive (They are not used for exchanging large amounts of data, as are pipes, FIFOs and message queues)

- The kernel maintains a structure for every set of semaphores in the system. It include the following information :

    operation permission,

pointers to first semaphore in the set
number of semaphores in the set
time of last semop
time of last change

- Each semaphore of a semaphore set has the following attributes :

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

struct sem {
       ushort  semval;        /* semaphore value */
       pid_t   sempid;        /* pid of last operation */
       ushort  semncnt;       /* # awaiting semval > cval */
       ushort  semzcnt;       /* # awaiting semval = 0 */
};
```

- System calls

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

  - A new <u>set of semaphores</u> is created, or an existing set of semaphores is accessed with the semget(2) system call.
  - semget(2) creates n semaphores, the number is from 0,1,..n-1. (Initially. semval = 0)

  - Using semop(2), a process can increase semval or decrease semval of semaphores.
  - It can also wait until the semval reaches or exceeds some minimum value, or until it becomes 0.
  - Semaphore operations can be set by using the following structure :

```
struct sembuf {
       ushort_t    sem_num;      /* semaphore # */
       short       sem_op;       /* semaphore operation */
       short       sem_flg;      /* operation flags */
};

where : /* read manual about the sem_flg */
```

- /* increase sem_value */
  sem_op > 0, sem_value = sem_value + sem_op

- /* wait until sem_value become 0 */
  sem_op = 0, wait until sem_value = 0

- /* decrease sem_value, may be block */
  sem_op < 0, wait until sem_value >= |sem_op|
  set sem_value = sem_value + sem_op

- Using semctl(2), a process can obtain and modify status information about the semaphore. You may fetch and set the semaphore value of a particular semaphore in the set.

- Example :

```
op_lock[2] = { 0,0,0        /* wait until sem#0 to become 0 */
               0,-1,0       /* then increment the value by 1 */ }

op_unlock[1] = { 0,1,IPC_NOWAIT   /* set sem#0 value = 1 */ }


mylock()
{
        semid = semget(KEY,1,IPC_CREAT);
        semop(semid, &op_lock[0],2);
}


my_unlock()
{
        semop(semid. &op_unlock[0],1);
}
```

Note : (1) Need to remove the semaphore when all processes are done
(2) If a process aborts for some reason while it has the lock, the semaphore value is left at one.

7.4. Shared Memory

- Shared memory is implemented in a very similar way to messages.
- Every shared memory segment in the system has an associated shared memory identifier (shmid)

- The kernel maintains a structure for every shared memory segment in the system. It contains the following information :

  operation permission
  segment size
  pid of last operation

number of processes attached
others

- System calls

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- shmget(2) creates a shared memory segment, but does not provide access to the segment for the calling process.

- Using shmat(2), a process can attach the shared memory segment. This system call returns the starting address (pointer) of the shared memory segment (similar to mmap(2))

- When a process is finished with a shared memory segment,it detaches the segment by calling shmdt(2) system call.

- Again, once the shared memory segment is created, it exists until it is explicitly removed by creator or superuser. (use shmctl(2) or ipcrm(1))

- Example :

Process1 : /* write ello", read orld" */

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

typedef struct {
        char word[100];
} String;

main()
```

```
{
        int shmid;
        String *pointer;

        shmid = shmget(1000, sizeof(String), 0600|IPC_CREAT);
        pointer= (String *) shmat(shmid, 0, 0);
        strcpy(pointer->word,"hello\0");
        sleep(3);
        if ((strcmp(pointer->word,"world\0")) == 0)
        {
                printf("process 1 read world\n");
                shmdt(pointer);
        }
        shmctl(shmid, IPC_RMID,0);
}
```

Process2 : /* read ello”, write orld” */ Only partial program

```
        shmid = shmget(1000, sizeof(String), 0600|IPC_CREAT);
        pointer= (String *) shmat(shmid, 0, 0);

        if ((strcmp(pointer->word,"hello\0")) == 0)
        {
                printf("process 2 read hello\n");
                strcpy(pointer->word,"world\0");
        }

        shmdt(pointer);
```

7.5. Sockets (Introduction)

- Sockets are a flexible general communication facility supported by Berkeley UNIX

- Client-server Model :

    - A server is a process that is waiting to be contacted by a client process so that the server can do something for the client. (Typically, 1 server and at least 1 client)
    - Note : A process may be both clients and servers

- Note : In this chapter, we only cover some basic Socket IPCs

- There are several domains across which sockets can operate, including :

    - AF_UNIX : UNIX domain → among UNIX processes in the same machine, take filesystem pathname

    - AF_INET : Internet Domain → among processes in Internet, take internet address

- There are several types of message passing in each domain, including :

    - SOCK_STREAM : Connection oriented, two way, reliable, variable length messages, expensive. It uses transmission control protocol (TCP).

    - SOCK_DGRAM : Connetionless, not reliable, fixed length (up to 8k), more effieceint. It uses datagram protocol (UDP)

- We only consider the following model in this chapter:

    - AF_UNIX, SOCK_STREAM
    - AF_INET, SOCK_STREAM

- **Typical sequences of steps to form a socket connection for SOCK_STREAM:**

    <u>Part I of Server :</u>

    1. Create an unnamed socket using socket(3XN)
    2. Attach the socket to a name in the file system using bind(3XN)
    3. Set the maximum number of pending connection using listen(3XN)
    4. Accept connection to the socket using accept(3XN)

    ….. wait until connection from client …..

    <u>Part I of Client :</u>

1. Create an unnamed socket using socket(3XN)
2. Try to connect to the named server socket using connect(3XN)
….. Connection Establishment ……

Part II of Server & Client :

Use the descriptors returned by accept(3XN) in server and socket(3XN) in client, to communicate using read(2) and write(2) (or recv()/send() or sendto()/recvfrom())

Note : typically, server fork() a new child process or thread to handle the client request. The parent process can continue to accept() more requests.

<u>End of communication :</u>

need to close all the descriptor by using close(2)
server needs to unlink the named socket by using unlink(2)

- System calls :

  #include <sys/types.h>
  #include <sys/socket.h>

  - int socket(int family, int type, int protocol)

    - creates an end point for communication.
    - returns a small integer value, called socket descriptor (sockfd).
    - usually protocol = 0 →the default protocol is used.

  - bind(int sockfd, struct sockaddr *name, int namelen)

    - bind unnamed socket to specific name.

    - For AF_UNIX :
      struct sockaddr {

```
        sa_family_t    sa_family;     /* address family */
        char           sa_data[14];   /* up to 14 bytes of
    direct address */
    };
```

- For AF_INET :
  ```
  struct sockaddr_in {
          sa_family_t         sin_family;
          in_port_t           sin_port;
          struct              in_addr sin_addr;
          char                sin_zero[8];
  };
  ```
  - see <netinet/in.h>
  - assign a port number to unamed socket,
  - cast it to sockaddr structure.

- listen(int sockfd, int backlog)

  - willing to receive connection.
  - the backlog specifies how many pending connections the system allows before refusing the connection.

- int accept(int sockfd, struct sockaddr *addr, int *addrlen)

  - accepts a connection on a socket from client.
  - it return a new socket descriptor to the server. This new descriptor can be used to communicate with a specific client.
  - The original descriptor sockfd can be used to accept more connections.
  - If addr is not empty, it will stored the client socket information in the structure.

- int connect(int sockfd, struct sockeaddr *name, int *namelen)

  - initiates a connection to a server.
  - Fill in information of server process

        - For AF_UNIX : See bind() above

        - For AF_INET : see <netinet/in.h>
          use structure sockaddr_in to fill in server information
          cast it to sockaddr structure.

    - Use read(2) and write(2) to communicate, and close(2) and unlink(2) to close and remove socket.

- Note :
    - You should check the return values of system calls. It is important that you make sure everything is fine before you move to the next step.
    - For illustration, I skip the error checking in the following two example.
    - Also, make sure to remove socket file before rerun the program, this will cause errors.

- **Example 1 : AF_UNIX**

```
/*
**      SERVER
**
**      In libra, use "cc -o server server.c -lsocket " to compile
**
**      Usage : server
**      Read from client, write to client
*/

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

main(int argc, char *argv[])
{
        struct sockaddr  server;
        int mySocket, newSocket;
        char myBuf[1024];
        /* Step 1 */
        mySocket = socket(AF_UNIX, SOCK_STREAM, 0);

        /* Step 2 */
        server.sa_family = AF_UNIX;             /* domain */
        strcpy(server.sa_data,"/tmp/MySocket");      /* name */
        bind (mySocket, &server, sizeof (server));

        /* Step 3 : set max # pending connection */
        listen ( mySocket, 4);

        newSocket = accept(mySocket, 0, 0); /* Step 4 */

        /* usually, fork(2) a child to service the request,      */
        /* parent goes back to accept                            */

        /* Use system calls, read and write, to communicate */
        read ( newSocket, myBuf, sizeof( myBuf ));
        printf("%s\n", myBuf);
        write ( newSocket, "OK! I got it \n" , 20);

        /* Close the socket */
        close (newSocket);
        close (mySocket);
        unlink("/tmp/MySocket");
}


/*
**      CLIENT
**      In libra, use "cc -o client client.c -lsocket" to compile
**
**      Usage : client message
**
**      Example : client "hello"
**      Write to server, read from server
*/
```

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

main( int argc, char *argv[])
{
        struct sockaddr client;
        int mySocket;
        char myBuf[100];

        /* step 1 : make a unamed socket */
        mySocket = socket(AF_UNIX, SOCK_STREAM, 0);  /* Step 1 */

        /* Step 2 : connect to named server */
        client.sa_family = AF_UNIX;              /* domain */
        strcpy(client.sa_data,"/tmp/MySocket");       /* name */
        connect ( mySocket, &client, sizeof (client));

        /* Use system calls, read and write, to communicate */
        strcpy(myBuf, argv[1]);
        write(mySocket, myBuf, sizeof( myBuf ));
        read(mySocket, myBuf, sizeof( myBuf ));
        printf("%s\n",myBuf);

        /* close the socket */
        close(mySocket);
}
```

- For network program, we introduce several system functions :

    - Different machine may store the bytes in an integer in different order.
    - You should convert to network byte order before sending data,
    - and covert the network order to host order when receiving the data

```
#include <sys/types.h>
#include <netinet/in.h>

htonl(3XN)
```

```
htons(3XN)
ntohl(3XN)
ntohs(3XN)
```

- To get the IP address of a host, use

```
#include <netdb.h>

gethostbyname(3XN)
gethostbyaddress(3XN)
```

  - Both functions return :

```
struct  hostent {
        char     *h_name;          /* official name of host */
        char     **h_aliases;      /* alias list */
        int      h_addrtype;       /* host address type */
        int      h_length;         /* length of address */
        char     **h_addr_list;  /* list of addresses from name server */
};
```

- To get the host name

```
#include <sys/utsname.h>
uname(2) → return the hostname
```

Example 2 : AF_INET

```
/*
**    SERVER
**
**    In libra, use "cc -o server server.c -lsocket -lnsl" to compile
**
**    Usage : server
**
**    Read from client, write to client
*/
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <sys/utsname.h>

#define PORT_NUMBER 1335

main(int argc, char **argv)
{
        struct utsname name;
        struct sockaddr_in sin;
        int mySocket, newSocket;
        char myBuf[1024];

        /* Step 1 */
        mySocket = socket(AF_INET, SOCK_STREAM, 0);

        /* Step 2 */
        uname(&name);
        host = gethostbyname(name.nodename);
       memcpy((char *) &sin.sin_addr, host->h_addr_list[0],
                                              host->h_length);
        sin.sin_family = AF_INET;
        sin.sin_port =htons( PORT_NUMBER );
        bind (mySocket, (struct sockaddr *) &sin, sizeof (sin));

        listen ( mySocket, 2);  /* Step 3 */

        newSocket = accept(mySocket, 0, 0); /* Step 4 */

        /* Use system calls, read and write, to communicate */
        read ( newSocket, myBuf, sizeof( myBuf ));
        printf("%s\n", myBuf);
        write ( newSocket, "OK! I got it \n" , 20);

        /* Close the socket */
        close (newSocket);
}
```

```
/*
**      CLIENT
**      In libra, use "cc -o client client.c -lsocket -lnsl" to compile
**
**      Usage : client hostname message
**
**      Example : client futon "hello"
**
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define PORT_NUMBER 1335

main( int argc, char *argv[])
{
struct hostent *myHost;
struct sockaddr_in sin;
int mySocket;
char myBuf[100];
```

```
        /* To change the hostname "futon" to Internet address */
        myHost = gethostbyname(argv[1]);
        bcopy(myHost->h_addr_list[0], &sin.sin_addr, myHost->h_length);

        sin.sin_family = AF_INET;
        sin.sin_port = htons ( PORT_NUMBER );
        strcpy(myBuf, argv[2]);

        mySocket = socket(AF_INET, SOCK_STREAM, 0);  /* Step 1 */

        /* Step 2 */
        connect ( mySocket, (struct sockaddr *)&sin, sizeof (sin));

        /* Use system calls, read and write, to communicate */

        write(mySocket, myBuf, sizeof( myBuf ));
        read(mySocket, myBuf, sizeof( myBuf ));
        printf("%s\n",myBuf);

        /* close the socket */
        close(mySocket);
}
```

- Manage a set of opened file descriptors

    - Processes may not want to block in accept(), read() or write() system calls.

    - We can use select(3C) or poll(2) to determine the activities of a set of file descriptors (for any system calls using file descriptors)

    - select(3C) :

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

int select(int nfds, fd_set *readfds, fd_set *writefds,
        void fd_set *execptfds, struct timeval *timeout);
```

nfds	: total number of file descriptors in your program
readfds	: the file descriptors that are waiting for incoming data (including accept())
writefds	: the file descriptors that are waiting for writing data
execptfds	: any error condition may be happen on file descriptors
timeout	: NULL → wait until an event has arrived 0 (in struct) → no blocking > 0 (in struct) → return after the given time

- select(3C) use one bit to represent a file descriptor. Bit manipulation on a set of file descriptors can be simplified by using the following macros

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

/* Initialize the fd_set */
void FD_ZERO(fd_set *fdset);

/* assign a bit for fd in fd_set */
void FD_SET(int fd, fd_set *fdset);

/* remove the assigned bit for fd in fd_set */
void FD_CLR(int fd, fd_set *fdset);

/* check whether the bit for fd in fd_set is set */
int FD_ISSET(int fd, fd_set *fdset);
```

- Simple example :

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

int fd1, fd2;
fd_set readSet;

fd1 = open(…);
fd2 = open(…);
```

```
FD_ZERO(&readSet);
FD_SET(fd1, &readSet);
FD_SET(fd2, &readSet);

select(5, &readSet, NULL, NULL, NULL);
if (FD_ISSET(fd1, &readSET)) {
        … fd1 may read data …
}

if (FD_ISSET(fd2, &readSET)) {
        … fd2 may read data …
}
```

- Note : before reuse the select() again, you have to make sure to set all bits to initial state.

7.6. Introduction to Named Stream Pipe (SVR4)

- Traditionally, pipes provide a unidirectional communication channel for parent and children processes.

- FIFOs (named pipe) allow independent processes to communicate by creating a FIFO file

- Stream pipe was introduced to provide more flexible IPC method.

- It is a bidirectional (full-duplex) pipe and is created by pipe(2) function.

    (Yes, it is same as pipe(2) introducing in 6.3.)
    e.g.

    ```
    int fd[2];
    pipe(fd[2]);
    both fd[0] and fd[1] can be used to read and write data
    ```

- Formally, pipe(2) create two stream heads that are connected to each other, i.e. one file descriptor per stream head.

```
fd[0]  <->     stream head
                 |        |
                 |        |
fd[1]  <->     stream head
```

- Stream heads allow user to push processing module onto the stream using ioctl(2) system call. e.g. push processing module on stream fd[1]

```
fd[0]  <->                           stream head
                                        |      |
                                        |      |
fd[1]  <->     stream head <-> processing module
```

- Streams and processing modules are advanced topics and are not covered here. We focus on a processing module, called   onnld”, which allows creation of named stream pipe.

- Named stream pipes allow independent processes (clients and server) to communicate

- **<u>Here are the major steps of server :</u>**

1. Create a pipe

2. Push   onnld” into one end of pipe, fd[1], using ioctl(2).
   Example : ioctl(fd[1], I_PUSH, "connld");

3. Create a file with proper permission (e.g. 666) and use fattach(2) to attach the pathname to the end of fd[1]. (Note : all clients need to know the pathname).
   Example : fattach(fd[1], "tempFile");

4. Use ioctl(2) to wait for a client connection at fd[0].
   Example :
   ```
           struct strrecvfd conversation_info;
           int retval, fd_server;
           ioctl(fd[0], I_RECVFD, &conversation_info)
           fd_server = conversation_info.fd;
   ```

   ← wait for client connection through open(2) →

   - when connected, a new stream pipe is created, one file descriptor, fd_client, is returned to client (see Client step 1), anothe file descriptor, fd_server, is returned to server thorugh ioctl(2).

```
fd_client   ←→                              stream head
                                               |      |
                                               |      |
fd_server   ←→                              stream head

fd[0]       ←→                              stream head
                                               |      |
                                               |      |
fd[1]       ←→   stream head ←→                 onnld”
```

5. Use fd_server to communicate with the client, i.e. use read(2) and write(2). Server can also wait for other client connection using fd[0] (see step 4). Here, you may use select() to handle multiple file descriptors.

- **<u>Here are the major steps of client</u>**

1. Use open(2) to open the file (created at Server step 3)
   Example : fd_client =open("tempFile", O_RDWR);

2. Use the file descriptor returning from open(2), fd_client, to communicate with server

Chapter 8. Introduction to multithreaded programming

8.1. Introduction to threads

- What are threads

    - A UNIX process is a program in execution. It may be view as a single thread with task (process address space, process control info. and etc)

    - New POSIX Pthreads interface standard and the UNIX International (UI) interface standard allows more than one thread within a task.

    - Threads share all process memory, e.g. if a thread modifies certain data, other threads can read the modified data.

- Comparing threads with processes

Threads	**Processes**
in the same task → share process memory	in different tasks → do not share process memory
communicate using regular variables in the program	communicate using IPC (complex to set-up)
faster to create a new thread (around 50 - 300 microsec)	to create a new process (about 1700 microsecs)

Threads	**Processes**
complex job in the same environment can be break into smaller functions for execution by several threads	may need several processes
signal features are more complex with threads	signal features are simpler in process environment
many library routines and system calls are not thread-safe	library routines and system calls are always OK
shared data → need to handle thread synchronization within program	only need to worry about synchronization in multi-process environment
need to be careful about local declared automatic variables when using threads to share data	do not have to worry about this

- Solaris threads scheduling
    - User threads
        - The kernel do not aware of the existence of user threads. The management of user threads is done by an applications (a thread library). It includes the user threads scheduling

    - Main advantages : Very fast (much faster creating a LWP), do not require involvement of OS and portability

  - Lightweight Process (LWP)

    - LWP management are handled by kernel. Each LWP is associated to one kernel thread that is scheduled by kernel directly.

    - Multiple user threads can share a one or more LWP. User threads are scheduled (and can be bound) onto LWP by the threads library

- Problems in multithreaded programming

  - thread safe functions

    - Functions are thread-safe or reentrant → may be called by more than one thread at a time and still produce correct solution
      (note : race condition by multiple threads may be happen in some old function calls → not thread-safe)

    - Example :

      Not thread-safe function : char *ctime(const time_t *clock) → input a pointer to number (i.e. *clock), return a readable time in string format.

      The returning string is allocated statically → any thread calls ctime() function again will overwrite the previous information in the string.

      To overcome this problem, should use thread safe function ctime_r() in multithreaded programs.

    - To find out whether or not a function is thread-safe :

        man page should state whether or not a function is thread-safe. If it is not thread-safe, it should suggest an alternative function.

    - Memory allocation functions, e.g. malloc(), are thread-safe.

- synchronization

    - when multiple threads access same common data objects, race condition may result unexpected results

    - several ways may be used to avoid this problem :

        - try to allocate data object dynamically, so that every thread get different data object
        - use lock and unlock to synchronize threads

- error

    - the library has been updated to allow each thread has its own errno variable

    - your program must include <errno.h>

- compiling with "-D_POSIX_C_SOURCE=199506L"
  → turns on thread safe definitions of interface.

<u>8.2. Basic thread control routines</u>

- Must include header file :   #include <pthread.h>

- To compile pthread program with thread-safe definitions of interfaces (including errno.h) :

```
tula% cc p1.c -lpthread -D_POSIX_C_SOURCE=199506L
```

- Zero indicates a successful return and a non-zero value indicates an error

- Thread control functions (Please refer to man page for complete information) :

1. Thread Creation : create a new thread. It will start to execute at specified function and terminate when it reaches the end of function.

    - prototype :

```
int pthread_create(
        pthread_t *new_thread_ID,
        const pthread_attr_t *attr,
        void * (*start_func)(void *),
        void *arg
);
```

      *new_thread_ID : pointer to a thread variable. If the function successfully creates a new thread, this point to the new thread ID.

      pthread_attr_t *attr : To modify the attributes of new thread. NULL → use default set of attributes. Attributes include

        contentionscope : use pthread_attr_setscope() and pthread_attr_getscope() functions to set and get this attribute.
        - PTHREAD_SCOPE_SYSTEM : bound to LWP
        - PTHREAD_SCOPE_PROCESS : unbound thread (default)

        detachstate : use pthread_attr_setdetachstate() and pthread_attr_getdetachstate() functions to set and get this attribute.
        - PTHREAD_CREATE_DETACHED : create a detached thread. The thread disappears when it is done without leaving a trace

- PTHREAD_CREATE_JOINABLE : the thread exit information is not free until pthread_join(3T). (similar to wait(2) to receive the child process exit status)

See man page for other attributes.

*start_func & *arg :

- The new thread starts by calling the function defined by start_func with one argument, arg.
- If more than one argument needs to be passed to start_func, the arguments can be packed into a structure, and the address of that structure can be passed to arg.
- If start_func returns, the thread will terminate with the exit status set to the start_func return value (see pthread_exit(3T) or thr_exit(3T)).

- Example :

<u>Default thread creation:</u>

```
pthread_t tid;
void *start_func(void *), *arg;

pthread_create(&tid, NULL, start_func, arg);
```

<u>User-defined thread creation:</u>

```
To create a thread that is bound to LWP :
pthread_attr_init(&attr); /* initialize attr with default attributes */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
      pthread_create(&tid, &attr, start_func, arg);
```

- Warning : make sure <u>arg</u> structure is not declared as automatic variable in the caller function which may exit before the thread is done referencing it.

2. <u>Thread Termination</u> :

   - Thread is destroyed when

     - it reaches the end of the starting function. The function may return a pointer value which will become the exit status
       OR
     - it calls pthread_exit(void *status) function

   - the exit status is available to other threads using pthread_join(3T)
   - warning : the exit status should not point to data allocated on the exiting thread stack → data may be freed and reused when the thread exits.

3. <u>Waiting for thread termination</u> :

   - use pthread_join(3T) to get a thread completion status.

   - Prototype :

     int pthread_join(pthread_t ***thread***, void **statusp)

     - ***thread*** : wait for a thread whose ID is ***thread***.
     - **statusp : completion status (note : this can be pointed to any structure, long string, array of structures and strings etc)

4. Get the thread ID : pthread_t pthread_self()

5. Comparing thread ID : int pthread_equal(pthread_t t1, pthread_t t2)
   same thread → return non zero; different threads → return 0

- Example :

```
/*
        producer thread :  read data from file and store into
                           dynamically allocated buffer
        consumer thread :  the main thread. It consumes data in buffer
                           whenever the data is ready
                           (note : consumer read the 1st set of data )

*/

#include <pthread.h>
#include <unistd.h>

void display_data(char *buf_ptr);
void *get_next_file(void *arg);


int main(int argc, char **argv)
{
        pthread_t producer_thread;
        int datafile_number;
        int total_number;
        char *data_buf;

        total_number    = 10;        /* can be read using argc and argv */

        for (datafile_number = 1; datafile_number <= total_number;
                                        datafile_number ++)
        {

              if (datafile_number == 1) {

                    /* consumer read the 1st data_file */
                   data_buf = get_next_file((void *)&datafile_number);

              } else {
                    pthread_join(producer_thread, (void **) &data_buf);
              }
```

```
            if (datafile_number < total_number) {
                  pthread_create(
                                    &producer_thread,
                                    NULL,
                                    get_next_file,
                                    (void *) &datafile_number
                                    );
            }

            display_data(data_buf);

      }
      return(0);

}

void display_data(char *buf_ptr)
{
      /* simulate displaying data */
      printf("        In display_data : the buffer is : %s\n",buf_ptr);
}


void *get_next_file(void *arg)
{
      char *buf;

      printf("In get_next_file : thread_ID=%d : arg=%d\n",
                  pthread_self(),*((int *) arg));

      /* simulate get data from file */
      buf = (char *) malloc(30);
      sprintf(buf, "this is string number %d", *((int *) arg));
      return(buf);
}
```

- Sample Run :

```
libra% cc try.c -lpthread -D_POSIX_C_SOURCE=199506L
libra% a.out
In get_next_file : thread_ID=1 : arg=1
        In display_data : the buffer is : this is string number 1
In get_next_file : thread_ID=4 : arg=2
        In display_data : the buffer is : this is string number 2
In get_next_file : thread_ID=5 : arg=3
        In display_data : the buffer is : this is string number 3
In get_next_file : thread_ID=6 : arg=4
        In display_data : the buffer is : this is string number 4
In get_next_file : thread_ID=7 : arg=5
        In display_data : the buffer is : this is string number 5
In get_next_file : thread_ID=8 : arg=6
        In display_data : the buffer is : this is string number 6
In get_next_file : thread_ID=9 : arg=7
        In display_data : the buffer is : this is string number 7
In get_next_file : thread_ID=10 : arg=8
        In display_data : the buffer is : this is string number 8
In get_next_file : thread_ID=11 : arg=9
        In display_data : the buffer is : this is string number 9
In get_next_file : thread_ID=12 : arg=10
        In display_data : the buffer is : this is string number 10
libra%
```

8.3. Basic synchronization

- There are few basic synchronization primitives available in pthread library. We cover mutual exclusion lock (mutex) and condition variable (cond) in this section.

- Mutual exclusion lock functions :

  1. Header file and default initialize the lock :

     - #include <pthread.h>
     - pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER

     /* dynamically allocate mutex */

     - pthread_mutex_t *mutex;
     - *mutex = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));

  2. Initializing mutexes.

     - int pthread_mutex_init(pthread_mutex_t *mutex,
       const pthread_mutexattr_t *attr);

     - initialize mutex with attributes specified by attr.
     - (see pthread_mutex_lock for description of attributes)
     - you must initialize mutex before using it and should only do it once
     - for pthread_mutex_init(3T), use NULL for default attribute

  3. Destroying mutexes

     - int pthread_mutex_destroy(pthread_mutex_t *mutex);

     - destroys the mutex object referenced by mutex;
     - the mutex object becomes uninitialized and it can be reused by reinitializing using pthread_mutex_init(3T)
     - it is safe to destroy an initialized mutex that is unlocked.

- Attempting to destroy a locked mutex results in undefined behavior.

4. Locking and unlocking mutexes

   - int pthread_mutex_lock(pthread_mutex_t *mutex);
   - int pthread_mutex_unlock(pthread_mutex_t *mutex);
   - int pthread_mutex_trylock(pthread_mutex_t *mutex);

   - to lock and unlock mutex
   - only the thread which locked the mutex may unlock it

   - pthread_mutex_trylock(3T) is identical to pthread_mutex_lock (3T) except that if the mutex object referenced by mutex is currently locked (by any thread, including the current thread), the call returns immediately.

- Condition variable functions :

1. Header file, initializing and destroying condition variables.

   - #include <pthread.h>

     /* set default attributes */
   - pthread_cond_t cond= PTHREAD_COND_INITIALIZER;

   - int pthread_cond_init(	pthread_cond_t *cond,
     const pthread_condattr_t *attr);

   - See pthread_mutex_locked_cond_wait(3T) for description of attributes
   - you must initialize cond before using it and should only do it once
   - for pthread_cond_init(3T), use NULL for default attribute

2. wait operations :

-
  ```
  int pthread_cond_wait(    pthread_cond_t *cond,
                            pthread_mutex_t *mutex);
  ```

-
  ```
  int pthread_cond_timedwait(pthread_cond_t *cond,
                             pthread_mutex_t *mutex,
                             const struct timespec *abstime);
  ```

- functions are used to block on a condition variable.
- they are called with mutex locked by the calling thread or undefined behavior will result.

- these functions atomically release mutex and cause the calling thread to block on the condition variable cond.
  (i.e. another thread can lock the mutex after this operation)

- The threads are waken-up by pthread_cond_signal(3T) or pthread_cond_broadcast(3T)

- The pthread_cond_timedwait() function is returned if the absolute time specified by abstime passes before the condition cond is signaled or broadcasted.

- Usually, the usage of cond variable is as follows :

```
while ( <<condition is not true >> )

/*  releases mutex, waits in cond  (atomic steps)
    when it is signaled or broadcasted,
    resumes execution, locks mutex (atomic steps),
    re-enter while()                                    */

        pthread_condition_wait(cond, mutex);
```

3. signal operations :

- int pthread_cond_signal(pthread_cond_t *cond);
- int pthread_cond_broadcast(pthread_cond_t *cond);

- These two functions are used to unblock threads blocked on a condition variable.

- The pthread_cond_signal() call unblocks at least one of the threads that are blocked on the specified condition variable cond (if any threads are blocked on cond). Note : this may wake-up several threads in multiprocessors environment.

- The pthread_cond_broadcast() call unblocks all threads currently blocked on the specified condition variable cond.

- Example : threads share and update the counter

```
#include <unistd.h>
#include <pthread.h>

#define NUM_THREADS        5
#define NUM_LOOP       1000000

int count = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void * count_function(void *arg)
{
        int n;

        for (n=0; n<NUM_LOOP; n++) {
                /* pthread_mutex_lock(&lock); */
                count++;
                /* pthread_mutex_unlock(&lock); */
        }
}

```

```c


main()
{
int i;
pthread_t threads[NUM_THREADS];
pthread_attr_t attr;


printf("There are %d threads\n", NUM_THREADS);
printf("Each thread increases the counter  %d times\n", NUM_LOOP);
printf("The expected result is count = %d\n", NUM_THREADS * NUM_LOOP);

/* bind each thread with LWP */
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

/* start child threads here */
for(i=0; i<NUM_THREADS; i++)
{
        pthread_create(&threads[i], &attr, count_function, NULL);
}

/* main thread wait for tremination of all child threads */
for(i=0; i<NUM_THREADS; i++)
{
        pthread_join(threads[i], NULL);
}

/* print the computed result */
printf("The computed result is count = %d\n", count);
}
```

- Sample Run :

```
/* use mutex to protect the share counter */
libra% !cc
cc lock.c -D_POSIX_C_SOURCE=199506L -lpthread
libra% a.out
There are 5 threads
Each thread increases the counter  1000000 times
The expected result is count = 5000000
The computed result is count = 5000000
libra%
```

```
/* do not use mutex to protect the counter → get incorrect answer */
libra% !c
cc lock.c -D_POSIX_C_SOURCE=199506L -lpthread
libra% a.out
There are 5 threads
Each thread increases the counter  1000000 times
The expected result is count = 5000000
The computed result is count = 2172084
```

8.4. From POSIX pthreads to UI threads

- man pthreads(3T) provide description of POSIX pthreads and UI threads.

- There are many similar functions available in UI thread interfaces (actually, you can use mixed system calls in the same program). Here is the list of system calls covering in this chapter and the corresponding functions from UI threads

POSIX (libpthread)	Solaris (libthread)
pthread_create()	thr_create()
pthread_exit()	thr_exit()
pthread_join()	thr_join()
pthread_self()	thr_self()
pthread_mutex_init()	mutex_init()

pthread_mutex_lock()	mutex_lock()
pthread_mutex_trylock()	mutex_trylock()
pthread_mutex_unlock()	mutex_unlock()
pthread_mutex_destroy()	mutex_destroy()

POSIX (libpthread)	Solaris (libthread)
pthread_cond_init()	cond_init()
pthread_cond_wait()	cond_wait()
pthread_cond_timedwait()	cond_timedwait()
pthread_cond_signal()	cond_signal()
pthread_cond_broadcast()	cond_broadcast()
pthread_cond_destroy()	cond_destroy()
many attribute related calls	none

- To use UI thread functions :
  - Must include header file : #include <thread.h>
  - To compile UI thread program with thread-safe definitions of interfaces : libra% cc -mt p1.c
  - To defined thread variable : thread_t threadVar;
- Example :

```
/*

       Taken from "man pthread_create"

    Example 1: This is an example  of  concurrency  with  multi-
    threading. Since POSIX threads and Solaris threads are fully
    compatible even within the same process, this  example  uses
    pthread_create()  if you execute a.out 0, or thr_create() if
    you execute a.out 1.

    Five threads  are  created  that  simultaneously  perform  a
    time-consuming function, sleep(10). If the execution of this
    process is timed, the results will show that all five  indi-
    vidual calls to sleep for ten-seconds completed in about ten
    seconds, even on a uniprocessor. If a  single-threaded  pro-
    cess  calls sleep(10) five times, the execution time will be
    about 50-seconds.

    The command-line to time this process is:

    /usr/bin/time a.out 0 (for POSIX threading)

    or

    /usr/bin/time a.out 1 (for Solaris threading)
*/

/* cc thisfile.c -lthread -lpthread */

#define _REENTRANT    /* basic 3-lines for threads */
#include <pthread.h>
#include <thread.h>

#define NUM_THREADS 5
#define SLEEP_TIME 10

void *sleeping(void *);   /* thread routine */
int i;
thread_t tid[NUM_THREADS];      /* array of thread IDs */

int
main(int argc, char *argv[])
{
        if (argc == 1)  {
                printf("use 0 as arg1 to use pthread_create()\n");
                printf("or use 1 as arg1 to use thr_create()\n");
                return (1);
        }

        switch (*argv[1])  {
                case '0':  /* POSIX */
```

```
            for ( i = 0; i < NUM_THREADS; i++)
                pthread_create(&tid[i], NULL, sleeping,
                    (void *)SLEEP_TIME);
            for ( i = 0; i < NUM_THREADS; i++)
                pthread_join(tid[i], NULL);
            break;

        case '1':  /* Solaris */
            for ( i = 0; i < NUM_THREADS; i++)
                thr_create(NULL, 0, sleeping,
                    (void *)SLEEP_TIME, 0, &tid[i]);
            while (thr_join(NULL, NULL, NULL) == 0)
                ;
            break;
        }  /* switch */

    printf("main() reporting that all %d threads have terminated\n", i);
    return (0);
}  /* main */

void *
sleeping(void *arg)
{
    int sleep_time = (int)arg;
    printf("thread %d sleeping %d seconds ...\n",
        thr_self(), sleep_time);
    sleep(sleep_time);
    printf("\nthread %d awakening\n", thr_self());
    return (NULL);
}
```

- Sample run :

```
libra% a.out 1
thread 4 sleeping 10 seconds ...
thread 5 sleeping 10 seconds ...
thread 6 sleeping 10 seconds ...
thread 7 sleeping 10 seconds ...
thread 8 sleeping 10 seconds ...

thread 5 awakening

thread 6 awakening

thread 7 awakening

thread 4 awakening

thread 8 awakening
main() reporting that all 5 threads have terminated


libra% a.out 0
thread 4 sleeping 10 seconds ...
thread 5 sleeping 10 seconds ...
thread 6 sleeping 10 seconds ...
thread 7 sleeping 10 seconds ...
thread 8 sleeping 10 seconds ...

thread 7 awakening

thread 5 awakening

thread 4 awakening

thread 8 awakening

thread 6 awakening
main() reporting that all 5 threads have terminated
libra%
```

8.5. References

Sun Solaris Manual Section 3T

Programming with Threads
by Kleiman, Shah and Smaalders
1996, Prentice hall
Note : Good reference for Pthread programming

Guide to Multithreaded Programming
SUN Microsystems, Part Number 801-3176-03

Practical Unix Programming : A Guide to Concurrency, Communication, and Multithreading
by Kay A. Robbins, Steven Robbins, Steve Robbins (Contributor)
1996, Prentice Hall;