Linux Programming

Course’ name : Linux Programming

Objective: After completing the course,
Trainees can understand generally about some programming technique in Linux.

Contents:
- Shell script in Linux
- Programming C on Linux (basic and advance)

Prerequisite: Basic C programming language

Methodology: 75% theory
25% practice

Course: Linux programming (6 sessions, 1 session/week)

Part 1: Shell script (1 weeks)

- Shell, input and output, shell programming, history, ...

Part 2:

- ***Part-A. Programming C on Linux (2 weeks)***
  - GCC, Compile, Macro, Makefile, GDB, Q&A
- ***Part-B. Advanced Programming C on Linux (3 weeks)***
  - File I/O concept
  - Memory allocation
  - Multiple processes and Multiple threads
  - IPC (Socket, pipe, shared memory...)

Part 1 - Shell script (1 sessions)

- **Main contents:**
  - What’s "the shell"?
  - shell Basic: cd, ls, find, ?*, {}, alias, ...
  - Variables and Parameters
  - Input and Output
  - Background Jobs
  - Special Characters
  - Command history
  - Basic Shell Programming
- **Q&A :**

Part 2A - Programming C on Linux (2 sessions)

- **Main contents:**
  - Compile source code using GCC
  - Makefile
  - Macro
  - Debug GDB
- **Q&A :**

Part 2B - Advance programming C on Linux (2 sessions)

- **Main contents:**
  - System programming concepts: System Calls, Library Functions, File I/O concept, Executable and Linkable Format, The /proc File System
  - Memory Allocation
  - Signal Overview
  - Multiple processes
  - Multi Threads
  - Interprocess communication: pipe, shared memory, sockets
- **Q&A :**

Part1 : Shell script

What is Shell:

- The shell is a program that takes keyboard commands and passes them to the operating system to carry out
- Almost all Linux distributions supply the bash shell
- The name “bash” is an acronym for “Bourne Again SHell”.
- When using a graphical user interface, we need another program called a terminal emulator to interact with the shell.
- KDE uses konsole and GNOME uses gnome-terminal

Part1 : Shell script

Navigation:

- The directory you are standing in is called the **working directory**.
  - [me@linuxbox me]$ pwd
  - /home/me
  -
- pwd: find the name of the working directory.
- ls: list the files in the working directory.
- cd: change your working directory.

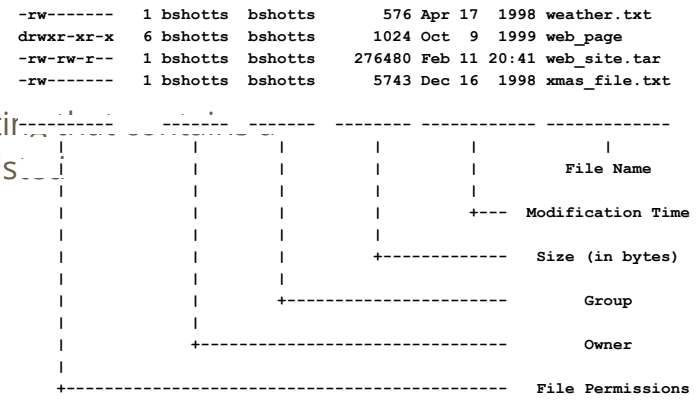
Part1 : Shell script

Navigation:

- The directory you are standing in is called the **working directory**.
- pwd: find the name of the working directory.
- ls: list the files in the working directory.
- cd: change your working directory.

Looking Around:

- use the -l option with ls, you will get a file listing that contains a wealth of information about the files being listed.



Part1 : Shell script

Variables and Parameters:

- Local variables: internal variables of shell
- Environmental variables: each process has an "environment", process inherits a duplicate environment of parent process. The **export** command makes available variables to all child processes
- Positional parameters:
  - Access command line parameters via $N (0 <= N <= 9), ${N} when N > 9
  - $* : all parameters that not within double quotes
  - $@ : all parameters within double quotes
  - .....
- The exit status or return code access via $?

Part1 : Shell script

Input and Output:

- Each process: standard input (0), standard output (1), standard error (2).
- I/O Redirection: By default, stdin, stdout and stderr are associated to your terminal.
  - <: reads input from file
  - > : write to new file
  - >> : append to file
  - >& : redirect to file descriptor
- PIPE: connect the standard output of one command to the standard input of another.

  Ex: cat 123.txt | ssh user@abc.com “cat > /tmp/123.txt”

- Discard the output with /dev/null:

  Ex: find /proc/ -name "environ" 2>/dev/null

Part1 : Shell script

Background Jobs:

- Normally, the shell will let the command have control of your terminal until it is done
- Put an ampersand (&) after the command to let it runs in the background, commands run in this way is called a background jobs
- There are several commands that can be used to control processes. They are:
  - ps: list the processes running on the system
  - kill: send a signal to one or more processes (usually to "kill" a process)
  - jobs: an alternate way of listing your own processes
  - bg: put a process in the background
  - fg: put a process in the foreground

Example:

```
# ping 192.168.81.31 >/dev/null &
# [1] 27131
# jobs
# [1]+  Running       ping 192.168.81.31 > /dev/null &
```

Part1 : Shell script

Special Characters:

- Special Characters
  - ~ : Home directory
  - ` : Command substitution (archaic)
  - # : Comment
  - $: Variable expression
  - & : Background job
  - * : String wildcard
  - ( ) : start & end subshell
  - \ : Quote next character
  - | : pipe
  - [ ] : character-set wildcard
  - { } : command block
  - ; : shell command separator
  - <: input redirect
  - >: output redirect
  - / : Pathname directory separator
  - ? : Single-character wildcard
  - ...
- Quoting: Sometimes you will want to use special characters literally, i.e., without their special meanings. You must surround a string of characters with single quotation marks (or quotes).

  Ex: echo '2 * 3 > 5 is a valid inequality.'

- backslash-escaping: Another way to change the meaning of a character is to precede it with a backslash (\).

  Ex: echo 2 * 3 \> 5 is a valid inequality

Part1 : Shell script

Command history:

- It's always possible to make mistakes when you type at a computer keyboard. Some shells support **history** mechanism
- What we call do with shell history:
  - Support scroll through the history:
    - UP arrow key: Scroll backwards in history
    - DOWN arrow key: Scroll forwards in history
  - Reviewing your command history: **# history**
  - Executing commands from your command history:
    - **!n** : Refer to command line n.
    - **!-n** : Refer to the command n lines back.
    - **!!** : Refer to the previous command. This is a synonym for '!-1'.
    - **!string** : Refer to the most recent command preceding the current position in the history list starting with string.

Part1 : Shell script

Basic Shell Programming:

- Shell Script is series of command written in plain text file
- Why to Write Shell Script ?
  - Shell script can take input from user, file and output them on screen.
  - Useful to create our own commands.
  - Save lots of time.
  - To automate some task of day today life.
  - System Administration part can be also automated.
- **test command or [expr]**: test command or [expr] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero for false.
  - Mathematics:
    - -eq : is equal to
    - -ne : is not equal to
    - -lt : is less than
    - -le : is less than or equal to
    - -gt : is greater than
    - -ge : is greater than or equal to
  - String:
    - string1 = string2 : string1 is equal to string2
    - string1 != string2: string1 is NOT equal to string2
    - String1 : string1 is NOT NULL or not defined
    - -n String1 : string1 is NOT NULL and does exist
    - -z string1: string1 is NULL and does exist

Part1 : Shell script

- File:
  - -s file : Non empty file
  - -f file : Is File exist or normal file and not a directory
  - -d dir : Is Directory exist and not a file
  - -w file : Is writeable file
  - -r file : Is read-only file
  - -x file : Is file is executable
- Logical Operators:
  - ! expression : Logical NOT
  - expression1 -a expression2 : Logical AND
  - expression1 -o expression2 : Logical OR

- **If...else...fi**: If given condition is true then command1 is executed otherwise command2 is executed.

Syntax:

```
if condition
then
        condition is zero (true - 0)
        execute all commands up to else
statement
else
        if condition is not true then
        execute all commands up to fi
fi
```

Ex:

```
if test $1 -gt 0
then
        echo "$1 number is positive"
else
        echo "$1 number is negative"
fi
```

Part1 : Shell script

- **while loop**:

Syntax:

```
while [ condition ]
do
        command1
        command2
        ....
done
```

Ex:

```
i=1
while [ $i -le 100 ]
do
        echo "Welcome $i times"
        i=`expr $i + 1`
done
```

- **case Statement**:

Syntax:

```
case  $variable-name  in
        pattern1)  ...
                command;;
        pattern2)  ...
                command;;
        patternN) ...
                command;;
        *)        ...
                command;;
esac
```

Ex:

```
option="${1}"
case ${option} in
  -f) FILE="${2}"
     echo "File name is $FILE"
     ;;
  -d) DIR="${2}"
     echo "Dir name is $DIR"
     ;;
  *)
     echo "`basename ${0}`:usage: [-f file] | [-d directory]"
     exit 1
     ;;
esac
```

Part1 : Shell script

- **read variable**: Use to get input (data from user) from keyboard and store (data) to variable

Ex:

```
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

- **read variable < filename**: Read line of a text file.

Ex:

```
while read p; do
        echo $p
done <data.txt
```

Part 2A - Programming C on Linux (2 sessions)

- **Main contents:**
  - GCC
  - Compilation process
  - Macro
  - Makefile
  - GDB Debugger Tool
- **Q&A :**

1. GCC

What is GCC ?

- GCC (GNU Compiler Collection) is a key component in GNU project.
- GCC supports many languages such as C, C++, Java, Ada.
- “gcc” is a very powerful and popular C compiler used for Linux distributions.

- There are “a thousand of” options can use with gcc.
- Show full description about gcc *:* ***$ man gcc***

Synopsis :

gcc [**-c**|-S|-E] [-std=standard]
[-g] [-pg] [-Olevel]
[-Wwarn...] [-Wpedantic]
[-Idir...] [-Ldir...]
[-Dmacro[=defn]...] [-Umacro]
[-foption...] [-mmachine-option...]
[**-o outfile**] [@file] infile...

```
$ gcc -Wall -c gcc_ex1.c -o gcc_ex1 -v
```

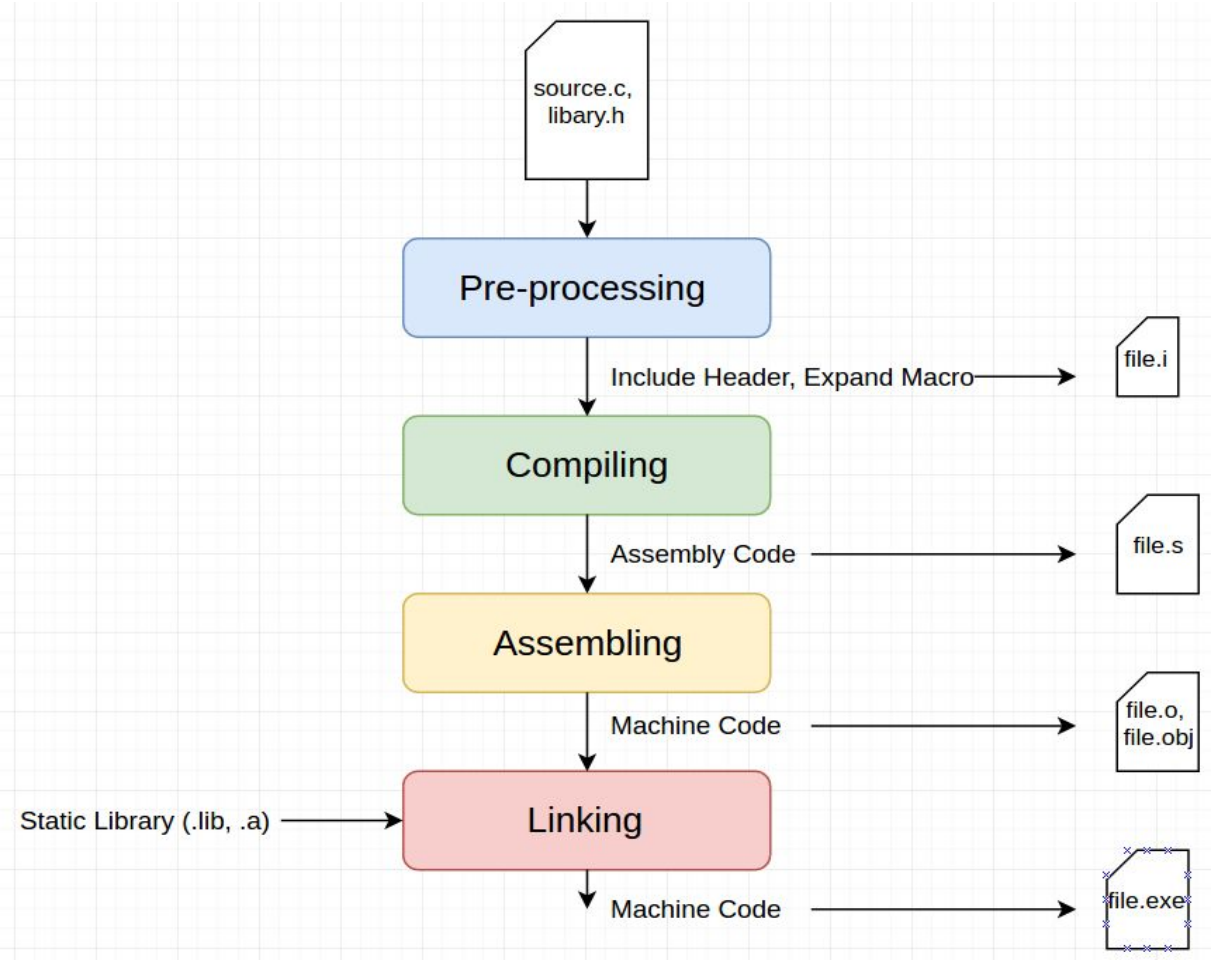
-c : source files
-o : output file name
-Wall : show all warning messages
-v : show output processing

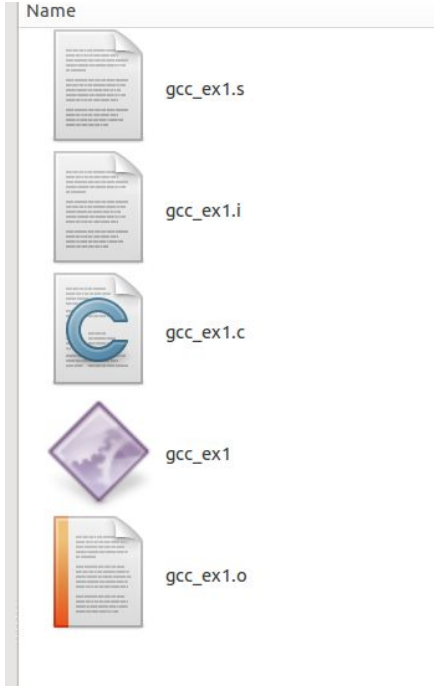
2. Compilation Process

- Compiling a C program is a multi-stage process.
- In general, the process can be slipped into **four** separate stages:

Preprocessing ⇒ Compilation ⇒ Assembly ⇒ Linking

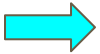
Compilation Processes





1. ***Pre-processing***
   - The C preprocessor modifies a source code before handing it over to the compiler.
     + Remove comments
     + Include libraries
     + Expand Macros

```c
#include <stdio.h>

#define ADD_FUNC(a,b)  a+b


int main ()
{

    int a, b;
    a = 5;
    b = 21;

    // Print a+b result:
    printf("%d + %d = %d\n", a, b,ADD_FUNC(a,b));
    return 0;
}
```



```c
extern int pclose (FILE *__stream);




extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 913 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 943 "/usr/include/stdio.h" 3 4

# 2 "gcc_ex1.c" 2




int main ()
{

 int a, b;
 a = 5;
 b = 21;


 printf("%d + %d = %d\n", a, b,a+b);
 return 0;
}
```

2. Compilation

- Compile ***.i** file into assembly instruction file (***.s**)

$ gcc -S *.i ⇒ *.s

```
    .file   "gcc_ex1.c"
    .section    .rodata
.LC0:
    .string "%d + %d = %d\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $5, -8(%rbp)
    movl    $21, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    -8(%rbp), %edx
    leal    (%rdx,%rax), %ecx
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    movl    %eax, %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"
    .section    .note.GNU-stack,"",@progbits
```

3. Assembly

- Assembler (**as.exe)** converts the assembly code into machine code in the object file ***.o**
- The code is converted into machine language.

$ as <*.s file> -o <*.o file>

```
gcc_ex1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
   f:   c7 45 fc 15 00 00 00    movl   $0x15,-0x4(%rbp)
  16:   8b 45 fc                mov    -0x4(%rbp),%eax
  19:   8b 55 f8                mov    -0x8(%rbp),%edx
  1c:   8d 0c 02                lea    (%rdx,%rax,1),%ecx
  1f:   8b 55 fc                mov    -0x4(%rbp),%edx
  22:   8b 45 f8                mov    -0x8(%rbp),%eax
  25:   89 c6                   mov    %eax,%esi
  27:   bf 00 00 00 00          mov    $0x0,%edi
  2c:   b8 00 00 00 00          mov    $0x0,%eax
  31:   e8 00 00 00 00          callq  36 <main+0x36>
  36:   b8 00 00 00 00          mov    $0x0,%eax
  3b:   c9                      leaveq
  3c:   c3                      retq
```

View : $ objdump -d <object file>

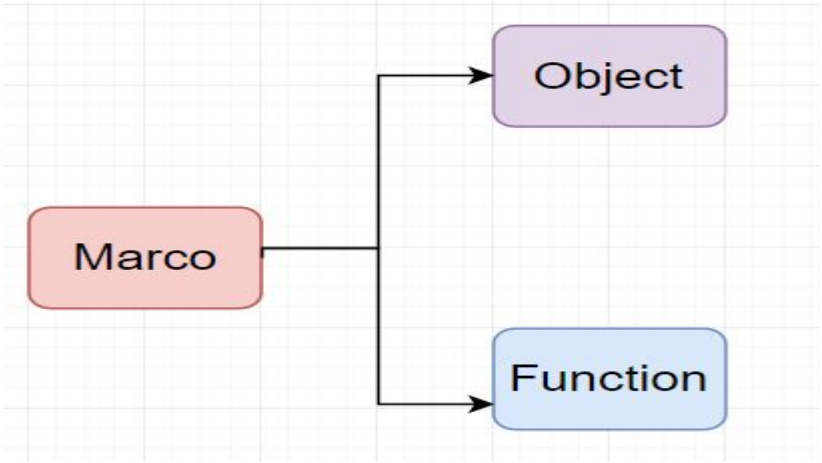
4. Linking:

- The linker (**ld.exe**) links the object code with the library code and produce the code into executable file (*.exe)

```
$ ld <*.o file>... <library>  -o  <*.exe file>
```

3. Macro

- Marco is a fragment of code which a name that defined by **#define** directions.
- Marco is expanded by Preprocessor.
- When the name is used, it is replaced by the content of the macro.
- There are 2 kind of macros: ***Object-like*** macro and ***Function-like*** macro



object-like macro

- ***Object-like macro*** is a simple identifier that look like a data object
- The common way to use ***object-like macro*** is to give name to numeric constant.

Ex :

```
// Object-like MACRO
#define PI              3.14
#define BUFFER_SIZE     1024
#define DEBUG_LOG       1
```

function-like Macro

- Use Macro to define a fragment that look like a function call.

Format:

```
#define  MACRO_NAME(arg1, arg2,...)  [code to expand]
```

ex:

```
1   #include <stdio.h>
2
3   #undef DEBUG_LOG          1
4   #define ADD(a,b) a+b
5   #define PRINT_LOG(msg, arg2...)  do{printf("\n" msg, ##arg2);} while(0)
6
7   int main ()
8   {
9       int a =5, b=6;
10  #ifdef DEBUG_LOG
11      PRINT_LOG("Result: %d\n", ADD(a,b));
12  #endif
13      return 0;
14  }
```

Makefile (GNU Make)

- **Problems with Multiple Source file ?**
  - When source code or header file is changed, the code must be recompiled.

Makefile (GNU Make)

- Makefile is one of component in GNU project
- Makefile are special format file that help build and manage the source code.
- It utility automatically determines which pieces of a large program need to be recompiled.

Makefile syntax:

- A makefile consists of a set of rules:
  - + Target
  - + Dependent files
  - + Commands

target: dependency_1 dependency_2
commands

Examples:

```c
#include <stdio.h>
#define ADD_FUNC(a,b)  a+b

int main ()
{
    int a =5, b=20;
    // Print a+b result:
    printf("%d + %d = %d\n", a, b,ADD_FUNC(a,b));
    return 0;
}
```

make_ex1.c

```
all:    make_ex1
make_ex1:   make_ex1.o
    @gcc -o make_ex1 make_ex1.o
    @echo "Done"

make_ex1.o: make_ex1.c
    @echo "Compiling..."
    @gcc -c make_ex1.c -o make_ex1.

clean:
    @rm make_ex1.o make_ex1
    @echo "Clean old files"
```

makefile

Makefile Syntax

- Target and dependent files are separated by a colon(:)
- The command must be preceded by a tab (NOT spaces)
- The command will not be run if the dependent files are not newer than the target.

Error: “make: Nothing to be done for `all'. “

Makefile: Variables

- A variable begins with a $ and is enclosed within a parenthesis (...) or {}

- **Automatic variables:**

Macro	Definition
$@	the target filename
$*	the target filename without the file extension.
$?	list of the changed dependencies
$<	the first dependency
$^	list of all dependencies (discard duplicates)

```
make_ex1.o: make_ex1.c
    @gcc -c $? -o $@
```

$? = make_ex1.c $@ = make_ex1.o

4. GDB (GNU Debugger)

What is GDB ?

- GDB is one of components in GNU project.
- GDB is a powerful debugging tool for both C/C++
- GDB can help to find bugs by supporting to print out the code, values of variable, control flow, break infinite loop, etc.

GDB command shortcuts

- All command in GDB can be shortened to their first unique letters
- Show all command :
  $ gdb help all

or

http://www.yolinux.com/TUTORIALS/GDB-Commands.html

Command	*Explain*
l <line_number>	*print source code*
p <var>	*print variable's value*
c	*continue*
s	*step*
b <N>	*put breakpoint in line N*
bt	*print a stack trace*
u	*go up*
d	*go down*
enter	
help	*show gdb help*
quit	*exit*

Debug code with GDB

Example: Using GDB to find Segment Fault error (demo)

gdb_example1.c

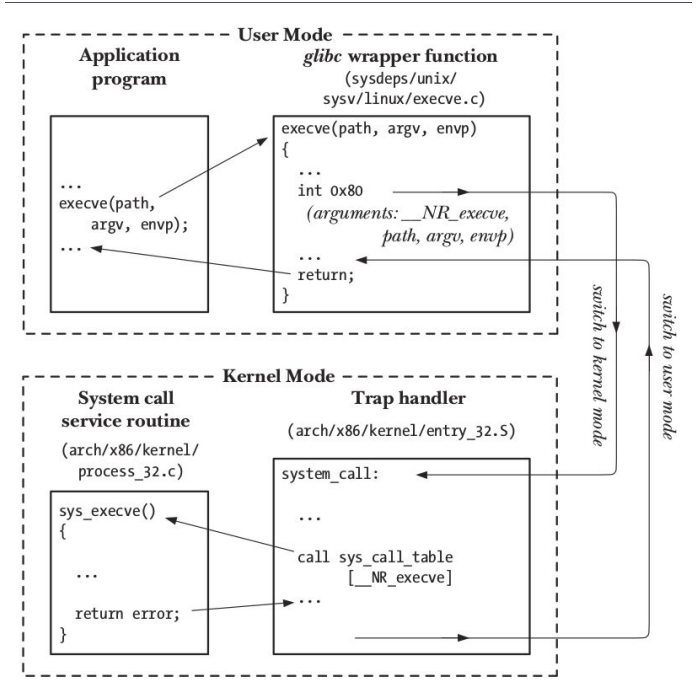
```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.
5.
6.  void func(char ** argv)
7.  {
8.     char arr[2];
9.     strcpy(arr, argv[1]);
10.            printf("Input: %s\n", arr );
11.    return;
12. }
13.
14. int main(int argc, char *argv[])
15. {
16.    func(argv);
17.    return 0;
18. }
19.
```

Part 2B - Advance programming C on Linux (2 sessions)

- **Main contents:**
  - System programming concepts: System Calls, Library Functions, File I/O concept, Executable and Linkable Format, The /proc File System.
  - Dynamic Memory Allocation
  - Signal
  - Multi process and Multi thread
  - IPC:
    - Pipe
    - Share memory
    - Socket programming
- **Q&A :**

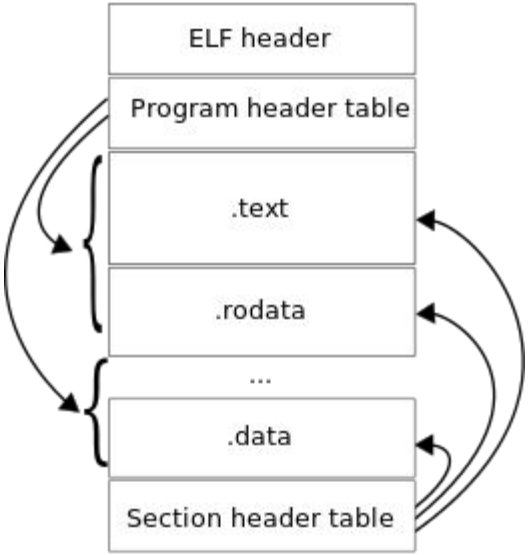
System programming concepts:

- **System Calls:**
  - Controlled entry point into the kernel
  - Changes the processor state from user mode to kernel mode
  - The set of system calls is fixed, each system call is identified by a unique number
  - Transfer specify information from user space to to kernel space and vice versa
- **Library Functions:**
  - A library function is simply one of the multitude of functions that constitutes the standard C library
  - The GNU C Library: provides the core libraries for the GNU system and GNU/Linux systems
  - Can be split into two group:
    - don’t make any use of system calls
    - System call wrappers



System programming concepts:

- **File I/O model:**
  - Each process has its own set of file descriptors
  - File descriptors are used to refer to all types of open files, including pipes, FIFOs, sockets, terminals, devices, and regular files
  - universality of I/O: four system calls—open(), read(), write(), and close()—are used to perform I/O on all types of files
- **Executable and Linkable Format:**
  - In Linux: Program is file with Executable and Linkable Format (ELF, formerly called Extensible Linking Format).
  - Executable and Linkable Format is a common standard file format for executables, object code, shared libraries, and core dumps
  - Objdump command is used to provide thorough information on object file.



System programming concepts:

- **The /proc file system:**
  - isn't associated with a hardware device such as a disk drive
  - is a window into the running Linux kernel
  - provides an easy mechanism for viewing and changing various system attributes
  - /proc/PID, where PID is a process ID, allows us to view information about each process running on the system
  - contents of /proc files are generally in human-readable text form
  - Information about a Process:
    - The **/proc/PID/fd** directory contains one symbolic link for each file descriptor that the process has open
    - The **/proc/PID/task** directory: contains a subdirectory named /proc/PID/task/TID, where TID is the thread ID of the thread.
    - The **/proc/PID/status** : contains process status in human readable form
    - The **/proc/self** any process can access its own /proc/PID directory using the symbolic link /proc/self
    - ....

Dynamic Memory Allocation:

- Kernel manage memory of process via virtual address space
- In 32-bit mode, each process has 4GB memory addresses (not memory)
- We can track memory of process via **/proc/<PID>/maps**, this is all addresses that it has.
- If a process tries to access an address that isn't existed in that map, it receives a SIGSEGV signal (Segmentation fault).
- Memory Layout of a Process (see the picture):
  - The text segment contains the machine-language instructions
  - The initialized data segment contains global and static variables that are explicitly initialized
  - The uninitialized data segment contains global and static variables that are not explicitly initialized
  - The stack is a dynamically growing and shrinking segment
  - The heap is an area from which memory (for variables) can be dynamically allocated at run time. The top end of the heap is called the **program break**.

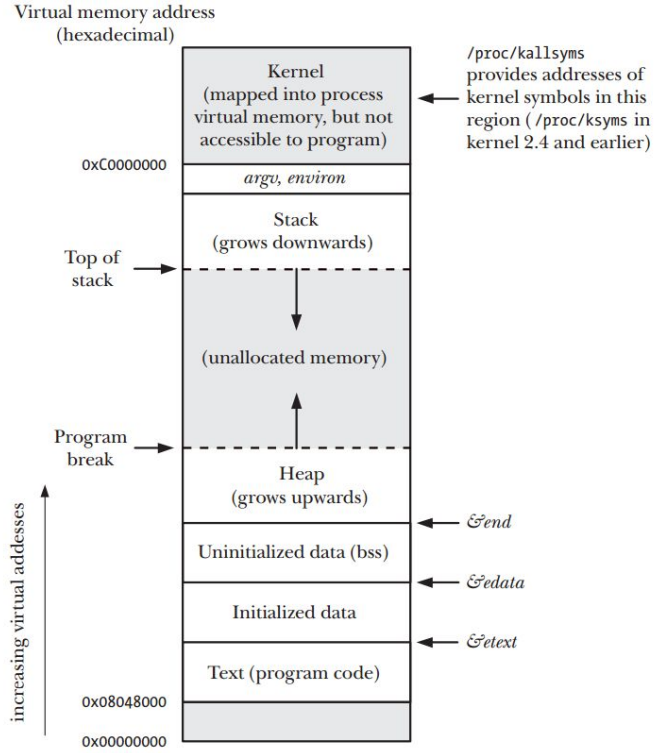


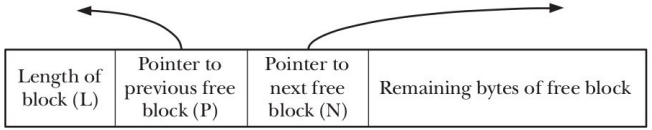
Figure 6-1: Typical memory layout of a process on Linux/x86-32

Dynamic Memory Allocation:

- Resizing the heap (i.e., allocating or deallocating memory) is actually as simple as telling the kernel to adjust its idea of where the process's program break is
- Initially, the program break lies just past the end of the uninitialized data segment
- On Linux: brk() and sbrk() are used to change program break (these functions should not be used directly).
- sbrk(0) returns the current setting of the program break without changing it.
- upper limit of program break can be set depends: resource limit, ocation of memory mappings, shared memory segments, and shared libraries
- In general, C programs use the malloc family of functions to allocate and deallocate memory on the heap:
    - standardized as part of the C language.
    - easier to use in threaded programs.
    - provide a simple interface that allows memory to be allocated in small units
    - allow us to arbitrarily deallocate blocks of memory, which are maintained on a free list and recycled in future calls to allocate memory

Dynamic Memory Allocation:

- Implementation of malloc and free (theory)
  - The malloc library maintains the list of free memory blocks
  - malloc:
    - scans the list of free memory blocks, find one whose size is larger than or equal to its requirements
    - If the block is exactly the right size, then it is returned to the caller
    - If it is larger, then it is split
    - no block is large enough, calls sbrk() to allocate more memory, putting the excess memory onto the free list
  - free:
    - Each allocation block has extra bytes to hold an integer containing the size of the block.
    - free() uses the bytes of the block itself in order to add the block to the list

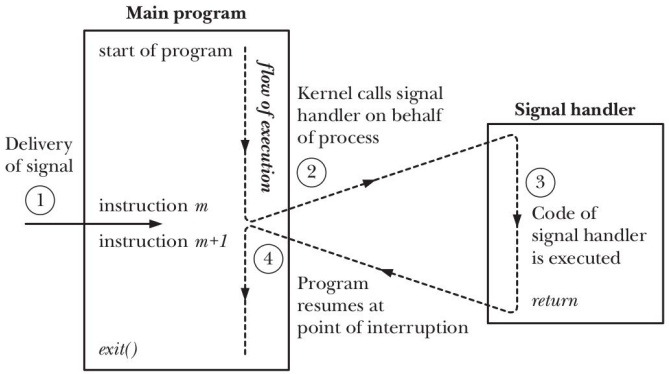


Signal:

- A signal is a notification to a process that an event has occurred
- Each signal is defined as a unique (small) integer. These integers are defined in <signal.h>
- Kernel to generate a signal for a process are the following:
  - A hardware exception occurred
  - The user typed one of the terminal special characters
  - A software event occurred. For example, input became available on a file descriptor, the terminal window was resized, ...
- It is also possible for a process to send a signal to itself.
- Some common standard signal:
  - SIGABRT: A process is sent this signal when it calls the abort() function
  - SIGALRM: The kernel generates this signal upon the expiration of a real-time timer
  - SIGCHLD: This signal is sent to a parent process when one of its children terminates
  - SIGSTOP: This is the sure stop signal. It can't be blocked or ignored
  - SIGCONT: When sent to a stopped process, this signal causes the process to resume
  - SIGHUP: When a terminal disconnect (hangup) occurs
  - SIGINT: terminal interrupt character (usually Control-C)
  - SIGKILL: This is the sure kill signal. It can't be blocked or ignored
  - SIGQUIT: The quit character (usually Control-\) on the keyboard
  - SIGTERM: This is the standard signal used for terminating a process and is the default signal sent by the kill and killall commands

Signal:

- A signal handler is a function that is called when a specified signal is delivered to a process
- Signal handler may interrupt the main program flow at any time, execution of the program resumes at the point where the handler interrupted it
- Each kind of signal has its own default action
- For simple cases, we can change the default action of a signal by using **sighandler_t signal(int** *signum*, **sighandler_t** *handler*) system call.
- sigaction() system is an alternative to signal() for setting the disposition of a signal. It provides greater flexibility.
- We can send a signal to process using the kill() system call



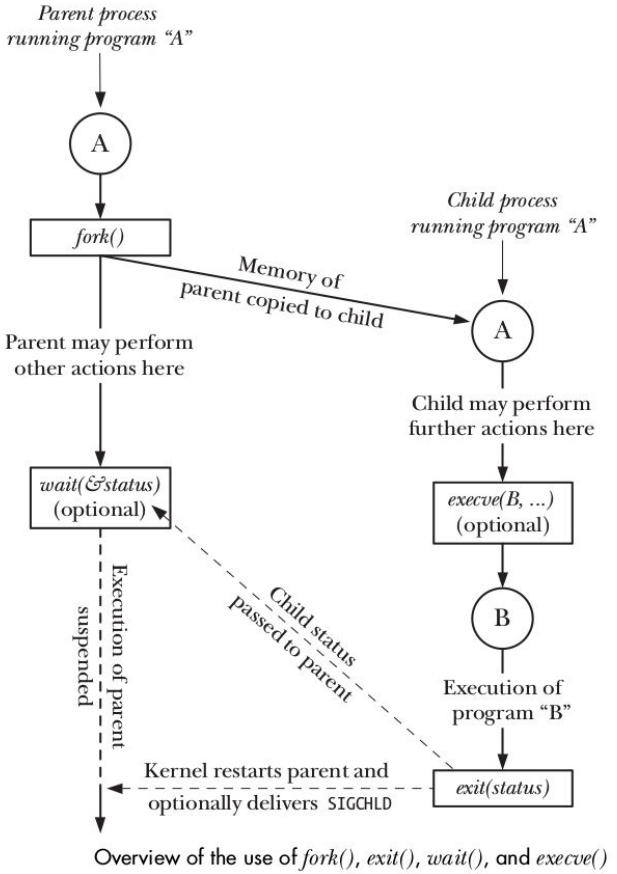
Example: see source code here, this example establishes the same handler for two different signals

```
$ ./intquit
Type Control-C
Caught SIGINT (1)
Type Control-C again
Caught SIGINT (2)
and again
Caught SIGINT (3)
Type Control-\
Caught SIGQUIT - that's all folks!
```

Multi process:

- Creating new process by System call fork
  - For parent fork return child's process id
  - For child fork return 0
  - For the problem fork return -1

```
pid_t childPid;             /* Used in parent after successful fork()
                               to record PID of child */

switch (childPid = fork()) {
case -1:                    /* fork() failed */
    /* Handle error */

case 0:                     /* Child of successful fork() comes here */
    /* Perform actions specific to child */

default:                    /* Parent comes here after successful fork() */
    /* Perform actions specific to parent */
}
```



Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

Multi process:

The child process has same program text but they have separate copied of stack, data, heap.

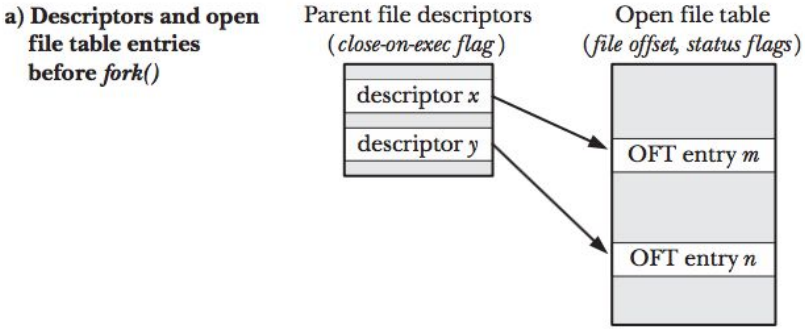
```
$ ./t_fork
PID=28557 (child)  idata=333 istack=666
PID=28556 (parent) idata=111 istack=222
```

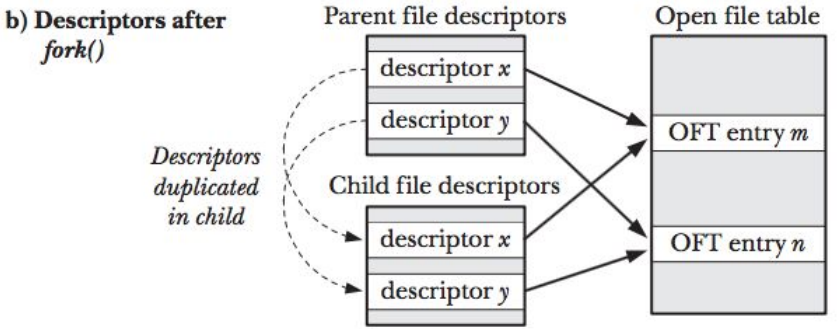
```
#include "tlpi_hdr.h"

static int idata = 111;             /* Allocated in data segment */

int
main(int argc, char *argv[])
{
    int istack = 222;               /* Allocated in stack segment */
    pid_t childPid;

    switch (childPid = fork()) {
    case -1:
        errExit("fork");

    case 0:
        idata *= 3;
        istack *= 3;
        break;

    default:
        sleep(3);                   /* Give child a chance to execute */
        break;
    }

    /* Both parent and child come here */

    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
            (childPid == 0) ? "(child) " : "(parent)", idata, istack);

    exit(EXIT_SUCCESS);
}
```

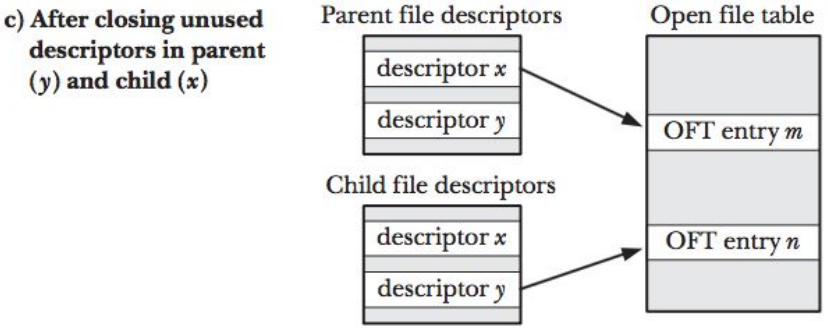
procexec/t_fork.c

Multi process:

The child receives duplicate of all of parent file descriptors.







Example: see source code here

```
$ ./fork_file_sharing
File offset before fork(): 0
O_APPEND flag before fork() is: off
Child has exited
File offset in parent: 1000
O_APPEND flag in parent is: on
```

Multi process:

It’s indeterminate which process - the parent or the child - next has access to the CPU.

If either process need to wait for other to complete an action, then they need an **Inter-Process Communication** (IPC) mechanism for this purpose.

Example: see source code here, this example uses signal

```
$ ./fork_sig_sync
[17:59:02 5173] Child started - doing some work
[17:59:02 5172] Parent about to wait for signal
[17:59:04 5173] Child about to signal parent
[17:59:04 5172] Parent got signal
```

Multi process:

It’s indeterminate which process - the parent or the child - next has access to the CPU.

If either process need to wait for other to complete an action, then they need an **Inter-Process Communication** (IPC) mechanism for this purpose.

Example: see source code here, this example uses signal to synchronize the parent and its children

```
$ ./fork_sig_sync
[17:59:02 5173] Child started - doing some work
[17:59:02 5172] Parent about to wait for signal
[17:59:04 5173] Child about to signal parent
[17:59:04 5172] Parent got signal
```

Multi process:

Sometime, the parent process needs to know when one of its child process change state (terminated or stopped)

The *wait()* system call waits for one of its child to terminate:

- If no child of the calling process has yet terminated, blocks until one of the children terminates
- If status is not NULL, information about how the child terminated is returned in the integer to which status points
- As its function result, wait() returns the process ID of the child that has terminated.

For advantage, the waitpid() system call let us obtain information about the child whose state has changed.
Example: see source code here, this example for creating and waiting for multiple children

```
$ ./multi_wait 7 1 4
[13:41:00] child 1 started with PID 21835, sleeping 7 seconds
[13:41:00] child 2 started with PID 21836, sleeping 1 seconds
[13:41:00] child 3 started with PID 21837, sleeping 4 seconds
[13:41:01] wait() returned child PID 21836 (numDead=1)
[13:41:04] wait() returned child PID 21837 (numDead=2)
[13:41:07] wait() returned child PID 21835 (numDead=3)
No more children - bye!
```

Multi process:

- If the parent died before its children, its children (***orphaned***) will be adopted by **init**, the ancestor of all processes, whose process ID is 1
- The ***zombie*** is the child terminates and its parent has not performed a ***wait()***. Zombie process can't be killed by a signal, not even the (silver bullet) SIGKILL. When the parent does perform a wait(), the kernel removes the zombie

Example: see source code here, this example for creating zombie process and try to kill it

```
$ ./make_zombie
Parent PID=1013
Child (PID=1014) exiting
 1013 pts/4    00:00:00 make_zombie                    Output from ps(1)
 1014 pts/4    00:00:00 make_zombie <defunct>
After sending SIGKILL to make_zombie (PID=1014):
 1013 pts/4    00:00:00 make_zombie                    Output from ps(1)
 1014 pts/4    00:00:00 make_zombie <defunct>
```

Multi process:

The execve() system call loads a new program into a process's memory

- The old program is discarded and the process's stack, data, and heap are replaced by those of the new program
- The new program commences execution at its main() function
- The process ID of the process remains the same

Example: see source code here and here, This first program creates an argument list and an environment for a new program, and then calls execve(). This second program simply displays its command-line arguments and environment

```
$ ./t_execve ./envargs
argv[0] = envargs
argv[1] = hello world
argv[2] = goodbye
environ: GREET=salut
environ: BYE=adieu
```

All of the output is printed by envargs

list

Multi process:

All file descriptors opened by a program that calls exec() remain open across the exec() and are available for use by the new program. The shell takes advantage of this feature to handle I/O redirection.

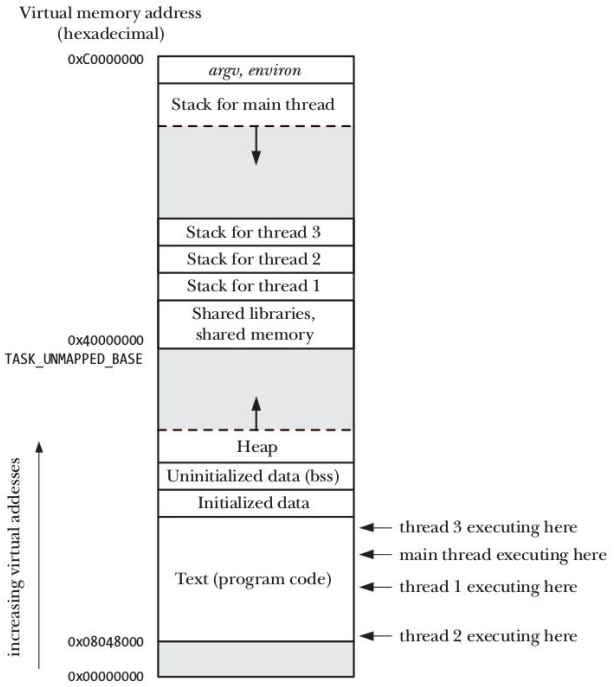
The close-on-exec flag (FD_CLOEXEC) is used when you want to ensure that certain file descriptors are closed before exec()

Example: see source code here, This example demonstrates the manipulation of the close-on-exec flag. If we provide any arguments, this program first sets the close-on-exec flag for standard output and then execs the ls program

```
$ ./closeonexec                    Exec ls without closing standard output
-rwxr-xr-x   1 mtk    users    28098 Jun 15 13:59 closeonexec
$ ./closeonexec n                  Sets close-on-exec flag for standard output
ls: write error: Bad file descriptor
```

Multi threads:

- Threads are a mechanism that permits an application to perform multiple tasks concurrently.
- Almost process's attributes are shared between threads in process but not for following:
  - Thread ID
  - Signal mask
  - Thread-specific data
  - The errno variable
  - Stack
- Thread creation is faster than process creation, because many of the attributes that must be duplicated in a child created by fork()
- Programs use the Pthreads API must be compiled with the cc –pthread
- The pthread_create() function creates a new thread.
- The pthread_exit(void * retval) function terminates the calling thread, *retval* should not be located on the thread's stack.
- If the main thread calls pthread_exit(), then the other threads continue to execute.



Multi threads:

- Threads are a mechanism that permits an application to perform multiple tasks concurrently.
- Almost process's attributes are shared between threads in process but not for following:
  - Thread ID
  - Signal mask
  - Thread-specific data
  - The errno variable
  - Stack
  - .....
- Thread creation is faster than process creation
- Programs use the Pthreads API must be compiled with the cc –pthread
- The pthread_create() function creates a new thread.
- The pthread_exit(void * retval) function terminates the calling thread, *retval* should not be located on the thread's stack. If the main thread calls pthread_exit(), then the other threads continue to execute.
- The pthread_join() function waits for the thread identified by *thread* to terminate.

Example: see source code here, this example creates another thread and then joins with it.

```
$ ./simple_thread
Message from main()
Hello world
Thread returned 12
```

Multi threads:

- By making a call to pthread_detach(), we don't care about the thread's return status.
- Each thread within a process is uniquely identified by a thread ID. This ID is returned to the caller of pthread_create() and pthread_self().
- Advantages of multithreaded:
  - Sharing data between threads is easy
  - Thread creation is faster than process creation
- Disadvantages of multithreaded
  - We need to ensure the functions we call are thread-safe or are called in a thread-safe manner.
  - A bug in one thread can damage all of the threads in the process.
  - Each thread is competing for use of the finite virtual address space of the host process

Multi threads:

- The term *critical section* is used to refer to a section of code that accesses a shared resource and whose execution should be *atomic*
- To avoid the problems that can occur when threads try to update a shared variable, we must use a *mutex* (short for mutual exclusion) to ensure that only one thread at a time can access the variable.
- A *mutex* has two states: *locked* and *unlocked*. When a thread locks a mutex, it becomes the owner of that mutex.
- Only the *mutex owner* can unlock the mutex.
- A mutex is a variable of the type pthread_mutex_t. mutex must always be initialized. For a statically allocated mutex, we can do this by assigning it the value PTHREAD_MUTEX_INITIALIZE
- To lock and unlock a mutex, we use the pthread_mutex_lock() and pthread_mutex_unlock() functions
- If the mutex is currently unlocked, *pthread_mutex_lock*() locks the mutex and returns immediately
- If the mutex is currently locked by another thread, then *pthread_mutex_lock*() blocks until the mutex is unlocked.
- On Linux, the thread deadlocks by default If the mutex is currently locked by itself.
- The *pthread_mutex_unlock*() function unlocks a mutex previously locked by the calling thread. It is an error to unlock a mutex that is not currently locked, or to unlock a mutex that is locked by another thread

Multi threads:

Example: see source code here, this example creates two threads, each of which executes the same function. The function executes a loop that repeatedly increments a global variable, glob, by copying glob into the local variable loc, incrementing loc, and copying loc back to glob. The value of glob should have been 20 million but it is not.

```
$ ./thread_incr 10000000
glob = 10880429
$ ./thread_incr 10000000
glob = 13493953
```

Example: see source code here, this example uses a mutex to protect access to the global variable glob. We see that glob is always reliably incremented

```
$ ./thread_incr_mutex 10000000
glob = 20000000
```

Multi threads:

- **Condition variables** allow threads to sleep waiting for a condition
- Declaring a condition variable - variable whose type is ***pthread_cond_t***.
- Threads that enable a condition (making a condition true) must "wake up" threads waiting on the condition.
- The condition must be associated with a mutex which is used to protect the shared data tested by the condition.
- The mutex must be locked when a thread waits on or enables the condition.
- For a statically allocated condition variable, this is done by assigning it the value PTHREAD_COND_INITIALIZER
- The pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr) function is used to dynamically initialize a condition variable
- The pthread_cond_wait(pthread_cond_t *restrict cond,pthread_mutex_t *restrict mutex) function blocks a thread until the condition variable cond is signaled
- The pthread_cond_signal(pthread_cond_t *cond) unblock at least one of the threads that are blocked on the specified condition variable *cond*

Multi threads:

Example1: see source code here, this example doesn't use condition variables serves to demonstrate why they are useful.

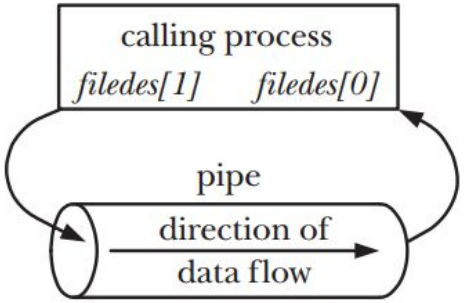
```
time ./prod_no_condvar 5 5 5
T=1: numConsumed=1
T=1: numConsumed=2
….
T=5: numConsumed=14
T=5: numConsumed=15

real    0m5.001s
user    0m5.000s
sys     0m0.000s
```

Example2: see source code here, this example use condition variables to optimize CPU usage.

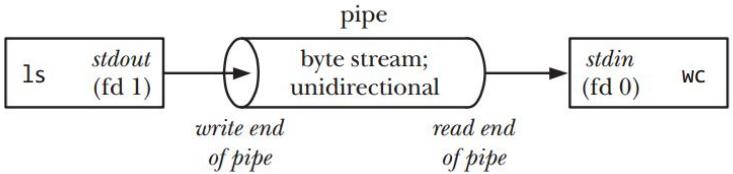
```
time ./prod_condvar 5 5 5
T=1: numConsumed=1
T=1: numConsumed=2
….
T=5: numConsumed=14
T=5: numConsumed=15

real    0m5.001s
user    0m0.000s
sys     0m0.000s
```

IPC: PIPE and FIFO

- A pipe is a byte stream. The data passes through the pipe sequentially.
- Pipes are unidirectional, data can travel only in one direction through a pipe
- Create PIPE via ***pipe(int filedes[2])*** system call
- Once written to the write end of a pipe, data is immediately available to be read from the read end
- Writes of up to PIPE_BUF (4096 for Linux) bytes are guaranteed to be atomic
- A pipe is simply a buffer maintained in kernel memory. This buffer has a maximum capacity. Once a pipe is full, further writes to the pipe block until the reader removes some data from the pipe
- Every user of the shell is familiar with the use of pipes in commands.

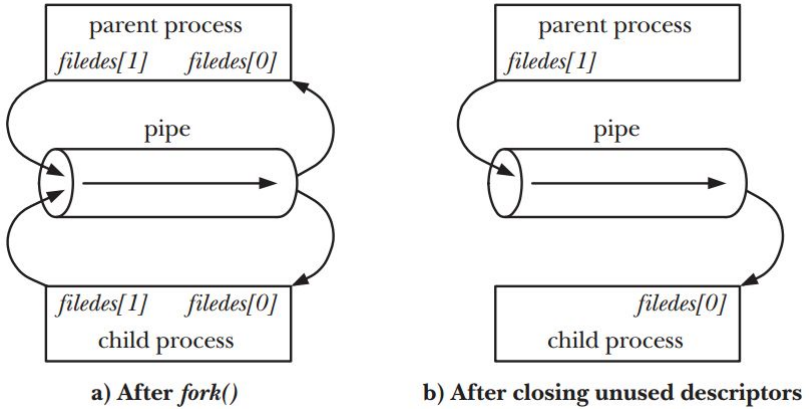


Example: ***$ ls | wc -l***



IPC: PIPE and FIFO

- A pipe has few uses within a single process
- We use a pipe to allow communication between two processes
- If the parent is to send data to the child, then it would close its read descriptor for the pipe (filedes[0])
- While the child would close its write descriptor for the pipe (filedes[1])



a) After *fork()*

b) After closing unused descriptors

Example: see source code here, this example used a pipe to communicate between a parent and child process. The parent writes its data in a single operation, while the child reads data from the pipe in small blocks.

```
$ ./simple_pipe 'It was a bright cold day in April, '\
'and the clocks were striking thirteen.'
It was a bright cold day in April, and the clocks were striking thirteen.
```

IPC: PIPE and FIFO

- Semantically, a FIFO is similar to a pipe
- The principal difference is that a FIFO has a name within the file system and is opened in the same way as a regular file
- We can create a FIFO from the shell using the mkfifo command: ***$ mkfifo [-m mode] pathname***
- The mkfifo(const char *pathname, mode_t mode) function creates a new FIFO with the given pathname
- The *mode* argument specifies the permissions for the new FIFO
- Once a FIFO has been created, any process can open it.
- Opening a FIFO for reading (the open() O_RDONLY flag) blocks until another process opens the FIFO for writing
- Opening the FIFO for writing blocks until another process opens the FIFO for reading.

IPC: POSIX Shared Memory

- POSIX shared memory allows to us to share a mapped region between unrelated processes without needing to create a corresponding mapped file
- To use a POSIX shared memory object, we perform two steps:
  1. Use the shm_open() function to open an object with a specified name
  2. Pass the file descriptor obtained in the previous step in a call to mmap() that specifies MAP_SHARED in the flags argument
- When a new shared memory object is created, it initially has zero length. We normally call ftruncate() to set the size of the object before calling mmap().
- When a shared memory object is no longer required, it should be removed using shm_unlink()

Example1: Creating a POSIX shared memory object.

```
$ ./pshm_create -c /demo_shm 10000
$ ls -l /dev/shm
total 0
-rw-------    1 mtk      users       10000 Jun 20 11:31 demo_shm
```

Example1: Write and read from POSIX shared memory object.

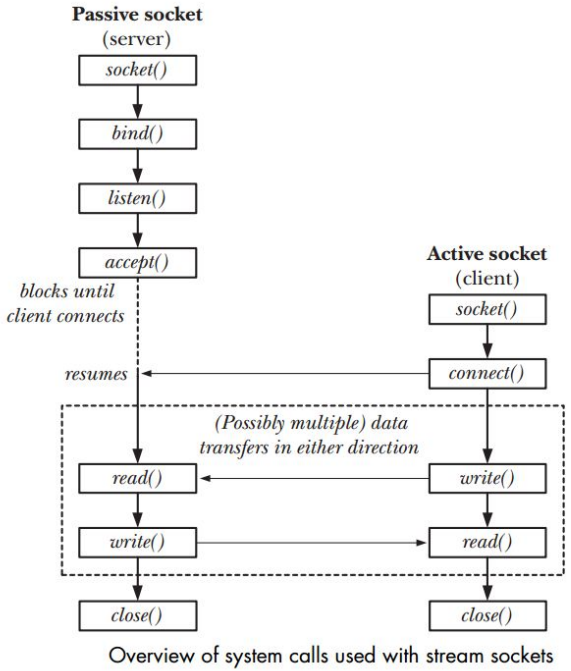
```
$ ./pshm_write /demo_shm 'hello'
$ ls -l /dev/shm                          Check that object has changed in size
total 4
-rw-------    1 mtk      users           5 Jun 21 13:33 demo_shm

$ ./pshm_read /demo_shm
hello
```

IPC: Socket

- Socket allow data to be exchanged between applications, either on the same host (computer) or on different hosts connected by a network
- In a typical client-server scenario, applications communicate using sockets as follows:
  1. Each application creates a socket. A socket is the “apparatus” that allows communication, and both applications require one.
  2. The server binds its socket to a well-known address (name) so that clients can locate it.
- A socket is created using the socket() system call: ***fd = socket(domain, type, protocol);***
- Sockets exist in a communication domain:
  1. The UNIX (AF_UNIX) domain allows communication between applications on the same host
  2. The IPv4 (AF_INET) domain
  3. The IPv6 (AF_INET6) domain
- Stream sockets (SOCK_STREAM) provide a reliable, bidirectional, byte-stream communication channel
- Datagram sockets (SOCK_DGRAM) allow data to be exchanged in the form of messages, data transmission is not reliable, doesn’t need to be connected to another socket in order to be used.

IPC: Socket

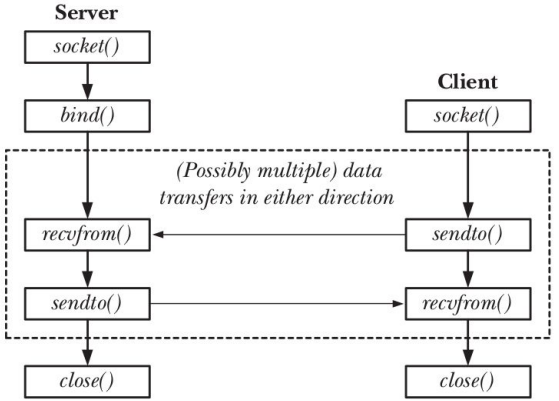
- The key socket system calls are the following:
  1. The socket() system call creates a new socket.
  2. The bind() system call binds a socket to an address. Usually, a server employs this call to bind its socket to a well-known address so that clients can locate the socket.
  3. The listen() system call allows a stream socket to accept incoming connections from other sockets.
  4. The accept() system call accepts a connection from a peer application on a listening stream socket, and optionally returns the address of the peer socket.
  5. The connect() system call establishes a connection with another socket
- Socket I/O can be performed using the conventional read() and write() system calls, or using a range of socket-specific system calls (e.g., send(), recv(), sendto(), and recvfrom()).
- System calls such as bind() are generic to all socket domains, In order to permit this, the sockets API defines a generic address structure: struct sockaddr



Overview of system calls used with stream sockets

IPC: Socket

- The recvfrom() and sendto() system calls receive and send datagrams on a datagram socket.
- Even though datagram sockets are connectionless, the connect() system call serves a purpose when applied to datagram sockets. The term *connected datagram socket* is applied to such a socket. After a datagram socket has been connected:
    1. Datagrams can be sent through the socket using write() (or send()) and are automatically sent to the same peer socket. As with sendto(), each write() call results in a separate datagram.
    2. Only datagrams sent by the peer socket may be read on the socket.



IPC: Socket

- An IPv4 socket address is stored in a sockaddr_in structure
  - *sin_family* is always set to AF_INET .
  - The *sin_port* and *sin_addr* fields are the port number and the IP address, both in network byte order.
- An IPv6 socket address is stored in a sockaddr_in6 structure, Like an IPv4 address, an IPv6 socket address includes an IP address plus a port number. The difference is that an IPv6 address is 128 bits instead of 32 bits
- Computers represent IP addresses and port numbers in binary. However, humans find names easier to remember than numbers. A *hostname* is the symbolic identifier for a system that is connected to a network (possibly with multiple IP addresses). A *service name* is the symbolic representation of a port number.
- The inet_aton() and inet_ntoa() functions convert an IPv4 address in dotted-decimal notation to binary and vice versa
- Given a hostname and a service name, getaddrinfo() returns a set of structures containing the corresponding binary IP address(es) and port number

IPC: Socket

Client-Server Example (Datagram Sockets)

Server

```
sfd = socket(AF_INET6, SOCK_DGRAM, 0);
if (sfd == -1)
   errExit("socket");

memset(&svaddr, 0, sizeof(struct sockaddr_in6));
svaddr.sin6_family = AF_INET6;
svaddr.sin6_addr = in6addr_any;              /* Wildcard address */
svaddr.sin6_port = htons(PORT_NUM);

if (bind(sfd, (struct sockaddr *) &svaddr,
        sizeof(struct sockaddr_in6)) == -1)
   errExit("bind");

/* Receive messages */
    numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                (struct sockaddr *) &claddr, &len);

    /* sent the message */

    if (sendto(sfd, buf, numBytes, 0, (struct sockaddr *) &claddr, len) !=
         numBytes)
       fatal("sendto");
```

Client

```
/* Create a datagram socket; send to an address in the IPv6 domain */

sfd = socket(AF_INET6, SOCK_DGRAM, 0);    /* Create client socket */
if (sfd == -1)
   errExit("socket"
memset(&svaddr, 0, sizeof(struct sockaddr_in6));
svaddr.sin6_family = AF_INET6;
svaddr.sin6_port = htons(PORT_NUM);
if (inet_pton(AF_INET6, argv[1], &svaddr.sin6_addr) <= 0)
   fatal("inet_pton failed for address '%s'", argv[1]);

/* Send messages to server; echo responses on stdout */

if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
        sizeof(struct sockaddr_in6)) != msgLen)
      fatal("sendto");

    numBytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
    if (numBytes == -1)
      errExit("recvfrom");
```

```
$ ./i6d_ucase_sv &
[1] 31047
$ ./i6d_ucase_cl ::1 ciao
Server received 4 bytes from (::1, 32770)
Response 1: CIAO
```

Send to server on local host

IPC: Socket

Client-Server Example (Stream Sockets)

Server program

- Initialize the server's sequence number either to 1 or to the value supplied in the optional command-line argument
- Ignore the SIGPIPE signal
- Call getaddrinfo() to obtain a set of socket address structures for a TCP socket that uses the port number PORT_NUM .
- Enter a loop that iterates through the socket address structures returned by the previous step. The loop terminates when the program finds an address structure that can be used to successfully create and bind a socket
- Set the SO_REUSEADDR option for the socket created in the previous step
- Mark the socket as a listening socket
- Accept a new connection
- Read the client's message which consists of a newline-terminated string specifying how many sequence numbers the client wants. The server converts this string to an integer and stores it in the variable reqLen
- Send the current value of the sequence number (seqNum) back to the client, encoding it as a newline-terminated string. The client can assume that it has been allocated all of the sequence numbers in the range seqNum to (seqNum + reqLen – 1).
- Update the value of the server's sequence number by adding reqLen to seqNum

Client program

- Call getaddrinfo() to obtain a set of socket address structures suitable for connecting to a TCP server bound to the specified host.
- Enter a loop that iterates through the socket address structures returned by the previous step, until the client finds one that can be used to successfully create e and connect a socket to the server
- Send an integer specifying the length of the client's desired sequence. This integer is sent as a newline-terminated string.
- Read the sequence number sent back by the server (which is likewise a newline terminated string) and print it on standard output

```
$ ./is_seqnum_sv &
[1] 4075
$ ./is_seqnum_cl localhost                Client 1: requests 1 sequence number
Connection from (localhost, 33273)        Server displays client address + port
Sequence number: 0                        Client displays returned sequence number
$ ./is_seqnum_cl localhost 10             Client 2: requests 10 sequence numbers
Connection from (localhost, 33274)
Sequence number: 1
$ ./is_seqnum_cl localhost                Client 3: requests 1 sequence number
Connection from (localhost, 33275)
Sequence number: 11
```