Advanced Linux Programming

AUT @CEIT - 10th Linux Festival

Contents

Compiling with GCC

- Turn human-readable code into machine-readable object code
- GCC

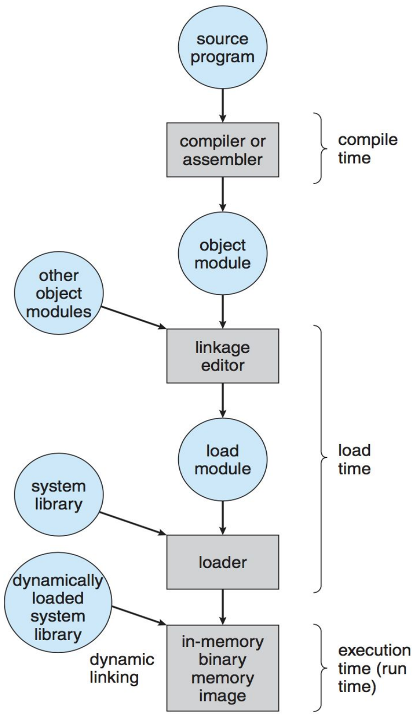

Figure 8.3 Multistep processing of a user program.

Compiling with GCC

- GCC flags
  - -c : compile (output is object file)
  - -I : header files path
  - Compile macros
    - NDEBUG : remove assertion
    - O : optimization level

```
$ g++ -c main.o
$ g++ -c -I ../include reciprocal.cpp
$ g++ -c -D NDEBUG
$ g++ -c O2 reciprocal.cpp
$ info gcc
```

Linking object files

- GCC flags
  - -o : linking files (output is runnable file)
  - -l : define libraries to link (default paths like /lib and /usr/lib)
  - -L : tell linker to search for other directories

```
$ g++ -o reciprocal main.o reciprocal.o
$ ./reciprocal 7
$ g++ -o reciprocal main.o reciprocal.o -lpam  (library: libpam.a)
$ g++ -o reciprocal main.o reciprocal.o -L/usr/local/lib/pam -lpam
$ gcc -o app app.o -L. -ltest
```

Automating build with Make

- Like how IDE builds your project automatically
- You tell make what *targets* to build
- Give *rules* explaining how to build them
- And also define *dependencies*
- And then just type *make*
- Comes handy when project gets bigger
- Easier to change
- $(CFLAGS) is make variable
-

Man pages

- Learn from manuals !!
- Divided into numbered sections:
  - (1) User commands
  - (2) System calls
  - (3) Standard Library Functions
  - (8) System/administrative commands
- Whatis: display all man pages for command
- Man -k : perform keyword search on summary lines

```
$ man sleep
$ man 3 sleep
$ man -k sleep
```

Writing Good GNU/Linux Software

Interaction with environment

- Special main function
- The argument list
- Work with argc and argv (example 2-1)
-

Standard IO

- ➢ stdin : standard input -> printf -> 0
- ➢ stdout : standard output -> scanf -> 1
- ➢ stderr: error output -> fprintf(stderr, (“error: …”)); -> 2
- ➢ Available to Unix commands with file descriptor

•

```
$ program > output_file 2>&1
$ program 2&1 | filter
```

Buffered output

- stdout is buffered
- fflush (stdout)

- stderr is not buffered

```
while (1) {
    printf(“.”);
    sleep(1);
}
```

```
while (1) {
    fprintf(stderr, “.”);
    sleep(1);
}
```

Program exit code

- When a program ends, it indicated its status with returning a small int
- Zero means successful
- echo $?
- return(0)

The environment

- Collection of variable/value pair
  - USER : your username
  - HOME: path to home directory
  - PATH: colon-seperated list for command directories
  - **printenv**
  - Use **export** to export a shell variable
- Functions in stdlib
  - **getenv:** access variable
  - **setenv:** set variable
  - **unsetenv:** clear variable

The environment

- Let’s enumerate all variables in the environment:
  - Access a special global variable called **environ**
  - Type: char**
  - NULL terminated array of pointers to character strings
  - Each string contains one environment variable in the format VARIABLE=value
  - Let’s go to code :)

The environment

Listing 2.3 (*print-env.c*) **Printing the Execution Environment**

```
#include <stdio.h>

/* The ENVIRON variable contains the environment.  */
extern char** environ;

int main ()
{
  char** var;
  for (var = environ; *var != NULL; ++var)
    printf ("%s\n", *var);
  return 0;
}
```

The environment

- Environment variables are commonly used to communicate configuration

  To programs
- Go to client example

Coding Defensively !!

Using assert

- Bugs or unexpected errors should cause the programs to fail dramatically, as early as possible
- Or they don’t show them until app is under user hand
- One of our tools is standard C **assert** macro
- The argument to assert is boolean
- Program is terminated if it’s false
- Printing line of code and message
-

Using assert

- ➢ runtime checks like asserts can impose a significant performance penalty
- ➢ Compile your production code with NDEBUG macro
- ➢ Appearances o **assert** macro will be preprocessed away
- ➢ Do not assign variables or call functions inside assert body

```
for (i=0; i < 100; i++) {
    assert(DoSomething() == 0);
}
```

```
for (i=0; i < 100; i++) {
    int status = do_something();
    assert(status == 0);
}
```

Using assert

- ➢ Check against NULL pointers
- ➢ Check conditions on function parameter values
  - ○ Helps you find misuses of function
  - ○ Makes it clear for someone reading the code

```
Assert (pointer != NULL);
```

```
Assertion ‘pointer != ((void *)0)’ failed.

Or

Segmentation fault (core dumped)
```

System call failures

- System calls can fail, and make your program crash !
    - Out of resource (too many open files, memory etc.)
    - Permission denied
    - Invalid arguments to syscall
    - Faulty device (disk not inserted!)

Error codes from syscalls

- Mostly return 0 for success and non-zero for failure
- Use a special variable called **errno** to store additional information
- Value of errno will be replaced, next time you make a syscall
- Error values are integers
- Possible values are given by macros
- Starting by “E” (like EACCESS and EINVAL)
- Include <errno.h> for errno values

Error codes from syscalls

- ➢ Use **strerror** function to get string description of errno
- ➢ Include <string.h> with strerror
- ➢ Use **perror** to print description directly to stderr
- ➢ Include <stdio.h> with perror

```
fd = open ("inputfile.txt", O_RDONLY);
if (fd == -1) {
  /* The open failed.  Print an error message and exit.  */
  fprintf (stderr, "error opening file: %s\n", strerror (errno));
  exit (1);
}
```

Error codes from syscalls

- Depending on your program and the nature of syscall:
  - Print an error message
  - Cancel operation
  - Abort the program
  - Try again
  - Ignore the error !
- EINTR
  - Blocking functions like read, select, sleep
  - If a program receives a signal while blocked, the call will return without completing
  - In this situation errno is set to EINTR
  - Retry !

Error codes from syscalls

- ➢ Take a look at syscall example

- ➢ When we detect a bug, we exit using abort() or assert()
- ➢ Causes a core file to be generated
- ➢ This can be useful for post-mortem debugging

- ➢ For other unrecoverable errors like out of memory we exit with non-zero value
- ➢ Because core file isn’t useful

Errors and resource allocation

- Often when a system call fails, it’s appropriate to just cancel the operation instead of exiting program (we may recover)
- Return from function, passing a return function to the caller, indicating error
- Remember to deallocate the resources in the function
  - Memory
  - File descriptors
  - Temp files
  - Synchronization objects etc
- Go to resource example

Errors and resource allocation

- Linux cleans up allocated resources when a program exits
- So it’s unnecessary to do so before exit()
- But, remember to deallocated shared resources
  Like temp files and shared memory

Writing and Using Libraries

- All programs are linked against one or more libraries
- Any program that uses a C function is linked against C runtime library
- Two methods:
  - Static Link:
    - Bigger program
    - harder to upgrade
    - Easier to deploy
  - Dynamic Link:
    - Smaller
    - Easier to upgrade
    - Harder to deploy

Archive (static library)

- A collection of object files stored as a single file
- Linker searches archive for needed object files and links directly to program
- Create archive using **ar** command
- Traditionally used .a extension
- **cr** flag tells **ar** to create archive

```
$ ar cr libtest.a test1.o test2.o
```

Archive (static library)

- Put archives at the end of gcc command

```
Int f(
     Return 3;
}
```

```
Int main(
     Return f();
}
```

```
$ gcc -o app -L. -ltest app.o
$ gcc -o app app.o -L. -ltest
```

Shared Libraries

- Again a grouping of object files
- The linked binary does not contain the actual code in library
- But a reference to shared library
- Several programs in system can use library (reference)
- Objects composing the shared library are merged into one object file
- So the program that is linked includes all of the code (instead of necessary parts)

Shared Libraries

- ➢ To create shared libraries, you must compile the objects using -fPIC option of compiler
- ➢ Then combine object files into a shared library

```
$ gcc -c -fPIC test1.c
$ gcc -shared -fPIC -o libtest.so test1.o test2.o
$ gcc -o app app.o -L. -ltest (for libtest.so in current directory)
```

Shared Libraries

- What if both libtest.a and libtest.so are available ?
  - First -L directories, then default
  - When one is found, stops search
  - If both available, uses shared library
- We can demand static archives using **-static** option
- Use **ldd** command to show linked shared libraries
- *ld-linux.so* is part of linux dynamic linking mechanisms

```
$ gcc -static -o app app.o -L. -ltest
$
```

Using LD_LIBRAR_PATH

- Linker only places the name of shared library (not path)
- When program is run, the system searches /lib and /usr/lib only !!
- One solution is using **-Wl, rpath** option when linking
- Another solution is to set the **LD_LIBRARY_PATH** variable when running the program

```
$ gcc -o app app.o -L. -ltest -Wl,-rpath,/usr/local/lib
```

Library Dependencies

- ➢ One library often depends on another library.

Listing 2.9 (*tifftest.c*) Using *libtiff*

```
#include <stdio.h>
#include <tiffio.h>

int main (int argc, char** argv)
{
  TIFF* tiff;
  tiff = TIFFOpen (argv[1], "r");
  TIFFClose (tiff);
  return 0;
}
```

```
$ gcc -o tifftest tifftest.c -ltiff
$ ldd libtiff
```

Library Dependencies

- Static libraries can not point to other libraries

Listing 2.9 (*tifftest.c*) Using *libtiff*

```
#include <stdio.h>
#include <tiffio.h>

int main (int argc, char** argv)
{
  TIFF* tiff;
  tiff = TIFFOpen (argv[1], "r");
  TIFFClose (tiff);
  return 0;
}
```

```
$ gcc -static -o tifftest tifftest.c -ltiff
$ gcc -static -o tifftest tifftest.c -ltiff -ljpeg -lz
```

Pros and Cons

- Shared library saves space on system
- Users can upgrade the libraries without upgrading all the programs that depend on them.
- It can be a disadvantage
- For example developing mission-critical software
- Upgrading other libraries shouldn’t affect your program
- Virtualized enviornments

Processes

Processes

- A running instance of a program is called a **process**
- Most functions used in this chapter are declared in **<unistd.h>**
- Each process is defined by its unique 16 bit **process id (pid)**
- Each process has a parent process, except:
  - Init process
  - Zombie process
- Arranged in a tree with *init* as root
- Parent process id is called **ppid**

Processes

- Use **pid_t** typedef in C program
- Defined in <sys/types.h>
- **getpid()**: get process id
- **getppid():** get parent process id

Listing 3.1 (*print-pid.c*) **Printing the Process ID**

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

int main ()
{
  printf ("The process ID is %d\n", (int) getpid ());
  printf ("The parent process ID is %d\n", (int) getppid ());
  return 0;
}
```

Viewing active processes

- **ps** command
- By default, shows processes controlled by terminal
- Options:
  - **-e:** display all processes
  - **-o:** what information to show

```
$ ps
$ ps -e -o pid,ppid,command
$ kill $pid
```

Creating processes

- Method 1 :
  - Using **system** function
  - Runs command in standard Bourne shell (/bin/sh)

Listing 3.2 **(*system.c*) Using the *system* Call**

```
#include <stdlib.h>

int main ()
{
  int return_value;
  return_value = system ("ls -l /");
  return return_value;
}
```

Creating processes

- Using **fork** and **exec**
- **fork()**: make a child process that is an exact copy of parent
- **exec():** copy address space of new program

fork

- After fork, both child and parent continue the program from the point that fork was called
- New process has new PID
- One way of distinguishing is calling **getpid**
- Fork function provides different return values for parent and child
- The return value for parent is the PID of the child
- And it’s zero for the child
- No process has PID of zero !!!
- Look at fork example !!

Using the exec family

- Exec family:
  - Containing letter **p** : accept a program name and search for the program in the current directory (**execvp** and **execlp**)
  - Containing letter **v**: accept the argument list as a NULL-terminated array of pointers to strings (**execv, execvp, execve**)
  - Containing letter **l**: accept the argument list as C language's varargs mechanism (**execl, execlp, execle**)
  - Containing letter **e**: accept an additional argument, an array of environmental variables

exec

- Because exec replaces the calling program with one, it never returns, unless an error occurs
- Argument list is passed to program argc and argv
- When a program is invoked from the shell, argv[0] is passed as the name of the program
- When using exec, you should pass the name of the program as the first argument
- Go to fork_exec example :)

Process scheduling

- Linux schedules the processes independently
- You can define a process is more/less important by **niceness** value
- Lower value means more important
- Default is zero
- Only a process with root privileges can run commands with zero values

```
$ nice -n 10 sort input.txt > output.txt
$ renice $value $pid
```

Signals

- Mechanisms for communicating and manipulating processes
- Signals are asynchronous
- Program processes the signal immediately
- Each signal type is referred by its signal number
- When a process receives a signal it decides based on **signal dispostion**
- Each signal has a *default disposition*, which determines what happens to the process if the program does not specify some other behavior
- Program can ignore or call a **signal handler** function
- If signal handler used, in case of signal, the program stops, runs handler and then goes on

Signals

- SIGBUS (bus error), SIGSEGV(segmentation violation) and SIGFPE (floating point exception) may be sent to process that is attempting to perform illegal action
- Default disposition for these signals are terminating process and creating core file
- A process may send signal to another process like SIGTERM and SIGKILL
- Might be used to send command to running process
- Two “user defined” signals are reserved for this purpose: **SIGUSR1** and **SIGUSR2**
- **SIGHUP** signal is sometimes used (also for waking up an idle program)

Signals

- IO operations should be avoided in handlers
- Handler should do the minimum work and return or terminate
- Most times just recording that signal has happened
- Assigning global variable can be dangerous because another signal can happen
- Variable should be of type **sig_atomic_t**
- Go to signal example

Mohammad Karimi

Authors