Lecture 2

Awk

C Compiler: Tools and Compilation

C Libraries: Static and Dynamic

AWK

Introduction to AWK

- Written by Alfred **A**ho, Peter **W**einberger, Brian **K**ernighan in 1977.
- awk is primarily a filter that provides a rich language in which to display and minipulate incoming data
- Whereas grep & Co. allows you to search through a text file and look for something, awk lets you search through a text file and *actually do something* once you’ve found what you’re looking for

awk and C

- awk shares many syntactic similarities with the C programming language (Kernighan was heavily involved in both)
- Whereas a C program requires the program author to open and close files, and move from one line to the next in the input, find and isolate the tokens within a given line, keep track of the total number of lines and the current number of tokens, awk does all this for you automatically
- Therefore, we say that awk is “input-driven”, it must work on lines of input

awk Processing

- awk processes incoming text according to lines which are called *records* and elements within those lines called *fields*.
- awk processes commands called pattern-actions, or rules. If a pattern matches, the associated action is performed
- Actions are enclosed in braces {}
- Patterns, if present, are stated before actions outside of braces
- In an awk rule, either the pattern or the action may be missing, but not both:
  - if the pattern is missing, the action is performed on *every* line of the input
  - if the action is missing, the default action is to print the line out to stdout

awk program structure

- Multiple BEGIN sections (optional)
- Multiple END sections (optional)
- Multiple recursive blocks which will operate on *each* record (line) of the input file

awk Program Flow

- Process optional BEGIN block
- Open the file (either specified during invocation or from STDIN)
- Read each line (record) of the input file and parse records into fields referenced by $*n*
  - $0 denotes the entire record
  - each field is demarked by $1, $2, $3, $4, etc.
- Execute each block defined in the awk program on each record (input line)
- Execute optional END block
- Close the file

awk Patterns

- Patterns may be composed of:
  - /regular expressions/
    - awk '/[2-3]/' five.lines
    - awk '$2 ~ /[2-3]/' five.lines
  - A single expression
    - awk ‘$2 > 3’ five.lines
  - A pair of patterns, separated by a comma indicating a range of records:
    - awk ‘$2 == “2”, $2 == “4”’ five.lines

awk Built-in Variables

- FS: Input field separator (default ‘ ’)
- OFS: Output field separator (default ‘ ’)
- RS: Record Separator (default ‘\n’)
- ARGC: C-style arg count
- ARGV: C-style arg vector (offset 0)
- NF: number of fields in current record
- NR: number of records processed so far
- NOTE: Do NOT put a $ in front of these variables (i.e., don’t say “$NR” but just “NR”)

Example Blocks
What do the following do?

- awk ‘$4 > 0 {print $1,”from”,$6}’ some.data
- awk ‘{print}’ some.data
- awk ‘{print}’
- awk ‘NF > 0’ some.data
- awk '/n/; /e/' five.lines
- awk ‘/text/ {print}’
- awk ‘BEGIN {print “Hello World”}’
- awk '{ $1 = "THE LINE"; print}' five.lines
- ypcat passwd | awk -F: ‘$1 ~ /mark/ { print $1,"is a bozo"}‘
- awk ‘BEGIN {print $3-$4 }’ some.data
- awk '{print "Balance for",$1,"from",$6,"is:",$3-$4}' some.data

A Sample Program

```
ypcat passwd |
awk 'BEGIN{FS=":"}     #could use –F”:” on comand line
{print "Login id:", $1;
print "userid:", $3;
print "group id:", $4;
print "Full Name:", $5;
print "default shell:", $7;
print " " ;}'
```

String-Matching Patterns

- */regex/*
  - matches when the current record *contains* a *substring* matched by *regex*
  - /ksh/ { ... } # process lines that contain the letters ‘ksh’
- *expression* ~ */regex/*
  - matches if the string value of *expression* (can be a field like $3) *contains* a *substring* matched by *regex*
  - $7 ~ /ksh/ { ... } # process records whose 7th field contains the letters ‘ksh’
- *expression* !~ */regex/*
  - matches if the string value of *expression* (can be a field like $3) *does NOT contain* a *substring* matched by *regex*
  - $3 !~ /[4-6]/ { ... } # process records whose 3rd field does not contain a 4, 5, or a 6

awk Functions

- math functions: cos, int, log, sin, sqrt
- length(s) returns length of string
- index(s,t) returns pos of substr s in string t
- substr(s,p,m) returns substring of string s beginning at p, going length of m
- split(string, arrayname[, fieldsep])
  split splits *string* into tokens separated by the optional *fieldsep* and stores the tokens in the array *arrayname*
- gawk C-like extensions:
  - toupper()
  - tolower()
  - sprintf("fmt",expr)
- Example (what is my regex matching, revisited):
  - echo '111111' | awk '{sub (/1/, "X"); print }'

awk Arrays

- awk provides functionality for one-dimensional arrays (and by extension, multidimensional arrays)
- Arrays are associative in awk, meaning that a *value* is *associated* with an *index* (as opposed to a memory-based non-associated array scheme in C for example)
- By default, array indices begin at 0 as in C

awk Arrays continued

- This means that indexes (which are always converted to strings) may either be integral or textual (i.e., a string)
  - array[1] may return “un”
  - array[three] may return “trois”

```
awk ‘BEGIN{
for (i in ARGV)
print “Item”,i,“is:”,ARGV[i]
}’ one two three
```

Array Syntax

- To reference an array element:
  - – array[*index*]
- To discover if an index exists in an array:
  - – if (three in array)
    - print “three in French is”,array[three]
- To walk through an array:
  - – for(x in array) print array[x]
- To delete an individual element at an index:
  - – delete array[index]

Creating an Array using split()

split1.sh:

```
echo 'un deux trois quatre' |awk '{split($0,array)}
   END{
for (x in array) print "index:",x":",array[x];}‘
```

split2.sh:

```
echo 'un deux trois quatre' |
awk '{split($0,array)}
END{if ( 3 in array )
print "three in French is",array[3]}'
```

Real World Example

- from Aho, Kernighan, Weinberger, *The AWK Programming Lanugage*, chap. 4:
- cat countries
- cat prep.3
- cat form.3
- awk -f prep.3 countries countries | awk -f form.3

Review of C Programming Tools

Compilation

Linkage

The Four Stages of Compilation

- preprocessing
- compilation
- assembly
- linking

gcc driver program (toplev.c)

- cpp: C PreProcessor
- cc1: RTL (Register Transfer Language) processor
- as: assembler
- ld: loader (linker)

The GNU CC Compilation Process

- GCC is portable:
  - multiplatform (intel, MIPS, RISC, Sparc, Motorola, etc.)
  - multiOS (BSD,AIX, Linux, HPUX, mach, IRIX, minix, msdos, Solaris, Windoze, etc.)
  - Multilingual (C, Objective C, C++, Fortran, etc.)
- Single first parsing pass that generates a parsing tree

The GNU CC Compilation Process

- Register Transfer Language generation
  - close to 30 additional passes operate on RTL Expressions (RTXs), constructed from partial syntax trees
  - gcc –c –dr *filename.c*
  - RTL is Lisp-like
    - cond(if_then_else *cond then else*)
    - (eq: *m x y*)
    - (set *lval x*)
    - (call *function numargs*)
    - (parallel [*x0 x1 x2 xn*])
- Final output is assembly language, obtained by mapping RTX to a machine dependency dictionary
  - ~/mark/pub/51081/compiler/i386.md

Assembler Tasks

- converts assembly source code into machine instructions, producing an “object” file (called “.o”)

Loader (Linker) tasks

- The Loader (linker) creates an executable process image within a file, and makes sure that any functions or subprocesses needed are available or known. Library functions that are used by the code are linked in, either statically or dynamically.

Preprocessor Options

- -E preprocess only: send preprocessed output to standard out--no compile
  - output file: file.c -> file.i file.cpp -> file.ii
- -M produce dependencies for make to stdout (voluble)
- -C keep comments in output (used with -E above):
  - -E -C
- -H printer Header dependency tree
- -dM Tell preprocessor to output only a list of macro defs in effect at end of preprocessing. (used with -E above)
  - gcc -E -dM funcs.c |grep MAX

Compiler Options

- -c compile only
- -S send assembler output source to *.s
  - output file: file.c -> file.s
- -w Suppress All Warnings
  - gcc warnings.c
  - gcc -w warnings.c
- -W Produce warnings about side-effects (falling out of a function)
  - gcc -W warnings.c

Compiler Options (cont)

- -I Specify additional include file paths
- -Wall Produce many warnings about questionable practices; implicit declarations, newlines in comments, questionable lack of parentheses, uninitialized variable usage, unused variables, etc.
  - gcc -Wall warnings.c
- -pedantic Warn on violations from ANSI compatibility (only reports violations required by ANSI spec).
  - gcc -pedantic warnings.c

Compiler Options (cont)

- -O optimize (1,2,3,0)
  - -O,-O1 base optimizations, no auto inlines, no loops
  - -O2 performs additional optimizations except inline-functions optimization and loop optimization
  - -O3 also turns on inline-functions and loop optimization
  - -O1 default
- -g include debug info (can tell it what debugger):
  - -gcoff COFF format for sdb (System V < Release 4)
  - -gstabs for dbx on BSD
  - -gxcoff for dbx on IBM RS/6000 systems
  - -gdwarf for sdb on System V Release 4

Compiler Options (cont)

- -save-temps save temp files (foo.i, foo.s, foo.o)
- -print-search-dirs print the install, program, and libraries paths
- -gprof create profiling output for gprof
- -v verbose output (useful at times)
- -nostartfiles skip linking of standard start files, like /usr/lib/crt[0,1].o, /usr/lib/crti.o, etc.
- -static link only to static (.a=archive) libraries
- -shared if possible, prefer shared libraries over static

Assembler Options (use gcc -Wa,-*options* to pass options to assembler)

- -ahl generate high level assembly language source
  - gcc -Wa,-ahl warnings.c
- -as generate a listing of the symbol table
  - gcc -Wa,-as warnings.c

Linker Options (use gcc -Wl,-*options* to pass options to the loader)

- gcc passes any unknown options to the linker
- -l lib (default naming convention lib*lib*.a)
- -L lib path (in addition to default /usr/lib and /lib)
- -s strip final executable code of symbol and relocation tables
  - gcc -w –g warnings.c ; ls -l a.out ; gcc -w -Wl,-s warnings.c ; ls -l a.out
- -M create load Map to stdout

Review of C Programming Tools

Building Static and Dynamic Libraries

Static Libraries and ar (cd /pub/51081/static.library)

- Create a static library: the ar command:
  - ar [rcdusx] libname objectfiles ...
- Options
  - rcs: add new files to the library and create an index (ranlib) (c == create the library if it doesn’t exist)
  - rus: update the object files in the library
  - ds: delete one or more object files from a library
  - x: extract (copy) an object file from a library (remains in library)
  - v: verbose output

Steps in Creating a Static Library (cd ~mark/pub/51081/static.library)

- First, compile (-c) the library source code:
  ```
  gcc -Wall -g -c libhello.c
  ```
- Next, create the static library (libhello.a)
  ```
  ar rcs libhello.a libhello.o
  ```
- Next, compile the file that will *use* the library
  ```
  gcc -Wall -g -c hello.c
  ```
- Finally, link the user of the library to the static library
  ```
  gcc  hello.o -lc -L. -lhello -o hello
  ```
- Execute: `./hello`

Shared Libraries (cd /pub/51081/shared.library)

- Benefits of using shared libraries over static libraries:
  - saves disk space—library code is in library, not each executable
  - fixing a bug in the library doesn't require recompile of dependent executables.
  - saves RAM—only one copy of the library sits in memory, and all dependent executables running share that same code.

Shared Library Naming Structure

- soname: libc.so.5
  - minor *v*ersion and *r*elease number:
    - libc.so.5.*v*.*r* eg: libc.so.5.3.1
  - a soft link libc.so.5 exists and points to the *real* library libc.so.5.3.1
    - that way, a program can be linked to look for libc.so.5, and upgrading from release to libc.so.5.3.2 just involves resetting the symbolic link libc.so.5 from libc.so.5.3.1 to libc.so.5.3.2.
    - ldconfig does this automatically for system libraries (man ldconfig, /etc/ld.so.conf)

Building a shared library:
Stage 1: Compile the library source

- Compile library sources with -fPIC (Position Independent Code):

  -
    ```
    gcc -fPIC -Wall -g -c libhello.c
    ```
  - This creates a new shared object file called libhello.o, the object file representation of the new library you just compiled
- Create the *release* shared library by linking the library code against the C library for best results on all systems:

  -
    ```
    gcc -g -shared –Wl,-soname,libhello.so.1 -o libhello.so.1.0.1 libhello.o –lc
    ```
  - This creates a new release shared library called libhello.so.1.0.1

Building a shared library:
Stage 2: Create Links

- Create a soft link from the *minor* version to the release library:
  - ln -sf libhello.so.1.0.1 libhello.so.1.0
- Create a soft link from the *major* version to the *minor* version of the library:
  - ln -sf libhello.so.1.0 libhello.so.1
- Create a soft link for the *linker* to use when linking applications against the new release library:
  - ln -sf libhello.so.1.0.1 libhello.so

Building a shared library:
Stage 3: Link Client Code and Run

- Compile (-c) the client code that will *use* the release library:

  ```
  gcc -Wall -g -c hello.c
  ```

- Create the dependent executable by using -L to tell the linker where to look for the library (i.e., in the current directory) and to link against the shared library (-lhello == libhello.so):

  ```
  gcc -Wall -g -o hello hello.c -L. -lhello
  ```

- Run the app:

  ```
  LD_LIBRARY_PATH=. ./hello
  ```

How do Shared Libraries Work?

- When a program runs that depends on a shared library (discover with ldd progname), the dynamic linker will attempt to find the shared library referenced by the soname
- Once all libraries are found, the dependent code is dynamically linked to your program, which is then executed
- Reference: The Linux Program-Library HOWTO