Jordan Bayles
(Collaboration with Meghan Gorman)
Monday, 21 January, 2013

Assignment 1: Written Portion

Problem 1

There are many ways to transfer files between a local client and remote server, and the one that you select depends on the capabilities of the server. Some common protocols include SFTP (SSH file transfer protocol), rsync, AFP (Apple Filing Protocol), which each have their respective strengths, such as ease of use and security.

Problem 2

Revision controls systems are primarily a method for tracking changes to a software project over time. Some common implementations include Git and Mercurial SCM. Some of the most common uses include controlling who has access to canon code (”Blessed repository” in Git lingo), ”branching” off the main code in order to develop a new feature, and determining when bugs were added to the code base (and by who, such as using `git blame`).

Problem 3

Redirection is defined as switching a standard stream of data to either a non-default destination or source. Typically in UNIX systems, the redirection symbols $\langle$ and $\rangle$ are used to redirect process streams to or from a file. Piping is actually a special case of redirection, and is the act of chaining process input and output by using the `|` symbol.

Problem 4

Make is a software utility that uses a special filetype known as a “Makefile” to turn source code into either program executables or libraries. Although for some languages the compiler can perform this task and many IDEs (such as Visual Studio) as well, make is still widespread in use, especially in UNIX derived systems. Some of the primary advantages of using make include that it is lightweight and included on many Linux and BSD distributions by default (or is easy to add), thus programs that depend on make are easy to compile and use. Make is also excellent for development, because it can partially rebuild a program when only some of its source files have been edited and make can do more than create packages: it can install/remove them, clean object files, and generate tag tables for them, among other things.

Problem 5

The makefile is general broken into two portions: macros and rules.

Code 1: Sample macros

```
PACKAGE          = package_name
CC               = icc
```

Macros are similar to the C `#define` instruction, and are used to contain constants such as the compiler name, program version and name, as well as flags for programs called in the program.

Code 2: Sample rule

```
all:
        echo "Do nothing by default"
        echo "Try 'make target'"
```

Rules, by comparison, are a set of instructions that are used to build a *target* such as “release” or “debug.” A rule contains all of the information required to complete a certain task, and can include any command supported by the operating system, allowing you to perform nearly any task. In practice, make is typically called with a rule supplied (the above rule would be run with the command `make all`, for example).

Problem 6

Code 3: Find rule

```
find . -type f -exec file '{}' \;
```

Sample output:

Code 4: Sample find output

```
jordan@fenrir /home/jordan/Dropbox/schoolwork/cs311/1  (master)
-> find . -type f -exec file '{}' \;
./latex_segment.aux: LaTeX table of contents,
./python_segment.py: Python script, ASCII text executable
./latex_segment.pdf: PDF document, version 1.5
./latex_segment.log: ASCII text
./c_segment.c: C source, ASCII text
./names.txt: ASCII text, with very long lines
./latex_segment.tex: LaTeX 2e document, ASCII text
./words.txt: ASCII text, with very long lines, with no line terminators
```

Problem 7: Sieve of Eratosthenes

Code 5: Sieve implementation

```
/*
 *       Author:  Jordan Bayles (baylesj), baylesj@engr.orst.edu
 *      Created:  01/14/2013 08:08:04 AM
 *     Filename:  sieve.c
 *
 * Description:  Implementation of the Sieve of Eratosthenes
 */

#include <stdio.h>
#include <stdlib.h>

#define SPECIAL -1
#define UNMARKED 0
#define COMPOSITE 1
#define PRIME 2

#define DEFAULT_VALUE 1000L

struct primes {
  long count;
  long *values;
};

struct primes sieve_eratosthenes(long n)
{
  long candidates[n];
  long count;
  long m;
  long k;
  long multiple;
  struct primes results;

  /* Mark 1 as special */
  candidates[1] = SPECIAL;

  /* Set count to all (excepting 0,1), decrement as we go! */
  count = n - 2;

  /* Set k=1. While k < sqrt(n)... */
  k = 1;
  while (k * k < n) {
    m = k + 1;
    /* Find first # greater than k not IDed as composite */
    while (candidates[m] == COMPOSITE) {
      ++m;
    }

    /* Mark multiples as composite */
    multiple = m;
    while (multiple < n) {
      if (candidates[multiple] != COMPOSITE) {
        count -= 1;
      }
      candidates[multiple] = COMPOSITE;
      multiple += m;
    }

    /* Put m on list */
    candidates[m] = PRIME;

    /* Set k=m and repeat */
    k = m;
  }
  printf("\n");

  /* Build the results to return */
  results.values = malloc(count * sizeof(long));
  results.count = 0;
  for (long i = 0; i < n; ++i) {
    if (candidates[i] == PRIME || candidates[i] == UNMARKED) {
      results.values[results.count] = i;
      results.count += 1;
    }
  }

  return (results);
}

void print_primes(struct primes results)
{
  printf("Results contain %ld primes\n", results.count);
  for (long i = 0; i < results.count; ++i) {
    printf("%ld ", results.values[i]);
  }
  printf("\n");
}

int main(int argc, char *argv[])
{
  struct primes results;
  if (argc == 2) {
    printf("Using user value of %ld\n", atol(argv[1]));
    results = sieve_eratosthenes(atol(argv[1]));
  } else {
    printf("Using default value of %ld\n", DEFAULT_VALUE);
    results = sieve_eratosthenes(DEFAULT_VALUE);
  }

  print_primes(results);

  return (0);
}
```