IE305: PRINCIPLES OF PROGRAMMING LANGUAGES

31 October 2023

Duration: 10 Days Total Marks: 20

Answer all the questions. Marks are given alongside the questions.

1. **(14 Marks)** Choose ONE OF THE TWO PROBLEMS given below.

PROBLEM 1:

This exercise makes you implement a NUMBER THEORY LIBRARY. As you build the library, you will learn how the runtime structure influences the running program.

Write and organise your code as follows:

(a) The code should be in four files:

`nt.h`: a header file that contains information necessary for all the functions. It can have **only** `include`, `define`, `typedef` and *function declarations*. There may be a few `extern` variables but their number should be a minimum.

Any C code in the header file will automatically get you a 0 on this question!!

`nt_primes.c`: a C code file containing the following functions.

i. `int *primes_below(int num)`: finds all the prime numbers less than `num` and returns them in an array. Don't use any lists or such structures as you will have to write a lot of excess code.

ii. `int is_prime(num)`: returns `0` if the `num` is a prime; returns the smallest factor that divides `num` if it is not a prime.

`nt_factors.c`: a C code file containing the following functions.

i. `int is_divisible(int num, int x)`: returns `0` is `num` is divisible by `x`; returns any positive value otherwise.

ii. `int *prime_factors(int num)`: finds all the prime factors of `num` and returns them in an array. If `num` is a prime number, then it returns `NULL`.

iii. `int *proper_factors(int num)`: This function finds the factors of `num` from `1` upto but not including the number `num` itself. For example, the proper factors of 12 are 1, 2, 3, 4 and 6.

`nt_props.c`: a C code file containing the following functions.

i. `int *twin_primes(int num)`: returns all *twin primes* below `num`. Twin primes are numbers n and $n+2$ where both are primes. Examples are: $(3,5)$, $(11,13)$, $(41,43)$ etc.

ii. `int *is_abundant(int num)`: a number is said to be *abundant* is the sum of all its factors is more than the number itself; *deficient* if the sum is less; and *perfect* if the sum is the same. This function returns `-1` if `num` is deficient, `0`, if perfect and `1` if abundant.

(b) There are two `extern` variables:

`_numplt1000`: the number of primes below 1000.

`_primes1000[]`: an array containing the list of primes below 1000. These two are used by the other functions to find if a number is a prime, otherwise its prime factors, proper factors, etc.

(c) Write a `main()` function in a file called `nt_main.c` and use it to test all the functions you wrote.

(d) Compile and run your code as follows.

i. Compile each of the files `nt_primes.c`, `nt_factors.c` and `nt_props.c` separately using the command

```
gcc -c -fPIC <filename.c>
```

You can also use `-lm` in the above command, if needed.

ii. Build them all into a single library `libnumtheory.so` with the following command

```
gcc -shared -o libnumtheory.so nt_*.o
```

iii. Now, compile your `main()` function as

```
gcc -o ntmain nt_main.c -L. -lnumtheory -lm
```

Again, `-lm` is used only if needed.

After implementing the code above, draw the run-time structure clearly showing the locations of different variables at any stage in your running program.

List *when* and *where* each variable and its *association* are created for all those variables that you have in your above run-time structure diagram.

PROBLEM 2:

This problem makes you implement a SIMPLE DOCUMENT PROCESSING library. As you build the library, you will learn how the runtime structure influences the running program.

Write and organise your code as follows:

(a) The code should be in four files:

`sdp.h`: a header file that contains information necessary for all the functions. It can have **only** `include`, `define`, `typedef` and *function declarations*. There may be a few `extern` variables but their number should be a minimum.

Any C code in the header file will automatically get you a 0 on this question!!

`sdp_strings.c`: a C code file containing the following functions.

i. `char starts_with(char *sent)`: returns the start character of the given `sent`. If there is any problem with `sent`, it returns `'\0'`.

ii. `char ends_with(char *sent)`: returns the lastt character of the given `sent` discounting any punctuation. If there is any problem with `sent`, it returns `'\0'`.

`sdp_words.c`: a C code file containing the following functions.

i. `void words_and_sentences(FILE *ip, char *words[], char *sent[])`: This function returns an array of keywords and the array of sentences found in the file pointed to by `*ip` in two separate arrays via the parameters. If a keyword occurs more than once in the file, it will have multiple entries in the array `words`. We will only deal with sentences ending with a '.' (full-stop)

ii. `char **words_with_suffix(FILE *ip, char *suff)`: This function returns a list of all the words ending with the given suffix string `suff` from the file pointed to by `*ip`. For example, `words_with_suffix(inp, "est")` should give the answer as `'richest'`, `'largest'`, `'highest'`, etc.

iii. `char **words_with_prefix(FILE *ip, char *pref)`: This function returns a list of all the words beginning with the given prefix string `pref` from the file pointed to by `*ip`. For example, `words_with_prefix(inp, "est")` should give the answer as `'establish'`, `'estuary'`, `'estranged'`, etc.

`sdp_sentences.c`: a C code file containing the following functions.

i. `int *keyword_freq(FILE *ip)`: Returns an array containing how many times each *keyword* occurs in the words found in the file pointed to by `*ip`. The frequencies are given in the same order as that of the keywords in the external array (see below).

ii. `int number_sentences(FILE *ip)`: returns the number of sentences with one or more keywords in them.

(b) There are two `extern` variables:

`int _num_kwords:` the number of keywords for use.

`char *_keywords[]:` an array containing the list of *keywords*. These two are used by the other functions you will write in this problem.

(c) Write a `main()` function in a file called `sdp_main.c` and use it to test all the functions you wrote.

(d) Compile and run your code as follows.

i. Compile each of the files `sdp_strings.c, sdp_words.c` and `sdp_sentences.c` separately using the command

`gcc -c -fPIC` <filename.c>

You can also use `-lm` in the above command, if needed.

ii. Build them all into a single library `libsimpledocs.so` with the following command

```
gcc -shared -o libsimpledocs.so sdp_*.o
```

iii. Now, compile your `main()` function as

```
gcc -o sdpmain sdp_main.c -L. -lsimpledocs -lm
```

Again, `-lm` is used only if needed.

After implementing the code above, draw the run-time structure clearly showing the locations of different variables at any stage in your running program.

List *when* and *where* each variable and its *association* are created for all those variables that you have in your above run-time structure diagram.

2. **(6 Marks)** A crude version of *coroutines* is shown in the figures below. `yield` statement *pauses* the coroutine execution and returns control to the main program. `send` statement *resumes* the paused coroutines and the value sent as the argument is the value returned by `yield` statement. When you see the statement, `printname.send(6)`, the value 6 is returned by the `yield` statement in `printname` coroutine and sets the value of `n = 6`.

```
 1  printname = print_name('Chak')       # Initialise coroutine 1
 2  printnum = print_num(12)             # Initialise coroutine 2
 3  printname.__next__()                 # Start execution of 1
 4  printnum.__next__()                  # Start execution of 2
 5
 6  printname.send(6)                    # Resume exec of 1 from L4
 7  printname.send(4)                    # Resume exec of 1 from L7
 8  printnum.send(2847932)               # Resume exec of 2 from L4
 9  printname.send(4)                    # Resume exec of 1 from L7
10  printnum.send(0)                     # Resume exec of 2 from L10
```

```
 1  def print_name(prefix):
 2      i = 0
 3
 4      n = yield
 5      print('Coroutine 1: To print the word ', prefix, n, ' times')
 6      while i < n :
 7          burst = yield
 8          for j in range(i, min(i + burst, n)) :
 9              print('Coroutine 1: ', j+1, ') ', prefix)
10          i += burst
11
12      yield
```

```python
def print_num(maxnum) :
    i = 0

    number = yield
    print('Coroutine 2: Printing a given number ', number,
          maxnum, ' times')
    while i < int(8) :
        print(i+1, ': ', number)
        i += 1
    yield
    for i in range(8, maxnum) :
        print(i+1, ': ', number)

    yield
```

What is the output of the above?

Also, explain how these two coroutines may be implemented using the CIP and CEP pointers discussed in class. Follow the style and detail of explanation in the textbook (Pages 399 – 401 in the 4th Edition).