CSc 422 — Homework 2

Due Tuesday, February 19, 2002

This assignment is again worth 40 points. There are only four problems, but they are longer than the problems on the first homework, so get started sooner rather than later!

The first two problems are worth 5 points each; the third problem is worth 10 points; the program is worth 20 points. Append a commented listing of your program to your answers to all the questions. Also submit your program electronically as described at the end of the assignment. Be sure to follow the programming style described in the handout for Homework 1.

You may discuss the meanings of questions with classmates, but the answers and program you turn in must be yours alone. Again explain your answers clearly and succinctly.

1. MPD book, Exercise 3.2. Your solution should use the cache efficiently and have the level of detail of the program in Figure 3.4.

2. MPD book, Exercise 4.3. By a simulation I mean an implementation of the semaphore operations as they are defined at the top of page 155. Your implementation of the `P(s)` operation should use busy waiting.

3. MPD book, Exercise 4.31. For part (a), your answer should be similar to the readers/writers solution in Figure 4.10; be sure to give a global invariant. For part (b), a good way to make your solution fair is to alternate directions for cars (as in real life): first let all waiting northbound cars cross the bridge, then all southbound cars, and so on.

4. Write a parallel program that uses the bag-of-tasks paradigm to solve the following problem. Use C and Pthreads or use MPD. Develop your program on Lectura and test it on Parallel. Run the timing tests described below, and turn in (on paper) a table of results.

There is an online dictionary in `/usr/dict/words` on both Lec and Par. It contains 25,143 words (and is used by the `spell` command). Recall that a *palindrome* is a word or phrase that reads the same in either direction, i.e., if you reverse all the letters you get the same word or phrase. Your task is to find all *palindromic* words in the dictionary. A word is palindromic if its reverse is also in the dictionary. For example, "noon" is palindromic, because it is a palindrome and hence it's reverse is trivially in the dictionary. A word like "draw" is palindromic because "ward" is also in the dictionary. Your program should output the total number of palindromic words in the dictionary and the number found by each worker. It should also write the palindromic words to a results file.

I suggest that you first write a sequential program and then modify it to use the bag-of-tasks paradigm. However, do some things in the sequential program that you will need in the parallel program, such as finding where each letter begins in the dictionary (see below for more on this).

Your sequential program should have the following phases:

- Read the file `/usr/dict/words` into an array of strings. You may assume that each word is at most 25 characters long. You should also have a second array of the same length that will be used to indicate which words are palindromic; initialize this array to zeros.
- Examine every word, one at a time. For each, first compute its reverse. Then do a linear search of the dictionary to see if the reverse of the word is in the dictionary. (If the word

itself is a palindrome, you will find it!) If so, mark the word and increment your counter of the total number of palindromic words.

- Write the palindromic words to a file named `results` and write the total number of palindromic words to standard out.

The first few words in the dictionary start with numbers (take a look!); you can either skip over them or process them, as you wish. (None are palindromic, so this choice will not affect your total count.) Some words start with capital letters (and hence the dictionary is not sorted in ASCII order). Leave the capitals alone and do case-sensitive comparisons. (One might ideally like to convert capitals to lower-case, but it will simplify your program to ignore this.)

After you have a working sequential program, modify it to use the bag-of-tasks paradigm. Your parallel program should use `W` worker processes, where `W` is a command-line argument. Use the workers just for the compute phase; do the input and output phases sequentially. Each worker should count the number of palindromic words that it finds. Sum these `W` values during the ouput phase. (This avoids a critical section during the compute phase!)

Use 26 tasks in your program, one for each letter of the alphabet. In particular, the first task is to examine all words that begin with "a" (and numbers), the second task is to examine all words that begin with "b", and so on. During the input phase you should build an efficient representation for the bag of tasks; I suggest using an array, where the value in `task[1]` is the index of the first "a" word, `task[2]` is the index of the first "b" word, and so on. You can also use this array during the search phase to limit the scope of your linear searches.

Your parallel program should also time the compute phase. If you use Pthreads, use the `times` function as in the program `clock.c`. If you use MPD, use the `age` function as in the program `find.mpd`. Read the clock just before you create the workers; read it again as soon as they have finished. The return value from `times()` is in hundredth's of seconds; the return value from `age()` is in milliseconds. Write the elapsed time for the compute phase to the standard output.

To summarize, your program should have the following output: total number of palindromic words, the number found by each worker, the elapsed time for the compute phase, and the palindromic words. Write the first three items to standard out; write the words to a file named `results` in the directory that contains your program.

Timing Tests. Execute your parallel program on Par using 2, 3, and 4 workers. Run each test 3 times. Include a table of results with your homework answers; it should contain all the values written to standard output (but not the words themselves) for all 9 test runs. If you use MPD, be sure to set the MPD_PARALLEL environment variable to 4 just before you run the timing tests.

Electronic Turnin. Use the `turnin` program on Lectura to submit your program. The assignment name is `hw2.palindrome`. The file name should be `palindrome.c` or `palindrome.mpd`.