Operating Systems 0512.4402

Homework 2: Dispatcher/Worker model with Linux pthreads

In this exercise we will practice the dispatcher/worker model in Linux using threads to offload work. The dispatcher will spawn a configurable number of worker threads (num_threads). The commands to be executed are specified in a command file, read by the dispatcher. Some commands in the command file will be executed serially on the dispatcher, while some commands (jobs) will be placed on a shared work queue, and later off-loaded to an available worker thread.

Make sure not to waste CPU cycles in your solution – you should use synchronization features so that when waiting for something, you'll put the thread to sleep instead of busy waiting.

Command syntax:

```
hw2 cmdfile.txt num_threads num_counters log_enabled
```

- cmdfile.txt: input text file containing the commands.
- num_threads: number of worker threads to create. Max 4096.
- num_counters: number of counter files to create. Max 100.
- log_enabled: 1 to write trace files, 0 witout trace files.

Dispatcher Initialization

The dispatcher, running in the main code (main thread), will perform at least the follwing tasks (it can do more tasks, as required by your implementation):

1. Analyze the command line arguments to hw2, make sure that the number of parameters given is correct.
2. Create num_counters "counter files". Each counter file is called countxx.txt, where xx is a number between 0 to 99 (use two digits, so for example counter 5 will be called count05.txt). The contents of the file should be initialized to a single text line containing the number 0. The counters should be in decimal, in %lld format (long long, 64 bit).
3. Create num_threads new threads. We only create worker threads during init, and they will be alive for the entire run of the program. It is not allowed to create additional worker threads or to exit the threads in the middle of processing the command file – if a thread has no work to do at some point, it should go to sleep until new work is available to pick up.

Command file syntax

Each command line in the file is either a dispatcher command, or a job for a worker thread.

A dispatcher command is executed serially – the dispatcher will complete the execution of that command before proceeding to process the next line in the file. The list of commands the dispatcher should handle sequentially by itself are:

1. **dispatcher_msleep x** Sleep for x milliseconds before continuing to process the next line in the input file.
2. **dispatcher_wait** Wait for all pending background commands to complete before continuing to process the next line in the input file.

A line which begins with a **worker** means that the dispatcher will queue the entire line as a job in a shared work queue (offload it), and later on an available worker thread will pick it up for execution. The line includes a list of **basic comands**, which should be executed serially by the worker thread. The basic commands are separated by **semicolon (;)**. Words in the line are separated by spaces. There can be any number of spaces, and the max line width is 1024 characters.

Although the basic commands within a specific job will be executed serially on a worker thread, when there are multiple worker theads, multiple jobs (multiple such lines) can be executed on parallel in different threads.

Basic commands (included in a job line)

- **msleep x** Puts the worker thread to sleep for x milliseconds.
- **increment x** Increment the counter value in counter file x.
- **decrement x** Decrement the counter value in counter file x.
- **repeat x** There can only one repeat basic command in a job. It means to repeat x times the entire sequence of basic commands starting from the one after the repeat to the end of the line. For example: **repeat 5; increment 5; msleep 1** is equivalent to **increment 5; msleep 1; increment 5; msleep 1; increment 5; msleep 1; increment 5; msleep 1; increment 5; msleep 1**

Note: The counters have to actally be implemented in the disk files, and when incremented or decremented, it should be done in the files while the threads are running. It is not allowed to create all the counters in memory and just write the final version in the files at the end.

Log files

Log files should be created in case log_enabled is set to 1:

1. Each worker thread should open a log file called threadxx.txt, where xx is replaced by the thread number (in the order you called pthread_create, starting from 0).

2. Each time a worker starts a new job it should print to the log file a line: TIME %lld: START job %s, where %lld is the total time passed in milliseconds since the running of hw2, and %s is a copy of the job line from the input command file.

3. Each time a worker finishes a job it should print to the log file a line: TIME %lld: END job %s, where %lld is the total time passed in milliseconds since the running of hw2, and %s is a copy of the job line from the input command file.

4. The dispatcher should open a log file called dispatcher.txt.

5. Each time it reads a line from the input command file, it should print to the log file a line: TIME %lld: read cmd line: %s, where %lld is the total time passed in milliseconds since the running of hw2, and %s is a copy of the dispatcher line from the input command file.

Statistics

After finishing to process the input command file, the dispatcher should wait for all pending background commands to complete, display statistics to a file stats.txt, perform cleanup (release memory, close files, etc), and exit the program. The file stats.txt should contain 5 lines, as follows:

```
total running time: %lld milliseconds
sum of jobs turnaround time: %lld milliseconds
min job turnaround time: %lld milliseconds
average job turnaround time: %f milliseconds
max job turnaround time: %lld milliseconds
```

All running times should be wall times. In the jobs statistics, only command lines sent to worker thread should be included (excluding commands for the distpatcher). A job (command line) turnaround time is the time it ends processing on a worker thread, minus the time the dispatcher read that line.

Submission guidelines

- The solution should be submitted in moodle in a gzipped tar file called hw2_id1_id2.tgz, where id1 and id2 are the “tehudat zehut” of the two students (or hw2_id.tgz if submitting alone).

- The tgz file should contain a subdirectory hw2_id1_id2, and in the subdirectory there should be a Makefile, as well as the .c and .h source files. Running “make” should build the code into an executable called “hw2”. Running “make clean” should clean up the executables and object files.

- Submit one input command file, test.txt, that you used to test your implementation.

- Make sure to have comments in your source code.

- Submit an external documentation pdf in a file called hw2_id1_id2.pdf, describing your solution.