Lecture Notes on Operating Systems

Lab: Master and Worker Pool

In this lab, you will implement a simple master and worker pool, a pattern that occurs in many real life applications. You are given a program that spawns one master thread. The master produces requests periodically. You must complete this program in the following manner: you must spawn a pool of worker threads/processes to process these requests. The master and workers must share a request buffer of limited size. The workers must wait for requests to arrive in this buffer and then process them. The master and workers must coordinate to ensure that every request is processed exactly once and only once by some worker. The master must pause request generation if the request buffer is full with unprocessed requests. As far as possible, the master must ensure that the request processing load is balanced among the multiple workers in the pool.

In this simple program, requests are just integers, and processing a request simply involves printing the integer to screen. In real life applications, request processing could be more complex. For example, a common architecture for web servers is the multi-threaded/multi-process architecture, where the server has one master thread and a pool of worker threads/processes. When a new connection arrives from a web client, the master hands the request over to one of the workers. The worker then reads the web request from the network socket and writes a response back to the client. Your simple master-worker program is similar in structure to such applications, albeit with much simpler request processing logic at the application layer.

You are given a file `master-worker.c` and you must add code to the same file to complete this lab. The main function spawns a master thread which generates 10K positive integers. You must write code to spawn 3 workers to print these integers to the screen. The master and all the workers must use one shared request buffer of size 1K integers to pass requests between themselves. A generated integer is added to the request buffer by the master, and must stay in the buffer until it is printed out by one of the workers. To test the correctness of your program, you must verify that your workers print out each of the 10K numbers once and only once to the screen (in any order). Your code must work correctly irrespective of the number of requests generated, the size of the request buffer, or the number of workers.

Part A: Worker threads

For this part, you will spawn 4 POSIX threads to act as workers, and use a shared array of 1K integers as the request buffer. You must use mutexes and conditional variables of the `pthreads` API to synchronize between the master and workers.
Part B: Worker processes

In this part, you will realize the worker functionality using processes. The master will fork worker processes to process the requests. The master and worker processes will use one 4KB page of shared memory as the request buffer. Note that, in addition to the request buffer, you will also need some other mechanism for signaling between the master and workers, e.g., for the master to signal to a worker that a certain request has been assigned to it, or for the worker to notify the master that processing has been completed. You may use some other IPC mechanism such as pipes, sockets, or message queues for this purpose.