PA6: High Concurrency without many Threads

Introduction

In this programming assignment, we try to further improve the performance of the client from PA4 (or PA5) by reducing the thread management overhead for a large number of worker threads. We do this by replacing the collection of worker threads by a single thread to manage all the request channels. This means that the solitary worker thread must be extremely efficient managing IPC channels with the server and avoid any idle time. One major idle time for the worker threads in PA4 is the time between sending a request and receiving its response, because, recall that the server inserts a random delay between the two for data requests. The single worker thread in PA6 avoids this idle time by decoupling two operations and use the time saved this way on the rest of the channels. The arriving responses will be handled on a First-Come-First-Serve basis with the help of `epoll()` that can efficiently monitor activity in several IPC end-points.

Background

Decoupling Write and Read Operations

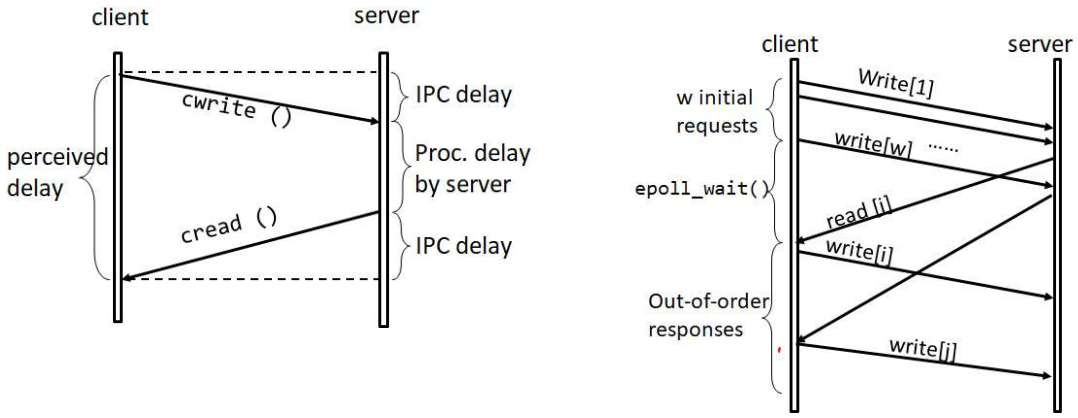
Each worker thread in your PA4 has the following structure for talking with the server:

```
while(true){
    ...
   chan.cwrite(msg)
   chan.cread (response)
   process (response)
    ...
}
```

In other words, the thread first sends a request to the server by calling `cwrite()` and then immediately blocks for the response to come back from the server by calling `cread()`. The problem is that each data response takes a random amount of time to come back which keeps the respective thread blocked for a significant duration. However, this problem of blocking is solved by introducing many worker threads so that while one thread is blocked many other threads continue to keep the CPU busy. The experience by a single worker thread is shown Figure 1a.

Using `epoll()` Suite for Monitoring IPC Activity

Now, the situation changes as a single worker thread must manage all the channels, because if it blocks on a response, valuable time is lost. To avoid this problem in PA6, we will decouple the read operations from the writes. To get started with, the worker thread sends w requests - one through each channel, without immediately waiting for them. Instead, the worker thread puts all channels in a watch list facilitated by `epoll_create1()` and `epoll_ctl()`. The worker thread then waits on this list as a group by calling `epoll_wait()`, which wakes up whenever one or more of the channels come back with some data. After that, say channel



(a) Blocking delay when `cwrite()` and `cread()` are used back-to-back.

(b) Minimized wait time at the cost of out-of-order arrival with I/O multiplexing.

Figure 1: Delay observed in case of blocking and multiplexing I/O

i has data, the worker thread first reads that data, processes it by updating the respective histogram and then pumps more data (i.e., found in the request buffer) through the channel i. The worker thread does this for all channels that have data available. This new situation is described in Figure 1b. `epoll(7)` in the linux manual page has more information about how to use this properly.

Note that `epoll()` does not work directly work with channels, rather only with file descriptors. For that reason, you can modify your `FIFORequestChannel` class by either making the read file descriptor a public member or by providing a getter function. Note that you do not need to monitor the write file descriptors. If activity has been detected on the read file descriptor, only then you would call `cread()`. In addition, the next request from the request buffer should also be sent using `cwrite()` on the same channel.

State Management

Since the send and receive functions are decoupled, some extra steps are needed for processing incoming responses, because they do not contain any information about the original request. Therefore, we must main some state information that memorizes which channel is carrying what message. This per channel state is updated everytime something new is sent through the channel and eventually used when a response is received for this channel. In your implemenation, it can be an array of size w where the i-th item is a char array itself, conveniently storing the last message sent through the i-th request channel. If you prefer C++, you can use a `vector<vector<char>>` to the same effect.

Termination

In PA4, the QUIT messages were used to break the main loop in each worker thread. This scheme will no longer work for PA6 because although a QUIT can indicate to the worker

thread when to stop sending messages, it does not tell the worker when to stop receiving responses. Note that the number of received responses always trail behind the number of sent requests because usually there several messages that our outstanding, i.e., their responses have not arrived yet. For that reason, the right way to stop the worker thread is to maintain two separate counts - the number of sent requests and the number of received responses. The main function should wait for the patient/file thread(s) and then push a single QUIT message in the request buffer. The worker/poll thread pops the QUIT and stops sending more requests. However, it continues receiving responses until the number of received messages equals that of the sent requests. Once everything is done by this polling thread, it then pushes necessary QUITs on to the response buffer to terminate the histogram threads as was in PA4/5.

Others

Another point to keep in mind is the priming phase. The `epoll_wait()` can only work on file descriptors that are expecting some receiption. For that to work, the worker thread must "prime" the channels by sending some initial data through each channel, which should of course come from the request buffer. If priming is not done, `epoll_wait()` will give a deadlock.

In addition, `epoll_wait()` gives back a list of file descriptors that have data. You need to map these file descriptors to the respective request channels, because you are going to pump more data through that channel. Now, to do this quickly, which you must, you should use some map data structure or simply an integer array, where the index is the file descriptor that leads you to the index into the channel array.

The Assignment

You are to write a program (call it client.cpp) that first forks of a process, `exec()`s the provided data server, sends a series of requests to the data server and collect their responses. The client should consist of p patient threads and one worker thread. The client program is to be called in either of the following two command formats. First, to request some data points, we will use:

```
./client -p <number of patients> -n <requests/patients>
  -b <bounded buffer size> -w <number of request channels> -h <number of
     histogram threads>
```

Note that in $PA4$, w was both the number of worker threads and the number of request channels. However, this time, w represents only the number of channels to use.

Use the following command for transferring files:

```
  ./client -f <file name> -m <buffer capacity>
    -b <bounded buffer size> -w <number of request channels>
```

What to Hand In

- Submit the solution directory containing all files and also a makefile.
- Analyze the performance of your implementation in a report, called report.pdf. Measure the performance of the system with varying numbers request channels and sizes of the buffer. How does the performance compare to your implementation in /PA4? Does increasing the number of request channels still improve the performance? If so, by how much? Is there a point at which increasing the request channels does not further improve performance? Submit a report that compares the performance to that of your solution in PA4 as a function of varying numbers of request channels (i.e., worker thread in the case of PA4).

Rubric

1. Modified worker thread (30 pts)
    - Priming the channels (i.e., sending initial requests): 5 pts
    - Removing unnecessary locking (because everything is being done by a single thread): 5 pts
    - Maintaining the state for detecting what response corresponds to which request: 10 pts
    - Proper termination (showing histogram at the end): 10 pts
2. BoundedBuffer (10 pts)
3. Not having global variables (5 pts)
4. Handling the case when $w > p * n$ (5 pts)
5. Cleaning up fifo files and all dynamically allocated objects (10 pts)
6. Correct counts in the histogram (20 pts)
7. Correct file transfer output. (10 pts)
8. Report (10 pts)
    - Should show plots of runtime under varying w in range $[1, 500]$ after setting $n = 15\text{K}$, $p = 15$, and $b = 1024$.
    - Compare the shape of the runtime curve with that of PA4. Do you see any difference? Does the point of diminishing return change?