Dennis McWherter
22 October 2011
CS 241
dmcwhe2

In MP6, I had to implement the MapReduce algorithm using N processes (depending on how the work was divided up) and a single worker thread. In order to combine all the work, I used *pipe()* and *poll()* for I/O multiplexing. I used this method purely due to familiarity with it over *select()*; other than that, there was no particular reason for using *poll()*. Furthermore, the advantage of only using a single thread vs. thread per process was to simplify the asynchronous I/O as well as allowing the main application to more easily use data as soon as it was ready.

In this MP, I ran different *map()* processes on the same machine, however, I could extend this to multiple machines using the network. In POSIX, a network socket is still a type of file descriptor which could have made my solution fairly portable to a multi-server solution since I used *pipe()*/*poll()*. In alternate solutions of MapReduce (particularly those which us shared memory instead of file descriptors), additional worker threads may not be needed. For instance, if *mmap()* was used instead of *pipe()* or *fifo()*, to retrieve data from the dictionary, you could simply check it from the shared memory whenever you needed to retrieve the value.

My solution required a worker thread because I wanted to asynchronously retrieve data as soon as it was ready while still processing other work on the main thread; to do this with a file descriptor requires waiting for the data (i.e. *poll()*/*read()*). With that in mind, the dictionary can be instantly updated by other processes in shared memory (i.e. using *mmap()*) and can be instantly viewed by any of the sharing processes (provided proper synchronization). This achieves the same asynchronous result since the other processes run independently and will update the dictionary directly rather than requiring a thread to handle the data being sent through the file descriptor.