CS 4348/5348 Operating Systems -- Project 3

In Project 1, you have built the skills for using fork, pipe, and thread creation. This project will involve some skills you have learnt as well as additional skills in thread synchronization, signal handling, and socket programming. You will use the Admin Process and Cal Process you constructed in Project 1 for this project, but with significant modifications. The computation in the Cal process will be replaced by a pipelining sorter with multiple layers of parallel threads that need to be carefully synchronized. Also, the Admin Process will be enhanced and will not only interact with the sorter, but also with clients through sockets.

This project is to be carried out by individual students, not a team project. Some code files to be used for the project and links to information that may be helpful for completing the project are available at https://personal.utdallas.edu/~ilyen/course/os/home.html.

1 A Sorter for the Cal Process

Replace the code in the Cal process in Project 1 by a pipeline sorter. The Sorter uses multiple layers of parallel threads to perform merge sort. The to be sorted data are stored in a Sorter Array, and progressively get sorted layer by layer. There are $\log_2 M + 2$ layers (layer 1 is the first layer) of sorter threads in the Sorter, where M is the number of parallel threads in the second layer.

1.1 Function of Each Sorter Layer

First layer is a **single Input thread**. It reads from pipe first the **data size *N*** (number of integers to be sorted) and then the N integers to be sorted. The N integers should be store in the Sorter Array. To avoid incurring the additional copying cost, you should directly use the Sorter Array as the buffer for the pipe read function.

Second layer consists of ***M* parallel threads**, each is responsible for sorting a segment of the Sorter Array and the size of each segment should be N/M. You can use any sorting algorithm to do this.

The remaining $\log_2 M$ layers are Mergers. Layer l consists of $M/2^{l-2}$ parallel threads, each is responsible for merging the data in two segments of the previous layer. As can be seen, the last layer only has one thread, responsible for merging 2 segments with $N/2$ data elements in each segment.

To avoid dealing with unnecessary details in the code, we consider that N and M are both powers of 2 and $N > M$. Also, we would like to have a fixed sorter structure without incurring the cost of dynamic thread creations, so, M is fixed and will be given as an input at the initialization time. With a fixed M, the number of layers in the sorter is fixed. So, you can create all the sorter/merger threads on initialization and they only terminate on a termination message. Note that N is dynamic, defined in each client sorting request. But for convenience, we will use a maximal data size of 1024 (i.e., $N \leq 1024$).

After the Sorter Array going through $M + 1$ layers of sorting, it is fully sorted. The single thread in the last layer should print out this sorted array in an organized format. To facilitate easy observation of the correctness of your code, we would like to see the printout of the Sorter Array at each layer when the **"*D*" flag** (indicating debug or details) is on (1 is on and detailed printing is needed, 0 means off). You can designate one thread of a certain layer to print the Sorter Array when required. The formats for the inputs and outputs will be given later after the specifications for all system components are given.

1.2 Pipeline Sorter and Thread Synchronization

The layered sorter structure discussed in the previous section is designed to handle a flow of sorting requests from the clients in a pipelining manner. When the Layer 2 threads finish sorting the individual

segments of the array and pass the Sorter Array to Layer 3 (absolutely no copying, even in the Input layer), they are free to start the next sorting request. Since the Sorter Array will be used by the successive merging layers, we need to use several Sorter Arrays to support pipelining. We have ***A* Sorter Arrays** in the sorter and A will be given as an input data. A Sorter Array can be reused by the first Input layer for a new sorting request only after the last layer finishes printing its contents.

To ensure the correct sorting results, Layer l merging should not start before the corresponding segments are sorted from Layer $l-1$. You need to design a synchronization mechanism to ensure the ordering of execution. Also, in order to control the reuse of a Sorter Array, the Input layer thread needs to synchronize with the last layer merger to ensure that it is not preliminarily overwritten. Moreover, if for a period of time, there is no new client requests and the existing sorting tasks are done by a certain layer, then it should wait till a new sorting task comes. Whenever there is a sorting task to be done by a layer, it should start sorting without being blocked. You need to design the synchronization mechanism and implement them for the sorter in the Cal Process to ensure its proper behaviors. There should be no thread creation or joining besides at the initialization and termination time. There should be no sleep() call or alike for fake synchronization.

You need to specify your thread synchronization mechanism in the **design.pdf** file. You can give the pseudo code and some brief descriptions to explain your design. Keep it precise and concise.

2 The Admin Process

Instead of getting input commands from a file, now the Admin Process uses a server socket to receive the input sorting requests from clients. Similar to Project 1, Admin forwards the sorting requests it has received to Cal via a pipe, first sending the data size N, then the N integers to be sorted.

Since there are only A Sorter Arrays in the Cal Process, the Input layer thread in Cal should not accept any sorting request from Admin when there is no free Sorter Arrays. This requires synchronization between the Cal Process and the Admin Process (essentially the Producer-Consumer problem but between processes). You need to design the synchronization mechanism such that the system can guarantee that (1) no request will be lost even if the pipe does not have any buffering capacity, (2) Admin should not be blocked from sending the sorting requests to Cal when Cal does have available Sorter Arrays, (3) Admin will not be blocked from receiving the sorting requests from the clients when there are client requests to be received, and (4) the overhead for synchronization mechanism should be reasonable.

Try not to use nonblocked read (can cause busy waiting) or multiple threads in Admin or the select() function, and instead, consider building the solution with the signal and pipe mechanism. However, if you cannot get a good solution with these constraints, you can choose to relax the four criteria or use some mechanisms mentioned above. You should describe your synchronization mechanism between Admin and Cal in the **design.pdf** file. You can give the pseudo code and some brief descriptions to explain your design. Keep it precise and concise.

To avoid the potential loss of any sorting request from the clients, Admin should maintain a queue for storing the received sorting requests that cannot be forwarded to the Cal Process right away. Let ***Q* be the queue size** and Q will be given as an input to Admin. Since we have already explored many synchronization mechanisms, we assume that the queue will never overflow so that we do not need to consider additional synchronization needs.

3 Clients

You need to implement a client program (name it **client.c** or **client.cpp**) that reads sorting requests from a user and sends the request to Admin. It should prompt to the user for entering an input file name. The file can have any name except for "End". Each file contains one sorting request, including the number of data elements (N, the first integer) and N integers to be sorted. Each client should be given a **CID (client ID)** as a command line input.

At the initialization time, the client set up the socket connection to Admin. After reading in the input file, the client sends the sorting request to the Admin Process and prompts to the user again for additional sorting requests. The client terminates when the user gives string "End" as the input file name. Upon termination, the client should close the connection to the server socket.

Each request from the client to the Admin Process should include the **CID, the request file name, and all the $N + 1$ integers**. Note that the client and Admin&Cal may run on different computers and (they may not have the shared file system like NFS (will be our testing setup). So, the client **should not** just send the file name but **should** include all the $N + 1$ integers in the socket message to Admin for each sorting request. The same applies to sorting request messages from Admin to Cal.

Note that even if all the current clients have terminated, the system cannot terminate because it does not know whether new clients will come and send new requests (like any client-server systems). In practice, a server system will continue to run till it shuts down or fails. In our case, we will use **cntl^C to terminate the Admin and Cal processes**. However, we would like a graceful termination. Thus, Admin should capture the cntl^C signal and should control the termination to be after finishing all the requests it has received from the clients. Admin does need to send a termination message via pipe to Cal to inform Cal to terminate. The Cal Process should finish sorting all the requests received from Admin before terminating. In fact, as a good practice and to avoid any problem for the Cal to read the remaining requests that are still in the pipe, Admin should wait() so that it terminates after Cal has terminated.

You should test your code by starting multiple clients with an Admin. Also, you should control your client request issuing time so that you can test out different scenarios. For example, Admin and Cal should work properly (no busy waiting) even if all the existing requests are fully processed and no new requests are received. Also, your sorting/merging thread synchronization mechanism should be thoroughly tested by issuing the client requests at different rates (e.g., overflow the A Sorted Arrays or starve them).

4 Input and Output Specifications

The configuration parameters M, A, Q, and D will be given as command line inputs to the Admin Process and some of the parameters should be made available to the Cal Process (**not via pipe**).

We would like to see proper formatting when printing the final sorted or intermediate results. For each request, the output should start with an empty line followed by a separator line and then followed by the printout of the Sorter Array contents. You should have 10 integers in each output line (besides the last line that may have fewer integers) with each number being separated by a space. The separator line should include CID, the request file name, and N, with some leading dividing string. For example:

```
========== CID=5, file=sortreq.1, N=32
1 13 20 21 25 52 55 108 111 133
158 230 339 567 585 596 677 812 850 876
1024 2048 4096 8192 10581 11323 11666 11777 11888 11999
20385 25999

========== CID=3, file=req.txt, N=16
…
```

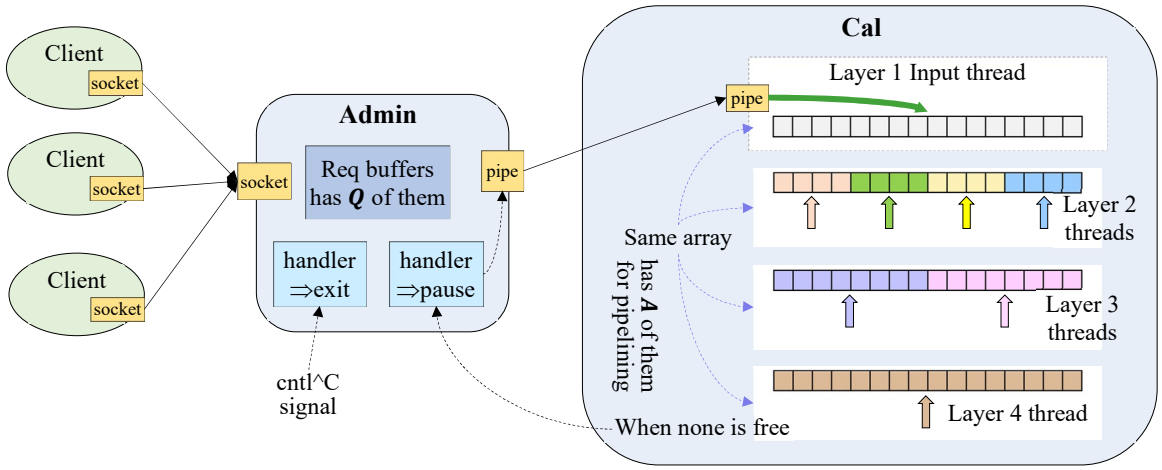
If the flag "D" is on, then we need to separate the outputs from different layers. You should add a divider line to divide the outputs of different layers, but without an additional empty line. The separator could contain '=' characters followed by the layer number. For example, for $M = 2$, we can have

```
========== CID=100, file=req.1, N=16
========== L=1
108 22 230 3 1 158 567 55 15 111
596 10 133 585 339 11
```

========== L=2
3 22 108 230 1 55 158 567 10 15
111 596 11 133 339 585
========== L=3
1 3 22 55 108 158 230 567 10 11
15 111 133 339 585 596
========== L=4
1 3 10 11 15 22 55 108 111 133
158 230 339 567 585 596

========== CID=5, file=sortreq.1, N=32
…

5 Remarks



To help you understand the design of the project better, above is a diagram to illustrate the pipeline sorter with multiple sorting/merging threads and the overall system.

It is always better to work on a project stage by stage. Here is a list of recommended subgoals for the implementation for this project.

Step 1. Implement the sorting/merging functions following the structure given in the figure (also specified in Section 1.1), but in a sequential manner without threads. You just need to implement one sorting function and one merging function with the identifier of the Sorter Array, the pointer to the beginning of the data segment in the array, and the data segment size as input parameters. Then, following the sorter structure, write a main driver function to invoke the sorting and merging functions multiple times with the proper input parameters. Invocation should be done sequentially following the layered structure. The Input layer at this point could simply read sorting requests from a file or alike into the Sorter Array. Make sure the sorter sorts correctly.

Step 2. Create threads and assign each to either the sorting or the merging function as what you have invoked in the main driver with the same input parameters. Synchronize the threads for proper sorting and for handling multiple requests in a pipelining manner. The input layer still reads input requests from a file or alike. Control the input layer so that you create situations where A requests come to the system at the same time and situations where no request comes to the system. Ensure the correctness of your system.

Step 3. Use your Admin-Cal communication code from Project 1 to send the input requests from Admin to Cal. Implement socket interface in Admin. Implement the Client code to read in input requests and send them to Admin via socket. At each communication interface, print out what is sent and what is received to ensure that the communication mechanism works properly. Do not invoke the sorter threads but just activate the Input layer for receiving input requests so that you can focus on debugging your communication system.

Step 4. Activate the sorter threads and implement the synchronization solution between Admin and Cal. Finalize your project and test it thoroughly.

6 Project Submission

You need to submit your program **before** midnight (11:59pm) of the due date (check the **web page** for the due date). Use UTD elearning system for submission: "elearning.utdallas.edu".

Your submission should include the following:

- All source code files making up your solutions to this assignment.
  - Updated “admin.c(pp)” and “cal-new.c(pp)”, and new “client.c(pp)”
  - Any additional source code files you have for the project
  - A makefile to generate admin.exe and client.exe (strictly follow the naming)
- A “readme” file
  - Indicate how to compile and run your programs in case it deviates from the specification
  - If you did not finish the project, you need to specify which part(s) you have completed and which part(s) are missing
- A “design.pdf” file
  - Discuss the synchronization mechanism for the sorting, merging, and Input threads
  - Discuss the synchronization mechanism between Admin and Cal
  - Any other design ideas you may have that is different from the specification
- You can zip or tar all your submission files together and then submit it on elearning