CS69001 Computing Laboratory – I

Assignment No: C2

Date: 16–October–2019

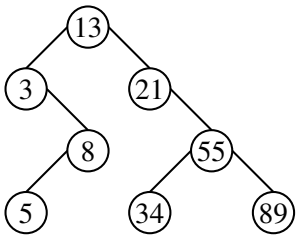
In this assignment, you implement a multi-process application. The processes run independently in different terminals, and are not related to one another. The processes collaboratively build a binary search tree (BST) in a shared-memory segment. There is one controller process acting like a daemon. Multiple user processes access and modify the tree. The processes synchronize using named pipes (FIFOs).

In the rest of this assignment, we use the following notations.

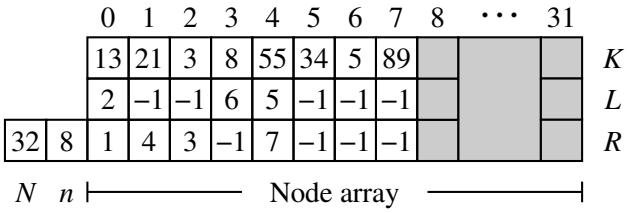
T = The binary search tree.
M = The shared-memory segment storing T (and a few other things).
N = The maximum number of nodes allowed in the BST (take $10 \leqslant N \leqslant 100$).
n = The actual number of nodes in the BST (we have $0 \leqslant n \leqslant N$).
P = The maximum number of user processes that can participate (take $2 \leqslant P \leqslant 10$).
C = The controller process.
U = A user process.
RQ = The FIFO to which a user process sends a BST request to the controller process C (the same for all user processes).
$GQ(U)$ = A FIFO through which the controller process C grants the request of a user process U (specific to the user process U).
$DQ(U)$ = A FIFO through which a user process U notifies the controller process C about the end of a BST task (specific to the user process U).

Part 1: Binary search tree in shared memory

A binary search tree T of maximum node capacity N is stored in a shared-memory segment M. The segment stores N and the current number n of nodes in T in addition to the keys and the child information of the nodes. A node is specified by a triple (K,L,R), where K is the key, L is the index of the left child in the node array, and R is the index of the right child in the node array. A non-existent child is represented by the index -1. All of N,n,K,L,R are integer values. The following figure describes the organization of the memory segment M. It should have enough space to accommodate N, n, and N triples of the form (K,L,R).



(a) A binary search tree



(b) Storage of the tree in shared memory

Initializing T to an empty tree is effected by setting $n = 0$. As new keys are added to T, the triple array is filled from the beginning (index 0) to the end (index $N-1$). The current size n indicates the index of the next node to be added. If $n = N$, we say that the memory M is full, and no further insertion in T is possible.

Write functions *BSTsearch*, *BSTinorder*, and *BSTinsert* for T stored in this format. These functions are needed by the user processes.

Part 2: The controller process *C*

Write a program *controller.c*(*pp*) to implement the work of the controller process C. This process initially creates the shared-memory segment M for storing the BST in the given format. C also creates the request queue RQ. Then, C enters a loop for reading requests from user processes. The loop breaks when all of the P allowed user processes send quit requests. Finally, C removes the shared-memory segment M, and exits.

A request from a user process U is one of the following. Here, PID refers to the system-generated PID of U.

`R PID`	Registration request of a new user process U.
`S PID Key`	Request for searching a key in the tree T.
`P PID`	Request for the inorder printing of the keys in T.
`I PID Key`	Request for the insertion of a key in T.
`Q PID`	Request for quitting.

The controller process C keeps on reading these requests from the running user processes. If C reads a registration request from a new user process U, C first checks whether the number of user processes is less than P. If so, it remembers the PID of U, and sets up the communication channel via the two queues $GQ(U)$ and $DQ(U)$. If the number of user processes is already P, this registration request is ignored.

For a quit request from a user process U, the controller process C marks the process as EXITED. When a total of P quit requests come to C, it itself breaks from the request-processing loop.

For a BST-related request (search, print, or insert) from U, C grants the request by sending a token to $GQ(U)$. Since search and print operations do not modify the tree T, this granting token is sent to U as soon as the request is read. An insert request, however, requires some additional care from C before the granting token is issued (see Part 4).

Part 3: The user processes *U*

Write a program *user.c*(*pp*) to implement the work of each user process U. First of all, U checks whether the request queue *RQ* is available (use `open` with the `O_NDELAY` flag). If the queue is not available, the controller process C is not ready, so U immediately quits. Otherwise U proceeds as follows.

U first creates the two FIFO queues $GQ(U)$ and $DQ(U)$. These two queues are for synchronization between C and this user process U only. Use the PID of U in the names of the FIFOs.

After the queues are set up, U sends a registration request to C. If the request is granted, U enters a loop which keeps on generating random BST-related requests (search, print, and insert) and writing those requests to *RQ*. When granted access to the BST T, U calls the functions of Part 1 to carry out the desired tasks. The loop continues until U tries to insert a new key to T already containing N nodes (that is, until the memory M gets full). When this happens, U sends a last print request to C. After the printing, U sends its quit request to C, and exits.

After sending each request to C, U waits on $GQ(U)$ to receive the granting token from C. Moreover, after the completion of each BST task, U sends a token to the done queue $DQ(U)$.

Part 4: Synchronization

Synchronization is achieved by the blocking read calls on appropriate FIFOs. Each user process U needs the granting token from C before it applies the requested operation on T. Notice that after all writer processes exit from a FIFO queue, the reader process sees an EOF in the read end of the pipe. On the other hand, when all reader processes for a queue exit, an attempt to write to the queue sends the signal SIGPIPE to the writer process. In this application, the user processes may be non-overlapping (in time). The exiting of a user process should not allow *RQ* to indicate EOF, because in future another user process may try to use that queue. To avoid this problem, the controller process should open the queue with the option `O_RDWR` to ensure that both the ends of the queue remain open as long as C is running. For the grant and done queues, this is not a requirement, but it is safe (and harmless) that C opens all FIFOs in the read + write mode.

The search and print operations on T do not modify the tree, so C issues granting tokens as soon as it reads these requests. However, an insert operation on T modifies the tree, and throughout each insert operation, only the process U performing this operation should be granted access to T. Strictly after U completes insertion, other requests may be processed. This synchronization is achieved as follows.

The controller process C keeps track of how many unprocessed granting tokens have been issued to each running user process. One or more of these processes may be not yet done with the read-only requests on T. Whenever C reads an insert request from *RQ*, it does not grant the request immediately. Each user process, upon the completion of a BST task, sends an end-of-work token to its done queue, and C knows how many such tokens must be received to ensure that all these granted operations are complete. It reads the exact

number of tokens from each done queue. When all these reads finish, it is ensured that each running process either is waiting on its grant queue or is yet to make the next request. C now sends the granting token to $GQ(U)$, where U is the process which has made the insert request.

Consequently, only this user process U springs into action. C must wait until U finishes its insertion task, so C waits on $DQ(U)$ for the end-of-work token from U. After this is received, C goes to process further requests pending in the request queue RQ.

Sample output

A verbose sample-output file is provided separately in the lab web-site.

Submit two C/C++ files *controller.c*(*pp*) and *user.c*(*pp*).
Do not use global/static variables or STL data types.