CS69001 Computing Laboratory – I

Assignment No: C1

Date: 30–September–2019

In this assignment, you prepare and work with a process tree T. Each process P in the tree T is given a user-specified key which is a positive integer. With respect to these keys, T should be a binary search tree (BST). The root process is denoted by R. This (and only this) process has access to a file which stores the sequence of the user-specified keys (with possible duplicates) delimited by the invalid key -1. The file specifies the sequence of inserting the keys in the BST. The terminating key -1 is needed because the file does not store how many keys are contained in the file. The sample output shows an input file with 25 keys (of which three are duplicates).

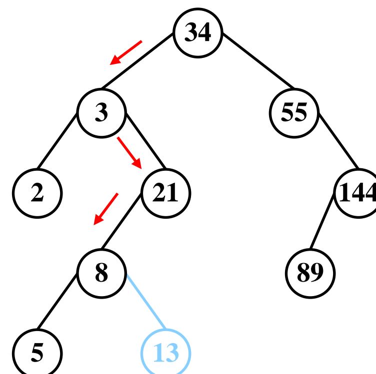
Communication is possible between a process P and its two child processes P_L and P_R. A pipe is opened by P for communication with P_L if it exists. Likewise, another pipe is opened for communication with P_R if it exists. If P does not have a left/right child, the corresponding pipe is not opened. Just before P forks a new child, the corresponding pipe is opened. P itself needs to communicate with its parent Q (unless P is the root). This communication is to be made using the appropriate pipe (left or right depending upon whether P is the left child or the right child of Q) created by Q just before it forked P. All pipes are bidirectional (that is, no end is closed by any process), because you need to communicate both down and up the tree.

Each process should maintain the following data in this assignment. You may have other process-specific data, but the following items must be maintained.

- The user-specified key (a positive integer).
- The PID of the left-child process (UNDEFINED if this child does not exist).
- The PID of the right-child process (UNDEFINED if this child does not exist).
- The PID of the parent process (UNDEFINED if the process is the root R).
- A pipe to talk with the left child (not opened until this child is forked).
- A pipe to talk with the right child (not opened until this child is forked).
- A pipe to talk with the parent (created by the parent, UNDEFINED for R).

Immediately after a process P is created (the root R is created when you run your code), its key and its child PIDs are set to UNDEFINED. The root R always keeps its parent UNDEFINED, but any other process sets its parent PID by calling **`getppid`**. This convention lets the root process identify itself uniquely as is needed on several occasions later. The left- and right-child pipes are not opened immediately after P is born. However, if P is not the root R, its parent pipe is set from the data passed by its creator.

The adjacent figure illustrates a process tree T. This is just a BST, but the nodes of T are processes, and each parent-child link is essentially a pipe. The figure also demonstrates the insertion of a new key to the process tree. The communication path is shown by the lines with arrows.



Part 1: Read data from the input file

The root process R reads data from the input file. However, it plans to read using **`scanf`** instead of **`fscanf`**. It opens the input file using **`fopen`**, gets the file descriptor using **`fileno`**, and redirects its **`stdin`** to this descriptor (use **`close`** and **`dup`**). All reading of data (keys) by R is done in Part 2. Write a function *initroot* that only sets up this redirection for future use.

Part 2: Build the process tree

Write a function *buildproctree* that builds the process tree (a BST with respect to the key values read from the input file). The root reads the keys sequentially from the redirected `stdin`. If it can accommodate this key (this happens only when the first key is read), it sets its key to this value, and does nothing else. Otherwise (that is, if the key of the root is already set), it passes the key to its appropriate child (left or right) depending upon whether the new key is smaller or larger than the key at the root. If that child does not exist, it is forked, and then the new key is passed to that child.

All processes perform essentially the same task. A non-root process P reads a new key from its parent pipe (only the root reads using `scanf`). If the key of P is undefined, it sets its key to the new key, and does nothing else. Otherwise, it compares its key with the new key, and based upon the comparison result, it passes the key to its appropriate child, after forking it if the child did not exist.

When a duplicate key appears in the input file, it eventually reaches a process whose old key is already set to this new key read from the file. When this happens, the key is discarded, and the insertion process stops for this new key. For synchronization (see Part 5), it may be a good idea that each process remembers how many keys it discards. This can be an additional item maintained by each process.

Eventually, the root process reads the terminating key -1. It broadcasts this key to the entire tree as follows. When a process gets the new key -1 (the root gets it from `stdin`, and any other node from its parent pipe), it forwards it to its children whichever exist(s). After this forwarding, *buildproctree* returns (to `main`).

Part 3: Compute the heights of the nodes in the process tree

Each process P in the tree invokes a function *calcheight*. It initializes $h_L = h_R = -1$ (the heights of the two subtrees). If the left child of P exists, P waits until this child writes its height to its parent pipe, and obtains the exact value of h_L. Likewise, a communication gives the exact value of h_R to P provided that the right child of P exists. P then calculates $h = 1 + \max(h_L, h_R)$. If P has a parent (that is, $P \neq R$), P writes h to its parent pipe. Finally, the function *calcheight* returns h. The return value of *calcheight* called by P is the height of the subtree rooted at P. If P is the root R, this return value is the height of the entire tree T.

Part 4: Inorder listing of the keys in the process tree

This is the last task to be done by all the processes. Write a function *inorder* that works as follows. If a process P has a left child, it waits until that child exits. Each process remembers the PIDs of its two child processes (if they exist). Use `waitpid` (not `wait`) because P wants the left child to exit first. Then, P prints (and flushes) its key. Now, P waits for the termination of its right child if it exists. Finally, P itself exits.

Part 5: Synchronization among the processes

Part 4 achieves synchronization using `waitpid` calls. This is acceptable because the inorder printing of the keys is the last task the processes do before they exit. But we need Part 2 to finish before Part 3, and Part 3 to finish before Part 4, in *all* the running processes. This cannot be achieved using `waitpid` calls.

Why is this synchronization is necessary? In Part 2, communication happens down the tree (from parent to children), whereas in Part 3, communication happens up the tree (from children to parent). Take a process P with parent Q. Suppose that Q in *buildproctree* has written some keys for P in the pipe shared between P and Q. Because of the order of scheduling of the processes (over which you do not have any control, it is an OS job), Q returns from *buildproctree* earlier than P gets a chance to read all the keys (including the final terminating key -1) sent by Q. If Q is allowed to run *calcheight* at this point, it will try to read the height of Q from the pipe between P and Q. But the pipe still contains keys not read by P, and is thus not empty. So Q mistakenly reads a key from the pipe, and treats this as the height of P. More frustratingly, if Q reads the terminating key -1 from the pipe, P never gets a chance to return from *buildproctree*. One way to avoid this problem is to use two sets of pipes, one set for downward communication, and the other for upward communication. This requires $p-1$ extra pipes, where p is the number of processes in the tree. We will soon see that this problem can be avoided by only two pipes shared by all the processes.

Likewise, the printing of the inorder listing of the keys in Part 4 must be initiated strictly after the printing of the heights of all the nodes of T (Part 3), for otherwise the printing may be garbled.

There are many ways of synchronizing processes. One possibility is that each process sleeps for some time (like one second) after returning from *buildproctree*, and again for some time after returning from *calcheight*. This idle wait gives each part to finish in all the processes before the next part starts. It is however an ugly solution, because an idle wait unnecessarily keeps processes alive longer than they should be. Moreover, in practice, you do not usually know in advance how much wait is sufficient in each synchronization step. A long sleep slows down the program, whereas a short sleep defeats the synchronization objective.

Implement a pipe-based sleepless synchronization mechanism between consecutive parts. The mechanism works because **`read`** from a pipe is a blocking call which waits until sufficient data is available in the pipe. The processes use two pipes p and q to this end. These pipes are different from the communication pipes between parent and child processes. Every process shares the knowledge of these two pipes. One particular process, preferably the root process R, plays the role of the coordinator.

Let there be n (valid) keys read by R from the file (the root should accumulate and remember this count). Let $n_d(P)$ denote the number of discarded keys at a node P (this has different values at different nodes). The root R, after returning from *buildproctree*, waits for reading a total of $n - n_d(R) - 1$ bytes of data from the pipe p. Each non-root process P, after returning from *buildproctree*, writes $1 + n_d(P)$ bytes to p. After all non-root processes finish writing to p, the wait of R is over, and the root is synchronized with the other processes. But the objective of synchronizing all the processes together is not yet achieved, because non-root processes write to p, which is a non-blocking call.

After each non-root process P writes to p, it then blocks over a read of $1 + n_d(P)$ bytes of data from the other pipe q. When the root is done with the reading from p, it writes $n - n_d(R) - 1$ bytes of data to q. This eventually releases the other processes from the blocking read on q. All the processes may now proceed (in whatever order) to the function *calcheight*. Notice that the exact data written to or read from p or q is irrelevant. It is the blocking waits that matter.

For the second synchronization (between *calcheight* and *inorder*), use a different pair p', q' of pipes (why?).

Write a function *synchronize* to be called by all the processes during each synchronization step. Pass p, q in the first step, and p', q' in the second.

The *main*() function

- When you run your code, only one process (the root R) is created. After initial book-keeping, it redirects its **`stdin`** by calling *initroot*.
- Then, call *buildproctree*. This is called by only one process, the root R. Inside the function, multiple processes are created, and these new processes keep on executing the same loop as the root. The task of each process is the same in this loop. Only the root reads from redirected **`stdin`**, whereas all non-root processes read from their respective parent pipes. A process can decide whether it is the root by looking at its parent (UNDEFINED for root, system-given PIDs for others).
- Call *synchronize* with pipes p, q after *buildproctree* returns.
- Call *calcheight*. Each process prints the return value.
- Call *synchronize* with pipes p', q' after *calcheight* returns.
- Call *inorder*. This is done by all processes. Each process terminates in this function.

Read the man-pages of the following calls: **`fileno`**, **`getpid`**, **`getppid`**, **`sprintf`**, **`sscanf`**, **`fflush`**, **`sleep`**, **`usleep`**.

Submit a single C/C++ source file. Do not use global/static variables.

Sample output

The input file is given first.

```
23   90    4   28    7   23   17   64   48   22   15   69   29   73   43   35
83   39   83   23   11   42   70   20   31
-1
```

The output of the program on this input file follows.

```
+++ Building process tree
    Process  8578 with parent    -1 gets key =   23  [ New,  assigned ]
    Process  8578 with parent    -1 gets key =   23  [ Old, discarded ]
    Process  8578 with parent    -1 gets key =   23  [ Old, discarded ]
    Process  8579 with parent  8578 gets key =   90  [ New,  assigned ]
    Process  8580 with parent  8578 gets key =    4  [ New,  assigned ]
    Process  8581 with parent  8579 gets key =   28  [ New,  assigned ]
    Process  8582 with parent  8580 gets key =    7  [ New,  assigned ]
    Process  8583 with parent  8581 gets key =   64  [ New,  assigned ]
    Process  8584 with parent  8582 gets key =   17  [ New,  assigned ]
    Process  8585 with parent  8583 gets key =   48  [ New,  assigned ]
    Process  8586 with parent  8584 gets key =   22  [ New,  assigned ]
    Process  8588 with parent  8584 gets key =   15  [ New,  assigned ]
    Process  8587 with parent  8583 gets key =   69  [ New,  assigned ]
    Process  8589 with parent  8585 gets key =   29  [ New,  assigned ]
    Process  8590 with parent  8586 gets key =   20  [ New,  assigned ]
    Process  8591 with parent  8588 gets key =   11  [ New,  assigned ]
    Process  8593 with parent  8589 gets key =   43  [ New,  assigned ]
    Process  8592 with parent  8587 gets key =   73  [ New,  assigned ]
    Process  8594 with parent  8593 gets key =   35  [ New,  assigned ]
    Process  8595 with parent  8592 gets key =   83  [ New,  assigned ]
    Process  8595 with parent  8592 gets key =   83  [ Old, discarded ]
    Process  8596 with parent  8592 gets key =   70  [ New,  assigned ]
    Process  8597 with parent  8594 gets key =   39  [ New,  assigned ]
    Process  8598 with parent  8594 gets key =   31  [ New,  assigned ]
    Process  8599 with parent  8597 gets key =   42  [ New,  assigned ]

    Height of the process  8599 with key   42 is 0
    Height of the process  8597 with key   39 is 1
    Height of the process  8590 with key   20 is 0
    Height of the process  8595 with key   83 is 0
    Height of the process  8598 with key   31 is 0
    Height of the process  8596 with key   70 is 0
    Height of the process  8591 with key   11 is 0
    Height of the process  8586 with key   22 is 1
    Height of the process  8588 with key   15 is 1
    Height of the process  8592 with key   73 is 1
    Height of the process  8587 with key   69 is 2
    Height of the process  8582 with key    7 is 3
    Height of the process  8593 with key   43 is 3
    Height of the process  8584 with key   17 is 2
    Height of the process  8589 with key   29 is 4
    Height of the process  8580 with key    4 is 4
    Height of the process  8585 with key   48 is 5
    Height of the process  8594 with key   35 is 2
    Height of the process  8583 with key   64 is 6
    Height of the process  8581 with key   28 is 7
    Height of the process  8578 with key   23 is 9  [Height of entire tree]
    Height of the process  8579 with key   90 is 8

+++ Inorder listing of the keys: 4 7 11 15 17 20 22 23 28 29 31 35 39 42 43 48 64 69 70 73 83 90
```