CS39002 Operating Systems Laboratory
Spring 2024

Lab Assignment: 7
Date of submission: 13–Mar –2024

Design your own thread library

Foosoft Inc., Barland plans for its custom-made thread library (like the pthread library). The top engineers of Foosoft are given the specifications of the library (called `foothread`). Foosoft utilities run only on Linux, so the engineers must focus on Linux-based designs. A kernel-level design allows the engineers to exploit special hardware features like compare-and-swap instructions, but that requires superuser access, so the library is decided to be built at the user level. Standard process-level synchronization tools can be used (the administration allows the use of semaphores and no other IPC primitives). No available thread library (like pthread or OpenMP) can be used.

Part 1: Specifications of the foothread library

Write two files `foothread.h` and `foothread.c`. This involves the following components.

Threads

A process may create many threads (a limit `FOOTHREAD_THREADS_MAX` should be imposed and defined in the header file). In Linux terminology, the main thread is called the **leader**. We can call all the other created threads **followers**. All the followers must have the same PID and PPID as the leader. They will however have mutually distinct TIDs (a Linux-specific feature). The followers should share almost everything (as in pthread) except the stack. The default stack size (to be defined as `FOOTHREAD_DEFAULT_STACK_SIZE` in the header file) is taken to be 2MB (2097152 bytes). Each follower may be joinable with the leader (`FOOTHREAD_JOINABLE`) or detached from the leader (`FOOTHREAD_DETACHED`). Before termination, the leader can wait for all joinable threads, but not for the detached threads. The default behavior of a follower would be detached.

Data type for foothreads

Define a data type `foothread_t`. Include, in this data type, whatever is necessary for the working of the library.

Creating threads

Write a function with the following prototype.

```
void foothread_create ( foothread_t * , foothread_attr_t * , int (*)(void *) , void * ) ;
```

The details of the created thread are stored in the first argument (NULL permitted), the second argument is for setting the attributes of the new thread, the third argument is the function where the created thread will jump to, and the fourth argument is a pointer argument to that function.

The attributes of a foothread (define a type `foothread_attr_t`) consists of two fields: the join type and the stack size. Write the following function for setting attributes. If the second argument of `foothread_create` is NULL, then only the default values are to be used. A variable of type `foothread_attr_t` can also be initialized to `FOOTHREAD_ATTR_INITIALIZER` (a macro indicating the default values). Do not use a non-NULL attribute without any kind of initialization.

```
void foothread_attr_setjointype ( foothread_attr_t * , int ) ;
void foothread_attr_setstacksize ( foothread_attr_t * , int ) ;
```

As mentioned earlier, the join type can be either `FOOTHREAD_JOINABLE` or `FOOTHREAD_DETACHED`.

Use the **`clone()`** system call with appropriate arguments to implement `foothread_create()`.

Thread termination

Recall that all the foothreads in a program share the same PID and PPID (parent PID). They are distinguished from one another by their thread-specific IDs (called TIDs). The Linux-specific system call `gettid()` returns this TID. If the program is run from a shell, the PPID of each thread is that of the shell. This implies that the leader cannot wait for the termination of the followers (the shell too can wait only for the entire process, not for its individual threads). When the starting function of a follower returns, only that thread terminates. On the other hand, when the leader leaves the `main()` function, the entire process including all created threads terminate. This is not necessarily desirable. You can synchronize the termination of all the threads using the call:

```
void foothread_exit ( ) ;
```

If a follower is `FOOTHREAD_DETACHED`, the leader cannot wait for its termination. All the joinable followers and the leader should call `foothread_exit()` near the end in order to synchronize. This call is unlike `pthread_exit()` that lets the calling threads exit. It should be used as some kind of synchronization tool. The followers should return from their start functions after calling this. The leader may do some bookkeeping work after this call, before it really `exit()`'s.

Mutexes

Define a data type `foothread_mutex_t` (to implement binary semaphores). Implement the following functions with the obvious meanings.

```
void foothread_mutex_init ( foothread_mutex_t * ) ;
void foothread_mutex_lock ( foothread_mutex_t * ) ;
void foothread_mutex_unlock ( foothread_mutex_t * ) ;
void foothread_mutex_destroy ( foothread_mutex_t * ) ;
```

Unlike Linux semaphores, foothread mutexes need to satisfy the following restrictions.

1. Only the thread that locks a mutex can unlock it. An attempt to unlock by another thread will lead to an error.
2. A locked mutex can be attempted to be locked. But then, the new requester will be blocked until the mutex is available to it for relocking.
3. An attempt to unlock an unlocked mutex should lead to an error.

Barriers

Define a data type `foothread_barrier_t` and the following functions.

```
void foothread_barrier_init ( foothread_barrier_t * , int ) ;
void foothread_barrier_wait ( foothread_barrier_t * ) ;
void foothread_barrier_destroy ( foothread_barrier_t * ) ;
```

A foothread barrier must not be used without initialization.

All the synchronization (and mutual exclusion) in the library must be implemented using only Linux semaphores. No other synchronization mechanism (like pipes) will be accepted. Calls like `sleep()` or `usleep()` are not universal synchronization primitives, and must never be used in the implementation of the library. Moreover, do not use busy waits anywhere. Finally, a plain semaphore is neither a mutex nor a barrier. Alongside a semid, the data type `foothread_mutex_t` or `foothread_barrier_t` should store additional information. Synchronization among the threads using `foothread_exit` would require some system-level global data. Declare and use these in `foothread.c` (not in `foothread.h`—the user does not need to know or manipulate these data and must be happy only with the functionality of `foothread_exit`).

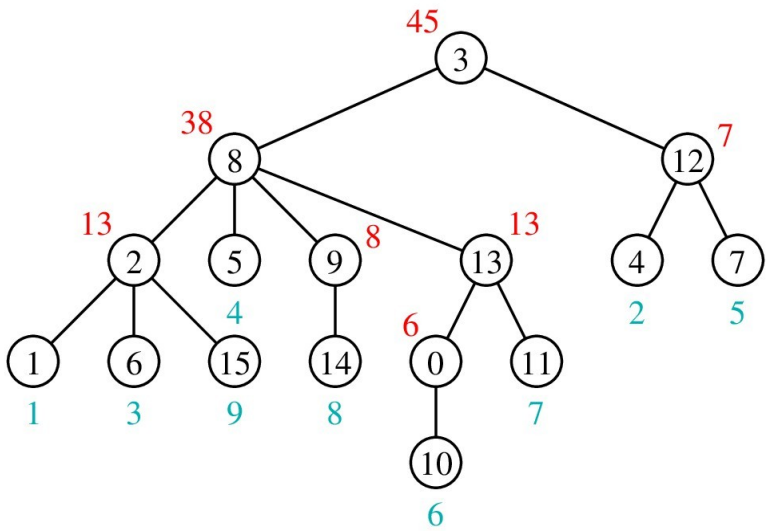
Foosoft would perhaps ask its engineers to implement condition-based synchronization. In this assignment, you do not have to do that.

Generate a dynamic library `libfoothread.so`. For compilation, a makefile is supplied. You just need to `make lib`.

Part 2. An application program

You now have a brand-new thread library in your bag. It is time to test it for correctness. Write an application program `computesum.c` that does the following.

Let T be a rooted tree with n nodes numbered 0, 1, 2, … , $n-1$. A text file `tree.txt` stores the parent representation of the tree. A 16-node tree in this representation is given below. The parent of the root is the root itself.

The file `tree.txt`	The tree
16 0 13 1 2 2 8 3 3 4 12 5 8 6 2 7 12 8 3 9 8 10 0 11 13 12 3 13 8 14 9 15 2	

The user interactively enters some positive integers at the leaf nodes, one by one. Subsequently, each internal node maintains its partial sum from its child nodes. Eventually, the root node prints the sum of all the user inputs.

In order to implement this algorithm, the leader node first reads n and the parent array $P[\,]$ from the input file `tree.txt`. It then sets up the synchronization resources to be shared by the followers. It then creates n follower threads, one for each node in the tree. The leaf nodes read the user inputs, and update the sums of their respective parents. An internal node, after all its children update its sum, adds this sum to that of its parent. Eventually, the root node gets the total sum of all the user

inputs, and prints the sum. After this, the leader cleans up the resources (using the destroy functions of the library), and then terminates. Use only the foothread synchronization functions (mutexes and barriers, and absolutely nothing else) throughout `computesum.c`. Use `foothread_exit` (not new barrier(s) and/or mutex(es)) to synchronize the threads at the end.

Other files

A random-tree generator `gentree.c` is supplied to you. You can run it to generate random trees. You can pass *n* (the number of nodes) as the only command-line argument. The default value of *n* is 25.

Additionally, a makefile is provided to do the following tasks.

`make` (or `make lib`)	Create `libfoothread.so`
`make app`	Compile `computesum.c`
`make run`	Compile and run `computesum.c` (assuming that `tree.txt` is available)
`make tree`	Compile `gentree.c` (This only generates `./gentree`. Run it separately.)
`make newrun`	Run with a new tree (after all necessary compilations)
`make clean`	Do this before your submission

What to submit

A single zip file containing `foothread.h`, `foothread.c`, and `computesum.c`. The other files supplied may also be there in the archive, but do not submit any binary files (see `make clean`).

Sample

A run on the example given in the text may proceed as given below.

```
$ cp TREE.txt tree.txt
$ make run
gcc -shared -Wall -o libfoothread.so foothread.o
gcc -Wall -Wl,-rpath=. -I. -L. -o computesum computesum.c -lfoothread
./computesum
Leaf node   1 :: Enter a positive integer: 1
Leaf node   4 :: Enter a positive integer: 2
Leaf node   6 :: Enter a positive integer: 3
Leaf node   5 :: Enter a positive integer: 4
Leaf node   7 :: Enter a positive integer: 5
Leaf node  10 :: Enter a positive integer: 6
Leaf node  11 :: Enter a positive integer: 7
Leaf node  14 :: Enter a positive integer: 8
Leaf node  15 :: Enter a positive integer: 9
Internal node  12 gets the partial sum 7 from its children
Internal node   0 gets the partial sum 6 from its children
Internal node   9 gets the partial sum 8 from its children
Internal node   2 gets the partial sum 13 from its children
Internal node  13 gets the partial sum 13 from its children
Internal node   8 gets the partial sum 38 from its children
Internal node   3 gets the partial sum 45 from its children
Sum at root (node 3) = 45
$
```