CEng 536 - Advanced Unix

Fall 2016-2017
HW 2
Due: 5/12/2016, 23:55

In this homework you are going to implement a socket based 2-D map locking service. Clients connecting a unix domain socket requests reader/writer locks of arbitrary rectangular regions in a hyphotetical map. When a rectangular area is locked for writing, no other lock can be given to any other intersecting lock requests. When a rectangular area is locked for reading, only read lock requests to the intersecting regions are given.

When a lock cannot be immediately given to a request, request blocks until the area can be locked. Unlocking of a region can unblock multiple blocked requests if they are compatible. If they are not, the system unblocks any of them.

Your implementation should create a new process per connection, thus a multi-process server. Main process listens socket connection and for each `accept()` a new child is created that we will call it an *agent*. The agent reads text based service requests from client and serves them. Each command is given on a single line ended by '\n'. The following commands should be implemented:

`LOCKR` ***xoff yoff width height***
Requests a read lock on the regions starts at ***xoff yoff*** with given width and height. All coordinates are `double` values. Lock request is blocked if there is an intersecting region already locked for writing. It will be unblocked when that region or any following intersecting write lock is unlocked. On success an integer id for the lock is sent to the client. This id is used in `UNLOCK` requests.

`LOCKW` ***xoff yoff width height***
Requests a write lock on the region. Write lock request is blocked if there is at least one intersection region is either read or write locked. It will be unblocked when no intersecting region left locked.

`TRYLOCKR` ***xoff yoff width height***
Non-blocking version of `LOCKR`. Immediately returns with success or failure. Success outputs integer id of the lock, failure outputs '`Failed`' to the client.

`TRYLOCKW` ***xoff yoff width height***
Non-blocking version of `LOCKW`. Immediately returns with success or failure.

`MYLOCKS`
Report all locks hold by the current connection. Each lock is output on a seperate line with its id.

`UNLOCK` ***id***
Unlock the region with lock id '***id***'. ***id*** should belong to one of the locks hold by the current connection.

`GETLOCKS` ***xoff yoff width height***
List existing intersecting locks on the region. It outputs each intersecting region on a line with type of the lock. Locks hold by the current connection are included in the output.

`WATCH` ***xoff yoff width height***
Report all successfull lock/unlock events intersecting the region. This commands work in background. While watching events, client can make other requests. Multiple `WATCH` requests can be active at a time. Each `WATCH` command is given an integer id and it is output as the first line of the response.

`WATCHES`
: List active watches of the client. Each watch is output on a single line.

`UNWATCH` *`id`*
: Stop watching the area specified in `WATCH` command with the *`id`*.

A sample client session is given below as a hint to output formats.

```
LOCKW 10.0 12.23 10.0 20.5
1

LOCKR 30.0 35.03 10.0 20.5
2

MYLOCKS
1 W 10.0 12.23 10.0 20.5
2 R 30.0 35.03 10.0 20.5

TRYLOCKW 10 100 20 20
 Failed

TRYLOCKW 100 10 20 20
3

UNLOCK 2
Ok

UNLOCK 4
 Failed

MYLOCKS
1 W 10.0 12.23 10.0 20.5
3 W 100 10 20 20

GETLOCKS
W 10.0 12.23 10.0 20.5
W 10.0 30.44 10.0 20.0
R 20.0 25.03 10.0 20.5
R 22.0 12.23 10.0 20.0
W 30.0 30.0 10.0 30.0
R 30.0 30.0 40.0 20.5
W 100 10 20 20

WATCH 10 10 100 100
1

WATCH 20 90 200 200
2

WATCHES
1 10 10 100 100
2 20 90 200 200

 Watch 1 30 30 10 10 R locked
 Watch 1 90 90 20 10 W locked
 Watch 2 80 80 20 10 W locked
 Watch 1 30 30 10 10 R unlocked
```

```
UNWATCH 4
 Failed

UNWATCH 1
Ok

Watch 2 80 80 20 10 W unlocked

BYE
```

In list outputs if there is an id, output is sorted on ids. Otherwise, output is sorted on xoff, yoff, width, height, read then write. Clients can enter deadlock while they lock regions. You don't need to do anything to detect or avoid it.

Note that there is no synchronization primitive for region based locks provided by the operating system. You need to keep a data structure for locks currently hold by all agents. All agents inspect/update this data structure concurrently. Since this data structure should be accessed by multiple processes, it needs to be on a shared memory region. Your basic task is to implement this concurrent data structure on a shared memory. Implementations without a shared memory is possible but not acceptable. Since shared memory allocation is fixed, maximum number of locks at any instance of time is limited by 100,000. You can also assume the map is limited to `0.0-1000.0`$\times$`0.0-1000.0`

You can use any IPC mechanisms you like as long as you stay within the standart system calls and libraries. You can use `pthread` and threads inside of an agent process, but not allowed to implement agents as threads.

A linear structure like a list or array for locks take $\mathcal{O}(\backslash)$ for intersection queries. Such an implementation is sufficient for this courses objective. However if you are interested in implementing a spatial tree structure and reduce cost to $\mathcal{O}(\Updownarrow\wr\}\backslash)$ you are going to take a 20 points bonus.
Command line of your code expects a file name to a Unix domain socket path as:
`maplock` *`socketpath`*

You need to submit a tar.gz (no zip, rar or others) file containing a `Makefile` and sources in directory named your METU userid. `Makefile` will create binary `maplock` for your homework. Following this specifications help me evaluating your homework so please obey them.
You can use `metuclass.metu.edu.tr` to submit your homework.