Synchronisation 3 and Practical Applications

Read-Write Locking On Dynamic Array

You have been tasked with implementing a read-write dynamic array. Any add or remove function will be a write operation while a get function will be considered a read operation.

You will need to arrange different test cases that will look into different read-write ratios. Your test cases should check that you can correctly add, remove, insert and retrieve elements from the dynamic array and your functions are thread safe.

Afterwards, implement benchmarks with the following read-write ratios with your data structure and compare against your coarse grained implementation. Consider giving your data structure a set number of elements at the beginning.

- 0% Write, 90% Read
- 40% Write, 60% Read
- 50% Write, 50% Read
- 60% Write, 40% Read
- 90% Write, 10% Read

```
struct dynamic_array;
struct dynamic_array* dynamic_array_new(size_t initial_capacity);
void dynamic_array_add(struct dynamic_array *ary, void*value);
void* dynamic_array_get(struct dynamic_array *v,
size_t index);
void* dynamic_array_remove(struct dynamic_array *ary,
size_t index);
void dynamic_array_destroy(struct dynamic_array* ary);
```

Busy-Waiting Barrier

Synchronisation primitives are all the rage these days and in this task, you will need to construct a barrier. Use the following struct to help guide your implementation. This synchronisation primitive will keep threads waiting until all threads have arrived at the barrier. Once ready, they will be released to continue with their execution.

```
struct sync_barrier {
    uint16_t threshold;
    atomic_uint counter;
    atomic_uint release_count;
};

struct sync_barrier* sync_barrier_new();
int sync_barrier* sync_barrier_wait(struct sync_barrier*);
void sync_barrier_destroy(struct sync_barrier*);
```

Case-Study - FTP Server (Optional)

For this task, you are required to construct a simple FTP server that handle a small set of operations (retrieving and uploading files). The below is a list of commands clients can send to your server.

- LIST - Gets the list of files and directories at the root of the server
- RETR `<path>` - Retrieves a copy of the file located at the path on the server
- STOR `<path>` - Server will create a file at path and accept data from the client to be written to the file
- DELE `<path>` - When received from the client, it will delete a file on the server
- RNFR `<filename>` - Sets up a file to be renamed
- RNTO `<new-filename>` - After sending RNFR, RNTO will rename the file to new-filename
- SIZE `<path>` - Retrieves the filesize of in bytes
- QUIT - Closes the connection

On top of implementing the above protocol, your program should accept the following command line arguments.

```
./ftpserver :PORT :DIRECTORY
./ftpserver 9001 ./myfiles
```

The above `:PORT` should specify the port the network port the application will listen on. The directory will be the folder that will be served to the clients.

The above specifies enough commands to run a simple FTP server in anonymouse mode. However, we need to test out server. Use the command nc or netcat to connect to your server and run the commands specified.

You can use the following scaffold to help get started.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdint.h>

int main(int argc, char** argv) {

    int serversocket_fd = -1;
    int clientsocket_fd = -1;
    int port = 9000;
    struct sockaddr_in address;
```

```
    int option = 1;
    char buffer[1024];
    serversocket_fd = socket(AF_INET, SOCK_STREAM, 0);

    if(serversocket_fd < 0) {
        puts("This failed!");
        exit(1);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(port);
    setsockopt(serversocket_fd, SOL_SOCKET, SO_REUSEADDR |
    SO_REUSEPORT, &option, sizeof(int));

    if(bind(serversocket_fd, (struct sockaddr*) &address,
        sizeof(struct sockaddr_in))) {

        exit(1);
    }

    listen(serversocket_fd, 1024);
    while(1) {
        uint32_t addrlen = sizeof(struct sockaddr_in);
        clientsocket_fd = accept(serversocket_fd,
        (struct sockaddr*) &address, &addrlen);
        read(clientsocket_fd, buffer, 1024);
        puts(buffer);
        close(clientsocket_fd);
    }
    close(serversocket_fd);
    return 0;
}
```

Using Threads (Optional)

Instead of handling each connection synchronously, spawn a thread or use a thread pool to handle multiple connections at once. Ensure that each thread has a property the main thread can see to check if the thread has finished. This will allow the main thread to rejoin threads that have finished their work.

Are threads always the answer? (Optional)

Modify your original server to use the poll call. Instead of using multiple threads to process con- nections asynchronously, the poll function and a file descriptor set can be use to monitor events on the sockets.

Monitor each file descriptor with POLLIN and POLLHUP. POLLIN events are generated when a file descriptor has data.

```
struct pollfd fds[3];
size_t nfds = 3;

// Set each file descriptor
//fds[i].fd = fd;
//fds[i].events = POLLIN | POLLHUP;
//
// Maybe change the file descriptor to use non-blocking

while(poll(fds, nfds, -1) >= 0) {
    for(size_t i = 0; i < nfds; i++) {
        if(fds[i].revents & POLLIN) {
            //What to do if POLLIN was received on
            //this file descriptor
        }
    }
    //event loop to check
}
```