Task #1: DNS proxy server

Write DNS proxy server with a domains blacklist feature to filter unwanted host names resolving.

Functional requirements

1. The proxy server reads its parameters during startup from the configuration file.
2. The configuration file contains the following parameters:
    - IP address of upstream DNS server
    - List of domain names to filter resolving ("blacklist")
    - Type of DNS proxy server's response for blacklisted domains (not found, refused, resolve to a pre-configured IP address)
3. DNS proxy server uses UDP protocol for processing client's requests and for interaction with upstream DNS server.
4. If a domain name from a client's request is not found in the blacklist, then the proxy server forwards the request to the upstream DNS server, waits for a response, and sends it back to the client.
5. If a domain name from a client's request is found in the blacklist, then the proxy server responds with a response defined in the configuration file.

Non-functional requirements

1. Programming language: C
2. It is allowed to use third-party libraries or code. If you use one then you must comply with its license terms and conditions. The only exception is that it is not allowed to use third-party libraries or code that implement DNS parsing and constructing DNS packets.

All other requirements and restrictions are up to your discretion.

Expectations

What we expect to receive from you when you complete this task:

1. Source code of DNS proxy server written according to the requirements. Source code should be in a plain ASCII text (it is not a joke - sometimes we receive source code in MS Word document).
2. Instruction how to build, configure and run the proxy server (including necessary files like Makefile etc).
3. Description how you tested the proxy server.
4. Document, if your solution has additional (known to you) limitations or restrictions, or it has more features that it was requested.

If you have GitHub (or similar) account then you can publish everything there; otherwise, put everything into an archive.

Task #2: Linux kernel module

Write Linux kernel module that sets watchpoint to a given memory address, and if this memory address is accessed then module's callbacks are called.

Functional requirements

1. Memory address to monitor is set by a module param and can be changed through sysfs module's entry.
2. The module sets watchpoint (hardware breakpoint) to a given memory address.
3. When the memory address is accessed (either read or write), then module's callbacks are called (separate for read and write) and a backtrace is printed

Non-functional requirements

1. Linux kernel module should be built by Yocto for qemux86.
2. Linux kernel module should have its own Yocto recipe.

All other requirements and restrictions are up to your discretion.

Expectations

What we expect to receive from you when you complete this task:

1. Source code of Linux kernel module written according to the requirements. Source code should be in a plain ASCII text (it is not a joke - sometimes we receive source code in MS Word document).
2. Yocto recipe for the kernel module.
3. Instruction how to build and run the module.
4. Description how you tested the kernel module.
5. Document, if your solution has additional (known to you) limitations or restrictions, or it has more features that it was requested.

If you have GitHub (or similar) account then you can publish everything there; otherwise, put everything into an archive.

Task #3: FAT32 emulator

Write FAT32 emulator program that manages FAT32 filesystem backed up by a file.

Functional requirements

1.The emulator program works with a regular file that backs up FAT32 filesystem.
2.Path to the file is provided to the emulator as a command line parameter.
3.The emulator should create the file of size 20 MB if it does not exist.
4.The emulator provides CLI (command line interface) with commands: cd, format, ls, mkdir, touch. The commands operate on FAT32 filesystem.
5.CLI shows a prompt "/path>" (without quotes) where "path" is the current directory.
6.Command "cd <path>" changes the current directory to the specified directory. Only absolute path is allowed. If a path does not exist then an error is shown, and the current directory is not changed.
7.Command "format" creates FAT32 filesystem inside the file. All information is lost.
8.Command "ls [path]" shows files and directories inside a given directory, or in the current directory if a path is not specified.
9.Command "mkdir <name>" creates a directory with a given name in the current directory.
10.Command "touch <name>" creates an empty file with a given name in the current directory.
11.If the backed file does not contain a valid FAT32 filesystem then all commands (except for "format") should fail and output an error.

Example:
./f32disk /home/user/test.disk
/>ls
Unknown disk format
/>format
Ok
/>ls
. ..
/>mkdir ttt
Ok
/>ls
. .. ttt
/>cd /ttt
/ttt>ls
. ..
.ttt>cd /
/>

Non-functional requirements

1.Programming language: C

2.It is allowed to use third-party libraries or code. If you use one then you must comply with its license terms and conditions.

3.The emulator should not be a wrapper around *mkfs*, *mount* and similar utilities.

All other requirements and restrictions are up to your discretion.

Expectations

What we expect to receive from you when you complete this task:

1.Source code of FAT32 Emulator written according to the requirements. Source code should be in a plain ASCII text (it is not a joke - sometimes we receive source code in MS Word document).
2.Instruction how to build, configure and run the emulator (including necessary files like Makefile etc).
3.Description how you tested the emulator.
4.Document, if your solution has additional (known to you) limitations or restrictions, or it has more features that it was requested.

If you have GitHub (or similar) account then you can publish everything there; otherwise, put everything into an archive.