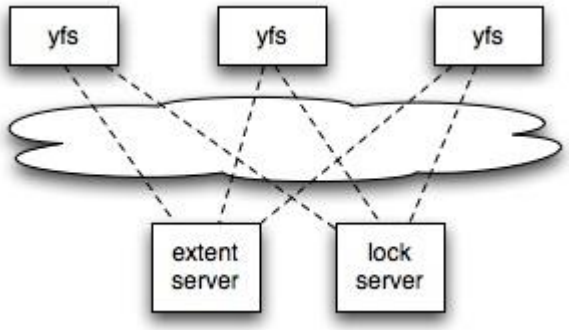
Lab Information

Lab Overview

In this sequence of labs, you'll build a multi-server file system called Yet Another File System (yfs) in the spirit of Frangipani. At the end of all the labs, your file server architecture will look like this:



You'll write a file server process, labeled yfs above, using the FUSE toolkit. Each client host will run a copy of yfs. Each yfs will create a file system visible to applications on the same machine, and FUSE will forward application file system operations to yfs. All the yfs instances will store file system data in a single shared "extent" server, so that all client machines will see a single shared file system.

This architecture is appealing because (in principle) it shouldn't slow down very much as you add client hosts. Most of the complexity is in the per-client yfs program, so new clients make use of their own CPUs. The extent server is shared, but hopefully it's simple and fast enough to handle a large number of clients. In contrast, a conventional NFS server is pretty complex (it has a complete file system implementation) so it's more likely to be a bottleneck when shared by many NFS clients.

Yfs will use processes to simulate the distributed environment thus you will not meet any difficulties to setup the programming tools in a single machine.

Programming Environment

Yfs uses the fuse tool to build the general purpose file system. Linux is the natural choice. Any distribution of Linux should be fine. You should be able to install the fuse module, library and headers on your own machine by following the instructions at fuse.sourceforge.net. Linux is suggested and BSD or MacOS should also be fine.
You should setup the programming environment by yourself. You can either use a physical machine or a virtual machine. If you meet some difficulties on setting the programming environments for labs, you can contact the TA to help you.

Installing FUSE on Ubuntu

To setup FUSE on your local machine, you will need the header files and the utilities. Install them like this:

```
sudo aptitude install libfuse2 fuse-utils libfuse-dev
```

Next, you need to make sure the fuse module is loaded:

```
% ls -l /dev/fuse
   crw-rw-rw- 1 root fuse 10, 229 2010-02-11 06:02 /dev/fuse
```

If your ls output matches the above output, then you can skip the modprobe step. If you do not see the above output, try running modprobe:

```
% sudo modprobe fuse
```

Finally, you need to add yourself to the fuse group to be able to mount FUSE file systems:

```
% sudo adduser {your_user_name} fuse
```

Make sure to logout of you current session to refresh your group list. When you logged in again, typing groups at the command line should show fuse as one of the groups:

```
% groups
   yourname users fuse admin
```

Aids for working on labs

There are a number of resources available to help you with the lab portion of this course:

C++ Standard Template Library:

You will use C++ STL classes such as map and string a lot, so it's worth while to be familiar with the methods they offer; have a look http://www.cplusplus.com/reference/.

pthreads:

All the labs use the POSIX threads API (pthreads). A comprehensive guide to programming with pthreads can be found here: http://www.llnl.gov/computing/tutorials/pthreads/

FUSE:

The labs use the FUSE interface to plug the lab file system into the operating system. See the FUSE website for more information. http://fuse.sourceforge.net/

Debugging and Core files:

- printf statements are often the best way to debug. You may also want to use gdb, the GNU debugger. You may find this gdb (http://www.delorie.com/gnu/docs/gdb/gdb_toc.html) reference useful. Below are a few tips.
- If your program crashes and leaves a core dump file, you can see where the crash occured with gdb program core, where program is the name of the executable. Type bt to examine the call stack at the time of the crash.
- If you know your program is likely to have a problem, you can run it with gdb from the beginning, using gdb program. Then type run.
- If your program is already running (or it is hard to start with gdb due to complex start-up scripts), you can attach gdb to it by typing gdb program 1234, where 1234 is the process ID of your running program.
- While in gdb, you can set breakpoints (use the gdb command b) to stop the execution at specific points, examine variable contents (print ...), get a list of threads (info threads), switch threads (thread X), etc.
- To apply a given gdb command to all threads in your program, prepend thread apply all to your command. For example, thread apply all bt dumps the backtrace for all threads.
- Look at the GDB manual for full documentation. http://www.gnu.org/software/gdb/documentation/