ADVANCED OPERATING SYSTEMS
Milestone 7: Nameserver
Fall Term 2016

Assigned on: **25.11.2016** Due by: **16.12.2016**

1 Overview

In this milestone you will bring together the bits and pieces you've implemented so far and make the last steps towards a complete operating system.

This particular part of the final project is building a name server, which is a key central component of any non-trivial distributed system and, increasingly, most operating systems as well.

You have already seen how different functionality in a microkernel or multikernel is implemented in server processes. Name servers address the problem of how a process that provides a service (the server, in other words) can be discovered and contacted by a process that wants to use that service (the client). The way a name server works is that other processes in the system tell the name server about how to bind to the services they provide, and also ask the name server about how to bind services they need. In this way the name server acts as a "clearing house": matching offers of services to requests for them.

This exercise will be the last exercise where you'll need to implement something. We will leave you much freedom in how you do the things we want from you as a minimal requirement, and also give you a lot of room for additional bonus objectives. In any case, try to finish the minimal requirements first before starting with other projects.

The work consists of:

- Creating a separate server process (the *nameserver*) which can respond to requests
- Providing *lookup* functionality: clients should be able to connect to the name server, submit name lookup requests, and receive back enough information to allow them to connect to another service.
- Providing *registration* functionality: servers should be able to connect to the name server and register their service IPC endpoint with the server under a particular name, so that clients can then query it.
- Create a mechanism for *bootstrapping* the name service: since a new client can't look up the name of the nameserver, there has to be a mechanism for any process to acquire a connection to the nameserver when it starts

- Demonstrating all other processes in the OS as a whole using the nameserver rather than other mechanisms for obtaining IPC channels.

2 Getting prepared

Conceptually, a name server can be as simple as a hash table from strings to endpoint identifiers (for example, capabilities). The subtlety is in the design of the naming scheme, and how the name server interacts with the rest of the system.

Name servers is modern *distributed* systems support many complex features like replication (for performance or fault tolerance), delegation (such as DNS' hierarchy), security (not all clients of the name server can see all services registered with it), etc. For your OS, these extra features are *not* required - a small OS simply doesn't need them.

However, even small microkernel or multikernel-based OSes really need some kind of name server to be practical. Since your fellow team members are implementing other functionality for their projects, it's quite likely that they will find early access to your name server's code useful for their own work.

There is no pre-existing code required for this milestone. All you should need is the code that you and the rest of your team have written up to this point. However, by the end you should also be able to integrate the code that your fellow team members are writing (see below).

Instead, the name server will be written "from scratch", but building on the communication primitives you have already seen. The challenges are in figuring out how to use these communication operations to implement the name server functionality, and how to organize the name server and it's API.

This process may uncover deficiencies and limitations in the code you have already written, although if you have designed and implemented your previous milestones well enough, this should not be problem. If it is, you should be prepared to fix these issues, but also talk about what you did in the project report and why it was required.

We suggest you start with a very simple example server which can be contacted by other processes and serve requests, and then extend your code from there into a fully-fledged name server.

You will also want to write a small library that clients of the name server use to communicate with it, whether they are registering a service or looking one up. In the rest of this handout we'll assume the API for this library is defined in a header file `include/nameserver.h`; part of your job is to write this header file and design the interface.

3 Registration

The first functionality to provide in the name server is registration of new services.

```
/**
* @brief register a name binding
*
* @param name  the name under which to register the service
* other parameters: the service endpoint itself
*
```

```
* @return SYS_ERR_OK on success
*         errval on failure
*/
extern errval_t register(char *name, ...);
```

The `register` function is called by a server (for example, a file system server) to announce that it's providing a service that clients can connect to. The first argument is the name that the service will be registered under. The remaining argument (or arguments) specify how to contact the service - for example, they might be a capability to local message passing channel.

It is up to you to decide on this representation. Note that it needs to be something you can pass over message channels, but also should be all that a client needs to connect to the service.

It is also up to you to design the *name space* to be used. A bare minimum is simply flat strings (such as "`memserver`" or "`face_detect`"); but even here you need to specify what is a valid name and what is not. For example, what kinds of characters are allowed in names? What about whitespace?

More advanced name servers use a hierarchy, like DNS or POSIX file system names, with some character separating *components* of the name (for example, "`/fs/sdcard`"). This makes it easier to manage the name space since different areas of functionality can be given different subtrees of names.

EXTRA CHALLENGE
Really sophisticated name servers support much more than simple strings for naming – they are closer to structured search engines. For example, it is quite common to use a set of key-value pairs instead of a single string, such as:

```
type=ethernet
mac=44:8a:5b:d3:b8:07
name=eth0
speed=1G
```

The lookup operation (see below) is then a search for records which satisfy some query, such as "An ethernet interface with a speed of 1G". If simple unique names are enough, they are a subset of this functionality (such as searching for a record with field `name` equal to `eth0`).

Such sophistication might not be needed for your system, but you might want to see how easy it is to implement it in some way.

As well as registering services, you should allow a server to deregister them - remove any record of them from the name server, using the `deregister` call.

```
/**
* @brief deregister a name binding
*
* @param name  the name under which the service was registered
*
* @return SYS_ERR_OK on success
*         errval on failure
*/
extern void deregister(char *name);
```

EXTRA CHALLENGE
Of course, if a server process crashes, then no client can contact it, even though the name server might have a record for it. Ideally, the name server would detect that the service has gone away itself and remove the "dead" service record.

Also, as specified above, anyone can call `deregister` with any name. It would be better if this could only be done by the server which originally called `register` for that name, or by an administrator account.

4 Lookup

Once you can register (and deregister) services with particular names, the next step is to allow clients of the services to look them up by name in the name server:

```
/**
* @brief lookup a name binding
*
* @param query  the query string to look up
* other parameters: the returned service endpoint itself
*
* @return SYS_ERR_OK on success
*         errval on failure
*/
extern errval_t lookup(char *query, ...);
```

`lookup` should take (at least) two arguments. The first is a query string. At its simplest, this query is simply a well-known name to lookup (as in the examples in the previous section), and we suggest you implement this simple interface first, before allowing more sophisticated lookup strings.

The remaining arguments are outputs, and return something that can be used to connect to the service (assuming the lookup succeeds). Naturally, if the name is not bound in the server, an appropriate error code should be returned.

5 Bootstrap

Now that you have implemented registration and lookup, you have the basic functionality required to use the name server. However, a remaining problem is that of how each process acquires an initial connection to the name server.

Consequently, the next step is to make sure that, after a small number of initial processes (including the name server itself) have started, *all* new processes start with a way to connect to the name server.

You have most likely already solved this problem before in the case of the memory allocator. It should be possible to go back and modify the rest of the OS so that processes can locate the memory server after they have started up by first contacting the name server.

6 Enumeration

By this point, you have almost certainly written debugging code to dump the state of the name server to make sure that registrations, etc. have worked.

The next step is to allow clients to obtain a list of all the names bound in the name server, or (better still) all names matching some search expression:

```
/**
* @brief lookup a name binding
*
* @param query  the query string to look up
* @param num    returns the number of names returned
* @param result array of <num> names returned
*
* @return SYS_ERR_OK on success
*         errval on failure
*/
extern errval_t enumerate( char *query, size_t *num, char **result );
```

Note that implementing this will require you to return an unpredictable (and potentially large) number of strings from the name server, which should probably be done using multiple messages.

7 Getting the rest of the system to use the nameserver

You have already implemented the core of a small operating system in the course of the project, and your project team partners will also be working on other components (such as a filesystem, shell, or network stack).

You should, as much as possible, get as much of the complete OS to use the name server as you can. This means that almost any service which can be contacted using inter-process communication (Local Remote Procedure Call on the same core, or User-level Remote Procedure Call between cores) should, by the time you are all finished, use the name server to register the services it offers, and look up the services it needs to use.

This isn't possible in all cases (for example, because the name server is itself a service that clients have to find), and you should make it clear in the report on your design where these limits are, if they could be overcome, and whether trying to do so would be a good idea.

However, ideally all new processes after the initial set should *only* need to start by contacting the name server. After that, all other services they should need can be obtained by looking up well-known names (or queries).

You should identify the minimal set of processes which, when the OS boots, need to have more than just a way to connect to the name server, and this set (and your reasoning!) should be documented in your report.

8 Lab demonstration

You are expected to demonstrate the following functionality:

- An implementation of a name server supporting the four operations `register`, `deregister`, `lookup`, and `enumerate` above.
- The name server is started when the machine boots, and runs until the machine is shut down.
- All OS services register with the name server when they start up
- As many OS processes as possible use the name server to find out how to contact any other services that they need

You can demonstrate this in part by running a program (ideally, interactively from a shell that one of your fellow team members has written) that can list the services known in the name server.