**CS 503 - Fall 2014**

**Lab 6  
Final Project  
Dynamic Program and Library Loading**

**Due Sunday, December 7th, 2014 at 11:59 PM**

**Midway submission due Sunday, November 23rd, 2014 at 11:59 PM**

**Objectives:**

By the end of this lab you will be able to:

* Understand how to dynamically load programs and libraries into memory.
* Understand how to alter the location for relocatable variables.
* Create new system calls to manipulate shared libraries.

**Background**

Today operating systems such as Linux and Windows support the execution of programs from a backing storage. The typical helloworld program when compiled by gcc creates an executable that can be executed from a shell (the default name generated by gcc is typically a.out). However in XINU, all processes are created and executed using a function pointer. There is currently no way to execute a program that has been compiled outside of XINU. Your job is to provide a mechanism and interface to allow for the execution of dynamic code in XINU. You will do this in two parts:

* Part 1: The loading and execution of a program that is stored outside of the XINU executable.
* Part 2: The dynamic loading of libraries that contain functions that can be called from within XINU.

**File Storage**

Executable and library files will be stored in XINUs remote file system (RFS). The RFS consists of two pieces:

* The rfserver - a network server that runs on a XINU frontend (e.g. xinu19.cs.purdue.edu)
* The RFS client within XINU - runs within XINU on the backend

The rfserver, which runs on your frontend work station, serves files to the XINU instance running on the backend. XINU connects to the rfserver through a UDP port, and sends/receives messages to perform file operations on the file system where the rfserver is running. For this lab, all you will need to use the RFS to read the binary images for the programs and the libraries.

The source code for the rfserver is located inside the tarbal for the lab (see below) in the rfserver directory. You will be provided with a UDP port number specifically for you to use for the RFS server via email. You will need to change the macro "RF\_SERVER\_PORT" in the file include/rfilesys.h

To allow XINU to connect to the rfserver, the IP address for the server needs to be set in include/rfilesys.h. The value of RF\_SERVER\_IP must be changed to match the IP address of your frontend workstation. For example, if your frontend workstation is xinu19.cs.purdue.edu then the RF\_SERVER\_IP value in rfilesys.h needs to be changed to 128.10.3.69

Running make in the directory rfserver/ generates the executable "rfserver" for the RFS server and places it in the programs/ directory. You will need to run this code before you start your XINU image on the backend, like so:

$ ./rfserver

NOTE: Every time you change rfilesys.h make sure you do a make clean and then make prior to running your xinu image on a backend

To open a file in the remote file system, you will use the open system call specifying the name of the file that you wish to open and the RFILESYS as the device name (this tells XINU that you are opening a new file in the remote file system). For example:

int fd = open(RFILESYS, "helloworld", "or");

This opens the file helloworld, located in the remote file system, for reading ("r" tells XINU to open for reading, "o" tells XINU to ensure that file exists, if it does not exist SYSERR is returned). The resulting file descriptor is stored in fd. If there is a problem opening the file (timeout connecting to the rfserver, etc.), then fd will contain the value SYSERR.

NOTE: If you do not use "o" in file open mode, a new file will be created.

NOTE: the remote file server creates a file system that is relative to the directory where the rfserver is executed from. For example, if your current working directory on your frontend workstation is xinu-fall2014-lab6/programs and you run the rfserver command, then the root directory that XINU will see on the open system call will be xinu-fall2014-lab6/programs on the frontend workstation.

NOTE: do NOT use a beginning slash '/' in your file, or '..' anywhere in file path. Doing so will cause open to fail. The RFS does not use a root directory with '/'. The remote file server will always look for your path in the directory starting with the current working directory when rfserver was executed.

Assuming the file opens successfully, you can later read the file with the read system call, specifiying the file descriptor, a read buffer, and read size. For example:

char read\_buffer[300];

int rc = read(fd, read\_buffer, sizeof(read\_buffer));

This will return the number of bytes actually read from the file or SYSERR if an error occurred.

You can request the size of the file in bytes by using the control function like so:

int32 filesize = control(RFILESYS, RFS\_CTL\_SIZE, fd, 0);

Here fd is the file descriptor returned by open(..)

Hint: To read the entire file, you can first find out the size of the file, allocate enough memory to hold the entire file and then read the entire file.

**File Format**

Compiled files will be created by gcc in the [executable and linkable format (ELF)](http://en.wikipedia.org/wiki/Executable_and_Linkable_Format). For part 1 (loading of an executable) the file will contain a single function with the symbol name "main". For part 2, the file may contain one or more functions of different names.

As part of this project you will have to familiarize yourself with the ELF format and be able to parse the headers and various sections of the ELF file.

Things to consider in the ELF file:

* A loaded program can make calls to existing XINU functions (e.g kprintf). When the compiler compiles the program, it does not know the address of kprintf. The compiler builds relocation entries for the symbols that it cannot resolve at compile time. These functions are present in the XINU text section at runtime. Your job is to read the relocation entries in the ELF file and fix up the relocation offsets as part of the load\_program and load\_library functions. For more information on relocation, see the ELF specification (links provided in references section).
* When a program is compiled resulting in an ELF file, the file contains a section called ".symtab" (symbol table). The symbol table contains the names and addresses of all symbols in the executable. As you know, XINU is also compiled into an ELF file (xinu.elf). At the time of resolving relocations in the loaded program, you will need the addresses of XINU functions. Your task is to read the xinu.elf file (using RFS), specifically the symbol table in the xinu.elf file. You only need to do this once during a run.
* When you run make from the compile/ directory to compile XINU, the resulting xinu.elf file is also placed in the programs/ directory. You will open this xinu.elf file using RFS, to read the XINU symbol table.

**Part 1: Running a program executable in XINU**

Recall the format for the create system call:

pid32 create(

void \*funcaddr, /\* Address of the function \*/

uint32 ssize, /\* Stack size in words \*/

pri16 priority, /\* Process priority > 0 \*/

char \*name, /\* Name (for debugging) \*/

uint32 nargs, /\* Number of args that follow \*/

...

)

Note, that the create system call takes as the first parameter the pointer to the function that the new process will execute. In part 1, you need to create an interface to allow a program to be loaded into XINU's memory from a program located in a file system. To do this you will be required to write a new system all with the following format:

void\* load\_program(char\* path);

This system call takes as a parameter the path name to a program and returns a function pointer to the program that has been loaded into memory. If the operating system could not load the program (e.g. the file doesn't exist, there is no more memory to load the program into, etc.) then the function returns SYSERR cast to a void pointer:

(void\*)SYSERR

As previously mentioned the program to be loaded must contain only a single function with the symbol named "main". If more than one function exists in the file, then return (void\*)SYSERR.

NOTE: the name "main" for the symbol of the function is only used to find the location of the program entry point. Once load\_program system call returns, the pointer to the function will be used.

Once the program is loaded, the program can then be executed by using the create system call. For example, consider the following "helloworld" program:

#include <xinu.h>

int32 main(int32 argc, char\* argv[])

{

kprintf("Hello World\n");

return 0;

}

Using the new system calls, this program should be able to be executed with the following:

void\* helloworld = load\_program("helloworld");

resume(create(helloworld, 4096, 20, "helloworld", 2, 0, NULL));

**Part 2: Loading a library into XINU**

For part 2 you will load a dynamic library (i.e. a file that contains multiple functions), and will create a mechanism that allows an existing XINU function to call any of the functions in the dynamic library. As an example, suppose a library file contains functions x, y, and z. Once the library has been loaded, a XINU process can:

1. Ask for the address of x and then
2. Use the address to call function x

To make the example concrete, suppose you wanted to create two utility functions in XINU and dynamically load them for programs to use:

int add1(int val)

{

return val + 1;

}

int add2(int val)

{

retun val + 2;

}

You created these functions in a file myadd.c and compiled the file to myadd. You later want to load these functions into XINU and call them. To do that you will need to provide two new system calls:

syscall load\_library(char\* path);

void\* find\_library\_function(char\* name);

The first system call, load\_library, is similar to load\_program as it loads the image from the remote file system and performs the fix up for relocatable variables. However, it does not return a function pointer. Instead it stores the pointers to all the functions within the file in a table. Another system call, find\_library\_function, can then be called to find the function pointer for a dynamically loaded function by name.

For the myadd example:

int myvalue = 2;

/\* Load the library \*/

if(load\_library("myadd") == SYSERR) {

return;

}

/\* Find the add1 function \*/

int32 (\*add1)(int32) = find\_library\_function("add1");

if((int32)add1 == SYSERR) {

return;

}

/\* Call the function \*/

add1(myvalue);

If the load\_library function cannot load the library for any reason, it returns SYSERR. Similarly if the find\_library\_function cannot successfully retrieve the function pointer for the given name, then it will return (void\*)SYSERR

For this lab:

* You will have to be able to support a maximum of 3 loaded libraries, each with a maximum of 10 functions each (30 functions total).
* A library can only be loaded once, so if a process attempts to load the same library twice then return SYSERR.
* A function loaded from a library can only exist once in XINU so if two libraries both define the same function name then return SYSERR.

**Compiling your programs and libraries**

In the directory 'programs' contains an example hello world program and a make file that you can use to create programs and libraries to run in XINU. When you run make any .c file in the programs directory will be compiled to a XINU compatible ELF file and stored in the file with the same name as the program (or library) without any extension. As mentioned earlier make sure you run the rfserver from the shell when your current working directory is the programs directory

**Setup**

In /homes/cs503/xinu there is a file called xinu-fall2014-lab6.tar.gz that contains a start to the code. Unpack:

tar zxvf /u/u3/cs503/xinu/xinu-fall2014-lab6.tar.gz

This will create a directory called xinu-fall2014-lab6.

Along with the main code for XINU, this tarball contains the following files (additional explanation of the contents of the files is in the following sections).

* system/load\_program.c - function declaration for the load\_program system call.
* system/load\_library.c - function declaration for the load\_library system call.
* system/find\_library\_function.c - function declaration for the find\_library\_function call.
* rfserver/\* - The source code for the rfserver (see above)
* programs/\* - The directory for your programs with an example hello world and Makefile
* include/prototypes.h - modified prototypes.h file which includes the load\_program, load\_library, and find\_library\_function system calls.

First things to be done when you untar the lab tarball:

* Change the macro RF\_SERVER\_PORT in file include/rfilesys.h to the port that is assigned specifically to you.
* Change the macro RF\_SERVER\_IP to the IP address of the Xinu frontend on which you will run the RFS server.
* Go into directory rfserver/ and run make. This will place the executable "rfserver" in the directory programs/ . You will have to do this step every time you change the RF\_SERVER\_IP.

How to run the RFS server

Go into directory programs/

$ ./rfserver

NOTE: Whenever you make any changes to the programs or to Xinu, you will have to restart the RFS server.

**Modification of the XINU shell**

In the shell directory you will find a modified version of the XINU shell which has been expanded to use dynamically loadable commands. As part of this lab, you will be required to implement the following two new commands for the XINU shell. Place these new commands in the 'programs' directory.

Provide the implementation for the following new shell commands as loadable programs from programs/ directory:

* semdump - this command takes no parameters and prints the contents of the semaphore table to the user. You must provide a readable table format for the output including a table header and the contents of each semaphore table entry. See semaphore.h for a description of the contents of the semaphore table. For example:
* xsh $ semdump
* Entry State Count Queue
* 0 S\_USED 3 3
* 1 S\_FREE 0 0
* ls - list the contents of a remote file system directory. This command takes zero or one parameter. If a parameter specifiying a directory is given, the contents of that directory within the remote file system are printed to the user or an error is printed indicating that the directory does not exist. For each entry within a directory that is also a directory put a '/' at the end. If no parameter is specified, the contents of the remote file system root directory are printed. For example:
* xsh $ ls mydir
* mydir1/
* mydir2/
* myfile1
* myfile2

**Examples**

A sample hello world program and myadd library are provided in the programs directory

**Additional Requirements**

* Make sure to remove all debug output from your system calls. When the TAs run your submitted code, calling a system call directly should not produce any output.
* The TAs will be replacing main.c with their own test cases after running your submitted test cases. Make sure you do not define any dependent variables in main.c. You are free to modify any other file(s) to implement the lab requirements. Just make sure that there are no dependent declarations in main.c.
* If your submitted code does not compile (either the exact submitted code or the code after the TA's replace any test case files), you will receive zero (0) points for code execution. If this happens, you will be allowed to resubmit for half credit only.
* Please run "make clean" prior to submission so that you don't submit object files
* NOTE: When you make xinu for this lab the make file will generate two files in the compile directory:
  + xinu - this is the file you will download to the xinu backend (you will NOT use xinu.xbin)
  + xinu.elf - this is the executable and linkable format version of the xinu binary. This is not the format to use when sending to the backends. It is useful for low level (assembly) debugging only.
* Some tips on writing an affective report can be found [here](https://www.cs.purdue.edu/homes/cs503/doc/REPORTS).

**Extra Credit**

**Unloading a library**

Since you can only load at most 3 libraries at any given time, there is a limit to the number of dynamically loaded functions that a program can use. For extra credit, implement a system call that allows a process to unload a library so that it can later load the same or a different library using load\_library. To do this implement the following system call:

syscall unload\_library(char\* path);

This system call unloads the library specified by the path name. Any functions that are contained in the library will be no longer usable (find\_library\_function will return SYSERR for those functions). You may assume that any processes using functions from the library to be unloaded have coordinated to ensure that no process is using the library while it is being unloaded. If the library to be unloaded has not been loaded, then return SYSERR.

In your report, consider what might happen if a process asks for a library to be unloaded while another process is using the library. What mechanism could you add to prevent such a problem? (Hint: consider a reference count).

**Loading a library archive**

In operating systems such as Unix and Linux libraries can be created and loaded from groups of files instead of just one. Statically linked libraries (also called archive libraries) can be created with the Unix 'ar' command (see man ar for more information on the ar command). When used, the compiler statically links the symbols that your program uses and makes the code part of your program. You can see this by creating a helloworld program and compiling with gcc -static. The resulting binary will be much larger than if you don't include -static. Dynamically linked libraries (also called shared object libraries) can be created using gcc with the -shared flag. When used, the compiler just checks for the existence of the symbol at compile time and the library itself is loaded by the operating system automatically at run time.

As extra credit, expand the load\_library system call to allow it to load archive libraries into the XINU address space. To do this you will need to review the standard for the '.a' archive library format created by the Unix 'ar' command. This file consists of a global header and a file header for each file within the archive.

NOTE: the GNU variant of 'ar' archives also contains a symbol table in the list of files. You will need to be able to support this.

NOTE: a single '.a' file even though it contains multiple files within it, is still considered (for the purposes of load\_library) to be a single file. You will have to be able to support the loading of up to three '.a' files that each consist of multiple object files

NOTE: For the purposes of this lab, a single '.a' file will still contain only a maximum of 10 functions within the files (as is stated in the load\_library description above. For example, a single '.a' file might contain the following 2 object files

* myadd - containing the add1 and add2 functions
* mysub - containing sub1 and sub2 which behave the same as the add1 and add2 yet subtract instead of add

This is to be considered a single library file containing 4 functions.

To create your own '.a' for testing use the following:

ar rcs LIBRARY\_NAME FILE1 FILE2 ...

Where:

* LIBRARY\_NAME represents the library file to be created
* FILE1, FILE2, etc. are the object files

A more concrete example:

ar rcs mymath.a myadd mysub

Creates a single archive library called mymath.a with the two object files myadd and mysub. This file can then be loaded into XINU using the system call:

load\_library("mymath.a");

More information on the Unix ar command and the file format can be found [here](http://en.wikipedia.org/wiki/Ar_(Unix)) and [here](http://linux.die.net/man/1/ar).

**Grading Criteria**

The lab will be graded out of 300 total points with additional points possible from the extra credit:

* 60 Points - Midway submission including your midway progress report
* 30 Points - The implementation of the new shell commands.
* 150 Points - The results from the TA's test cases.
* 60 Points - Your final report: How well your final report answers the discussion questions and your test case explanation.

**What to turn in**

**Midway Submission**

You will be required to perform a midway submission (see due date above). In your midway submission include all of your source code for XINU, any programs and/or libraries you have written, and a midway progress report in PDF format in the 'system' directory containing the following:

* What have you been able to implement and get working up until now?
* What do you plan on implementing before the final due date?
* What problems have you run into while preforming your implementation? What did you do to solve those problems?

To turn in your lab for midway submission, use the following command

turnin -c cs503 -p lab6\_part1 xinu-fall2014-lab6

assuming xinu-fall2014-lab5 is the name of the directory containing your code.

If you wish to, you can verify your submission by typing the following command:

turnin -v -c cs503 -p lab6\_part1

Do not forget the -v above, as otherwise your earlier submission will be erased (it is overwritten by a blank submission).

Note that resubmitting overwrites any earlier submission and erases any record of the date/time of any such earlier submission.

We will check that the submission timestamp is before the due date; we will not accept your submission if its timestamp is after the due date. Do NOT submit after 11:59 PM Eastern Time.

**Final Submission**

Submit using turnin command your complete source code (all of XINU) including the any files you added to complete the lab. In the 'system' directory include a PDF file with a write up discussing:

* The details behind your implementation. As part of this discussion write answers to the following questions:

To turn in your lab use the following command

turnin -c cs503 -p lab6\_part2 xinu-fall2014-lab6

assuming xinu-fall2014-lab5 is the name of the directory containing your code.

If you wish to, you can verify your submission by typing the following command:

turnin -v -c cs503 -p lab6\_part2

Do not forget the -v above, as otherwise your earlier submission will be erased (it is overwritten by a blank submission).

Note that resubmitting overwrites any earlier submission and erases any record of the date/time of any such earlier submission.

We will check that the submission timestamp is before the due date; we will not accept your submission if its timestamp is after the due date. Do NOT submit after 11:59 PM Eastern Time.

**References**

* [ELF Specification for System V](http://www.sco.com/developers/devspecs/gabi41.pdf)
* [ELF Specification System V - i386 supplement](http://www.sco.com/developers/devspecs/abi386-4.pdf)