CSC443/CSC343 Minix Example (for Project I)

Overview

This example illustrates modifying Minix by adding a system routine to the process manager that can be invoked from user processes. This routine should allow the calling user program to get the user's pid. A user process should also be able to use our system routine to get its parent's pid (ppid).

Note that Minix already has system routines that can be invoked by user processes to get the user's pid and ppid. But we are going to write a new system routine in the process manager and two new user level "library" routines: mygetpid() and mygetppid().

A user test program should be written that can call both our mygetpid and mygetppid functions and compare their return values to the POSIX getpid and getppid functions.

Details

The Minix source files are ***owned*** by user ***bin***.

So we log in as ***bin*** when we intend to modify these files as described below.

1. Find an unused integer for our new system call.

   There are two files to consult in order to find this integer.

   a. ***/usr/src/include/minix/callnr.h*** - This file contains preprocessor ***#define***s for symbolic constants (macros) for the integers used for system calls.
   b. ***/usr/src/servers/pm/table.c*** - This file contains an array, ***call_vec*** of function pointers, each of which points to a function in the process manager to execute. The index in the array corresponds to a system call integer and the function stored at that index is the process manager function that should be executed to implement the system call.

   We can identify an unused system call number in ***table.c*** by an entry of ***no_sys***. That is, if entry at index ***ix*** in the ***call_vec*** array is ***no_sys***, then ***ix*** is not currently a valid system call integer.

   Also in the file ***callnr.h*** there will not be a #define for this integer ***ix***.

   We need to check to see which number is free, but here I'll assume the integer 31 is not used.

2. Add a new entry in ***callnr.h*** for the integer we chose. We will need to select a name for the symbolic constant and use a ***#define*** to give it the integer value we chose.

   E.g., if we choose the name MYGETPID and the unused integer we found was 31, then we would add this line to ***callnr.h***:

   #define MYGETPID 31

3. We now need to change ***call_vec*** entry for the same integer. Replace the ***no_sys*** entry with a new function name.

   The naming convention for the system functions in process manager for each system call is ***do_xxxxx*** where ***xxxxx*** names the system call from the user's perspective.

   For example, we could name the funciton

   do_mygetpid

   Note that we haven't yet written this function.

4. Write the do_mygetpid function.

   Code for this function must be part of the process manager's code.

   In principle it can go in any of the existing ***.c*** files that are already used to compile and link and produce the ***pm*** file that the process manager process executes.

   For example, we can put the code in the file:

   /usr/src/servers/pm/misc.c

   The ***do_xxxxx*** functions ***must*** have the prototype of a function that has no parameters and returns an int.

   The ***do_mygetpid*** function should:

   a. Put both the pid of the calling process and its ppid into the reply message: ***mp_reply*** in the process table entry for the calling process.

How do we get the calling process's process table entry?

Answer: The message passing system will have placed a pointer to the calling user process's process table entry into the "global" variable:

struct mproc *mp;

Note that the process manager's portion of the process table is called ***mproc***, is an array of elements of type ***struct mproc***, and is defined in the file:

/usr/src/servers/pm/mproc.h

We will need to examine the entries of struct mproc to locate the pid of the calling process and its parent's pid.

b. We will also need to decide on which message format to use.

As noted in the text there are 7 different formats, but they all contain the same two first members.

The remaining members will be used for reply data and it just depends on how many and what type of values we need to send back.

In the case of ***do_mygetpid***, we need to send back two integers (i.e, two ***int***'s)

But all of the 7 message formats have two int's and we don't have to use every member. So we can use any one of the 7 message formats.

The only restriction is that the user level "library" routine that gets the reply message from the system routine needs to assume the ***same*** message format so that it can properly extract the values that the system routine as inserted into the reply message.

c. The ***do_xxxxx*** function must return an integer. Have it return the pid. This is a bit redundant since this value is also to be placed in the reply message, but it is convenient for the user level "library" routine and satisfies the requirement that the ***do_xxxxx*** function returns an integer.

5. The file ***proto.h*** in the process manager directory:

/usr/src/servers/pm/proto.h

has the prototype declarations of the functions used by the process manager.

This header file is included by ***pm.h*** which in turn is included by each .c file in the process manager directory.

This allows a function to be called from any of the .c files simply by including the ***pm.h*** header.

We need to following this general design and add a prototype declaration for wer new function in the ***proto.h*** file.

Use the _PROTOTYPE macro (discussed in class). That is, we should add this:

_PROTOTYPE(int do_mygetpid, (void));

6. Now that we have made the modifications as ***bin***, we need to rebuild Minix.

Log in as (or switch to) ***root*** and change directories to:

/usr/src/tools

We can type ***make*** to get a description of various options of how much to rebuild.

For example, one option is

make clean install

This removes executable files for process manager, file system, etc, and then rebuilds them, and finally "installs" the results in the expected directories.

7. Shutdown the system and restart it. E.g., log off other users, and as root:

shutdown -R

8. Log into a user account to create user "library" routines.

That is, log into ordinary user account (not root and not bin).

Create a user account if we haven't already.

*(We can use the **adduser** Minix command. Use **man adduser** for the manual entry.)*

Create a .c file and a .h file for ***two*** functions:

File: mygetpid.h

```
#ifndef MYGETPID_H
#define MYGETPID_H
#include <lib.h>  /* Make sure this is included! */
int mygetpid(void);
int mygetppid(void);

#endif
```

File: mygetpid.c

```
#include "mygetpid.h"
int mygetpid() {
  ...
}

int mygetppid() {
  ...
}
```

Both the functions ***mygetpid*** and ***mygetppid*** will send a message to the process manager like this:

```
message m;
int ret;
ret = _syscall(MM, MYGETPID, &m);
```

The ***_syscall*** function will return whatever our ***do_mygetpid*** returns as will as fill in the message ***m*** with the reply message.

So arrange for ***mygetpid*** to return the pid and for ***mygetppid*** to return the ppid.

9. Also as a user write a test program that calls our two functions, ***mygetpid*** and ***mygetppid*** and also calls the existing ***getpid*** and ***getppid*** and check that mygetpid and getpid agree. Similarly, check that mygetppid and getppid agree.