AIX 5L Application Programming Environment

(Course Code AU25)

Student Exercises

ERC 3.0

IBM Learning Services
Worldwide Certified Material

August 2002 Edition

Contents

Trademarks

Exercises Description

Each lab in this class is divided into sections, as described below. As you work through the labs, follow the instructions in the appropriate section.

Exercise Instructions

This section contains what it is you are to accomplish in the lab. There are no definitive details on how to perform the tasks. You are given the opportunity to challenge yourself by working through the labs using what you have learned in lecture combined with your past experience and maybe a little intuition.

Exercise Instructions With Hints

This section is an exact duplicate of the **Exercise Instructions**, except that additional specific details and/or hints are provided to step you through the lab. In fact, when we reach the later labs, actual programming solutions are provided.

A combination of using the **Exercise Instructions** along with the **Exercise Instructions with Hints** can make for a rewarding combination, providing you with challenges when you want them and helpful hints when you need them.

Optional Exercise

These labs are available for reference only. They have information that may be useful to you in your daily job. If you have the time and happen to be on the correct type of equipment to perform the lab, please feel free to do so.

Solutions

Solutions have been provided for labs where appropriate. Although many solutions are shown in the “Hints” section of your Exercise Guide, you may also want to look for the solutions online under the /home/workshop/AU25/exercise directory on your AIX system.

Exercise 1. Compiling Programs

What This Exercise is About

This lab allows the student to explore the features of the AIX compilers.

What You Should Be Able to Do

At the end of the lab, students should be able to:

- Set up a Concurrent Nodelock license for the compiler
- Use the **cc** command and many of its options to compile multifunction and multimodule programs
- Generate and examine various compiler listings
- Modify the /etc/vac.cfg file to alter the C compiler's default behavior
- Compile and execute programs in the 64 bit mode

Introduction

The exercise deals with compiling a source code and explores various features of the compiler. Licensing of the compiler is also dealt with in detail.

Required Materials

- A pSeries or RS/6000 system with AIX 5L Version 5.1.

Exercise Instructions

__ 1. If not already created, create an src directory under your $HOME directory. Add /usr/vac/bin directory into your PATH variable.

__ 2. Create the following three modules in your src directory:

1) mydefs.h

```
/*   mydefs.h   Typical header file     */
int factor;
int incnum;
int sad;
int myfunc(int, int, int);
```

2) mod1.c

```
/*   mod1.c   Contains main processing  */
#include <stdio.h>
#include "mydefs.h"    /* assumes local  */
#define MAX 15

main()
{
   int a;
   factor=5;
   incnum=3;
   #ifdef DEBUG
      printf("This is a test run.\n\n");
   #endif
   for (a=1; a<=MAX; a+=incnum)
       printf("a is %d, result is %d.\n",a,a*factor);
   printf("\nPassing to myfunc.\n");
   myfunc(a,factor,incnum);
   printf("\nEnding program.\n");
}
```

3) mod2.c

```
/*  mod2.c  Contains myfunc() code   */
#define MIN 0
myfunc(int x, int newfactor, int decnum)
{
   printf("Now in myfunc in mod2.c\n");
   for (x;x>=MIN;x-=decnum)
       printf("x is %d, result is %d.\n",x,x*newfactor);
```

```
    printf("Returning to main now.\n");
}
```

__ 3. Answer the following questions about these source files:

Q1. What is the difference between the two include directives in mod1.c?

____

Q2. What does the ifdef statement do?

____

Q3. How does the program get from mod1.c to mod2.c?

____

Q4. Why are factor and incnum defined in a header file?

____

__ 4. Examine the */etc/vac.cfg* file and answer the following questions:

Q1. What is signified by the list of libraries specified with the *library* attribute?

____

Q2. What is signified by the list of command line options specified with the *options* attribute?

____

__ 5. Compile the two source code files into an executable named myprog. Do it as a simple compile, without optimizing or any other special options. Run the myprog program to see the results. Did you see the message "*This is a test run.*" in the program's output? ________________. If you see a licensing error in this step, continue until Step 11 and repeat this step again. If there are no licensing errors or if the license is already setup, proceed to Step 12

__ 6. Log in as root using the **su** command.

__ 7. Configure your system as a Concurrent Nodelock server.

__ 8. Start the Concurrent Nodelock server daemon.

__ 9. Enroll a product license in your Concurrent Nodelock Server for 10 licenses.

Note: This command can take about two minutes. Depending on the system, this command may take up to 15 minutes or more to complete execution.

__ 10. Confirm that the license is available on your machine.

__ 11. Logout of root session. If you had a license problem in Step 5, go back to Step 5 and then skip to Step 12.

__ 12. Recompile the source code files to have the resulting executable display the message "*This is a test run.*".

__ 13. Try to compile just mod1.c or mod2.c without the other. Does it work? __________

__ 14. Compile again, but this time save the results of the C preprocessor to the .i file. Examine the .i files. Near the bottom of the mod1.i file, note what happens to the printf statement when you compile this source code with and without using the -DDEBUG compiler option on the compiler command line.

__ 15. This time, compile each module separately, without linking. Then link the resulting .o files into the executable.

__ 16. Compile these two files again, this time generating sample listings using -qattr, -qlist, -qlistopt, -qsource, and -qxref. Examine the listings as you go.

__ 17. Look up **cxref** in the AIX online documentation. Use it to get a more informative cross-reference of your source files. Examine the output file generated by the **cxref** command.

__ 18. Compile the two source code files into a 64-bit executable named myprog64.

__ 19. Try executing the executable myprog64. Does it execute? _______________

__ 20. If myprog64 does not execute, explain why it does not execute.

____

Exercise Instructions With Hints

__ 1. If not already created, create an src directory under your $HOME directory. Add /usr/vac/bin directory into your PATH variable.

```
$ cd
$ mkdir src
$ cd src
$ export PATH=$PATH:/usr/vac/bin
```

__ 2. Create the following three modules in your src directory:

1) mydefs.h

```
/*   mydefs.h   Typical header file     */
int factor;
int incnum;
int sad;
int myfunc(int, int, int);
```

2) mod1.c

```
/*   mod1.c   Contains main processing   */
#include <stdio.h>
#include "mydefs.h"    /* assumes local   */
#define MAX 15

main()
{
   int a;
   factor=5;
   incnum=3;
   #ifdef DEBUG
       printf("This is a test run.\n\n");
   #endif
   for (a=1; a<=MAX; a+=incnum)
       printf("a is %d, result is %d.\n",a,a*factor);
   printf("\nPassing to myfunc.\n");
   myfunc(a,factor,incnum);
   printf("\nEnding program.\n");
}
```

3) mod2.c

```
/*  mod2.c  Contains myfunc() code  */
```

```
#define MIN 0
myfunc(int x, int newfactor, int decnum)
{
   printf("Now in myfunc in mod2.c\n");
   for (x;x>=MIN;x-=decnum)
      printf("x is %d, result is %d.\n",x,x*newfactor);
   printf("Returning to main now.\n");
}
```

__ 3. Answer the following questions about these source files:

Q1. What is the difference between the two include directives in mod1.c?

The <stdio.h> include searches for the stdio.h file in the standard directory /usr/include. The "mydefs.h" include searches for the mydefs.h header file in the current directory.

Q2. What does the ifdef statement do?

ifdef tells the preprocessor to include the printf statement only if the constant DEBUG is defined.

Q3. How does the program get from mod1.c to mod2.c?

mod1.c calls the myfunc function.

Q4. Why are factor and incnum defined in a header file?

So factor and incnum may be used by multiple programs.

__ 4. Examine the */etc/vac.cfg* file and answer the following questions:

Q1. What is signified by the list of libraries specified with the *library* attribute?

The *library* attribute tells the compiler to include the specified libraries by default.

Q2. What is signified by the list of command line options specified with the *options* attribute?

The *options* attribute tells the compiler to use the specified compiler options by default.

__ 5. Compile the two source code files into an executable named myprog. Do it as a simple compile, without optimizing or any other special options. Run the myprog program to see the results. Did you see the message "*This is a test run.*" in the program's output? ________________. If you see a licensing error in this step, continue until Step 11 and repeat this step again. If there are no licensing errors or if the license is already setup, proceed to Step 12.

```
$ cc -o myprog mod1.c mod2.c
$ myprog
```

"This is a test run" should not appear

__ 6. Log in as root using the **su** command.

```
$ su - root
password:
#
```

__ 7. Configure your system as a Concurrent Nodelock server.

.

```
# /usr/opt/ifor/ls/bin/i4cfg -a n -S a
```

__ 8. Start the Concurrent Nodelock server daemon.

```
# /usr/opt/ifor/ls/bin/i4cfg -start
```

__ 9. Enroll a product license in your Concurrent Nodelock Server for 10 licenses.

```
# /usr/opt/ifor/ls/bin/i4blt -a -f /usr/vac/cforaix_cn.lic -R u -T 10
```

Note: This command can take about two minutes. Depending on the system, this command may take up to 15 minutes or more to complete execution.

__ 10. Confirm that the license is available on your machine.

```
# /usr/opt/ifor/ls/bin/i4blt -s -l cn
```

__ 11. Logout of root session. If you had a license problem in Step 5, go back to Step 5 and then skip to Step 12.

```
# exit
$
```

__ 12. Recompile the source code files to have the resulting executable display the message "*This is a test run.*".

```
$ cc -DDEBUG -o myprog mod1.c mod2.c
$ myprog
```

`"This is a test run"` should appear.

__ 13. Try to compile just mod1.c or mod2.c without the other. Does it work? ___________

```
$ cc mod1.c
$ cc mod2.c
```

Neither module can be compiled alone, because mod1.c does not resolve myfunc() and mod2.c does not contain a main() routine.

__ 14. Compile again, but this time save the results of the C preprocessor to the .i file. Examine the .i files. Near the bottom of the mod1.i file, note what happens to the printf statement when you compile this source code with and without using the -DDEBUG compiler option on the compiler command line.

```
$ cc -P mod1.c mod2.c
$ vi mod1.i mod2.i
$ cc -DDEBUG -P mod1.c mod2.c
$ vi mod1.i mod2.i
```

__ 15. This time, compile each module separately, without linking. Then link the resulting .o files into the executable.

```
$ cc -c mod1.c
$ cc -c mod2.c
$ cc -o myprog mod1.o mod2.o
$ ./myprog
```

__ 16. Compile these two files again, this time generating sample listings using -qattr, -qlist, -qlistopt, -qsource, and -qxref. Examine the listings as you go.

```
$ cc -qattr -o myprog mod1.c mod2.c
$ pg mod1.lst mod2.lst
$ cc -qlist -o myprog mod1.c mod2.c
$ pg mod1.lst mod2.lst
$ cc -qlistopt -o myprog mod1.c mod2.c
$ pg mod1.lst mod2.lst
$ cc -qsource -o myprog mod1.c mod2.c
$ pg mod1.lst mod2.lst
$ cc -qxref -o myprog mod1.c mod2.c
$ pg mod1.lst mod2.lst
```

__ 17. Look up **cxref** in the AIX online documentation. Use it to get a more informative cross-reference of your source files. Examine the output file generated by the **cxref** command.

On the AIX documentation search screen, type in **cxref** and hit enter:

```
$ cxref -c mod1.c mod2.c > xref.out
$ vi xref.out
```

__ 18. Compile the two source code files into a 64-bit executable named myprog64.

```
$ cc -q64 -o myprog64 mod1.c mod2.c
```

__ 19. Try executing the executable myprog64. Does it execute? ________________

- If your system hardware supports 64-bit mode, the program myprog64 would execute without any problem.
- If your system hardware does not support 64-bit mode, the program myprog64 cannot execute.

__ 20. f myprog64 does not execute, explain why it does not execute.

If myprog64 displays the following error:

exec(): 0509-036 Cannot load program ./myprog64 because of the following errors:

0509-032 Cannot run a 64-bit program on a 32-bit machine.

Because you are trying to run this 64-bit executable on a 32-bit machine.

END OF LAB

Exercise 2. The AIX Debuggers

What This Exercise is About

The dbx and idebug debuggers allow the user to work at the source code level while looking for errors in an executable program. Typical steps of debugging include setting breakpoints, stepping through the program, and examining and changing variable values. These procedures will be performed in this lab.

What You Should Be Able to Do

At the end of the lab, students should be able to:

- Invoke an AIX debugger on a program
- List portions of the source code
- Set breakpoints within the program
- Use the step-through approach to debug a program
- View and change the values of variables
- Locate and correct simple C program bugs

Introduction

This lab uses a program that contains three logical errors (not detected by the compiler). You will copy the source code for the program from a common directory into your own working directory. You will compile the program, then run it. The program should fail, producing a core dump. You will use an AIX debugger to locate the bug which caused the program to fail. After fixing the bug and recompiling, the program will run, but not generate the desired results. Your job is to locate the other two logical errors, fix the code and recompile, then test run the program to get the desired results.

The program example is a very simple C program. It is more important for you to practice using one of the AIX debuggers than to struggle with complex C programming examples. If you are a strong C programmer, you may be tempted to debug the program just from examining the code. We encourage you to follow the steps outlined in the lab and to use the facilities of the debugger to locate the bugs.

Desired results from buggy.c program

```
Starting...                Now on to Part 2 ...       Now Part 3 ...
The value of y is 1.       The value of x is 10.      The value of x is 0.
x divided by y is 10.      y divided by x is 1.       The value of x is 1.
The value of y is 2.       The value of x is 9.       The value of x is 2.
x divided by y is 5.       y divided by x is 1.       The value of x is 3.
The value of y is 3.       The value of x is 8.       The value of x is 4.
x divided by y is 3.       y divided by x is 1.       The value of x is 5.
The value of y is 4.       The value of x is 7.       The value of x is 6.
x divided by y is 2.       y divided by x is 1.       The value of x is 7.
The value of y is 5.       The value of x is 6.       The value of x is 8.
x divided by y is 2.       y divided by x is 1.       The value of x is 9.
The value of y is 6.       The value of x is 5.       The value of x is 10.
x divided by y is 1.       y divided by x is 2.
The value of y is 7.       The value of x is 4.
x divided by y is 1.       y divided by x is 2.
The value of y is 8.       The value of x is 3.
x divided by y is 1.       y divided by x is 3.
The value of y is 9.       The value of x is 2.
x divided by y is 1.       y divided by x is 5.
The value of y is 10.      The value of x is 1.
x divided by y is 1.       y divided by x is 11.
```

Required Materials

- One pSeries (or RS/6000) system with AIX 5L
- C for AIX compiler
- Filesets corresponding to idebug debugger, included as part of the compiler CD-ROM. The following are the filesets to be installed:
  - idebug.client.gui - Debugger Graphical User
  - idebug.engine.compiled - Debugger Engine for Compiled
  - idebug.engine.interpreted - Debugger Engine for Interpreted
  - idebug.help.en_US - Debugger Help--U.S. English
  - idebug.msg.en_US.engine - Debugger Engine Messages--U.S.
  - idebug.rte.hpj - High-Performance Java Runtime
  - idebug.client.gui - Debugger Graphical User Interface

Exercise Instructions

This lab comes in a dbx version and a idebug version. Complete one of the two versions. If you have sufficient time (and appropriate hardware), you may want to complete the other version.

Note that depending on hardware availability, you may not be able to do the idebug version, as it requires a GUI environment (that is, some flavor of X-windows).

dbx Version

__ 1. Create a new directory named dbxlab under your home directory. Copy the file buggy.c from the directory specified by your instructor to the dbxlab you just created.

__ 2. Compile the buggy.c program using the appropriate option flags for debugging and the -qcheck option to catch additional errors. Name the executable program buggy.

__ 3. Run the buggy program. It should fail with a core dump. List the contents of your current directory (dbxlab) and note the existence of a core file. Do not attempt to cat the core file.

__ 4. Invoke the dbx debugger on the executable program buggy. Remember that core is the expected core dump file name, so you need not supply it as an argument to the **dbx** command. The (dbx) prompt should appear along with the line of source code that caused the core dump. Note the line number.

__ 5. Issue the **help dbx** subcommand and examine the list of available commands and topics.

__ 6. List the source code lines. Remember that dbx maintains a line pointer that can be moved simply by listing a specified set of lines. Refer to the dbx help list or the lecture notes for more information on listing lines.

__ 7. While listing lines, locate a good place near the beginning of the source code file to place a breakpoint. Remember that breakpoints must be set at executable lines of code. Place a breakpoint at the line with the first printf statement.

__ 8. Run the program from within dbx. When the first breakpoint is reached, examine the values of the x and y variables. Step through the program, examining the variables at each line. Can you locate the point and reason for the core dump?

__ 9. If you have found the first bug, you will notice that it is caused by a variable with an undesirable value. Now run the program once again from within dbx. Try changing the value of this variable to 1 from within dbx, but before executing the line that caused the error. (Remember that this does not change the source code, only the value of a variable during this debugging session changes.) With the new value assigned to the problem variable, continue. Did this fix the first problem? From within dbx, edit the buggy.c file to fix the problem with the offending variable.

HINT: The offending variable lacks an initial value assignment. Assign this variable a starting value of 1. Recompile from within dbx. You need to be careful to name the output file to another name (not buggy). Since you are compiling from within dbx, and the executable is in use, the compiler will give an error if the output is given the same name. Exit dbx and then run the program. Another option is to exit from dbx first and then recompile with the same name for the output file. This time the program completes, but not with the desired results.

__ 10. Invoke dbx on the buggy program once again. Using breakpoints and the step technique, try to locate the bug that is causing the second part of the program to not print the desired results. Again, look carefully at variable values. Once you have located the problem, edit the source code to fix the bug, exit dbx, and recompile and test the program.

__ 11. There is still something wrong with Part 3 of this program. The error here has more to do with C syntax than anything else. If you are not a C programmer, use dbx to list other lines of the program and compare the syntax of the for statement in Part 2 with the for statement in Part 3. Do you notice anything unusual? Edit the source code file to fix the bug. Recompile and test the program. The results should now match the desired results printed earlier in this lab.

__ 12. **Extra Credit for the C Programmer**

You may have noticed that the division done within this program is in integer values, thus truncating and not giving true results. Modify this program to print the results of the divisions as real (floating point) numbers.

idebug Version

This version of the lab must be done from within the AIX GUI environment. idebug does not have an option to display global variables. Hence, use the buggy.c program with the x and y variables defined within the main module.

__ 1. Create a new directory named idebuglab under your home directory. Copy the file buggy.c from the directory specified by your instructor to the idebuglab directory you just created.

__ 2. Compile the buggy.c program using the appropriate option flags for debugging and the -qcheck option to catch additional errors. Name the executable program buggy. Run the executable. It must result in a coredump.

__ 3. Start idebug with the buggy program.

__ 4. Continue the program. It should be stopped by a SIGTRAP signal (a window pops up, displaying the error message). Note which line the program stopped on (see the right-pointing arrow in the source pane of the idebug window).

__ 5. Set breakpoints at the first and last printf statements in the main module. Setting breakpoints at the last executing statement is important because once the execution is through, the output window closes. We will not be able to see the output of the program otherwise.

__ 6. Examine the x and y variables. Can you identify why the program is stopping?

__ 7. Once you have found the first bug, you will discover that it is caused by a variable with an inappropriate value. Now, reload the program, and continue so that the program stops at the breakpoint. Change the variable with the inappropriate value to 1 from within idebug (remember that this does not change the source code, only the variable within this debugging session changes). With the new value assigned to the problem variable, continue. Did this fix the first problem? Exit from idebug and fix the error in the program (that is, make sure that the variable has an appropriate value before executing the statement which was failing). Recompile the program and run it again outside of the debugger. The program should complete but not with the desired results.

__ 8. Invoke idebug on the buggy program again. Using breakpoints and idebug's **step** command, try to locate the bug that is causing the second part of the program to not print the correct results (note that the output of the program will appear in the output window). Again, look carefully at variable values. Once you have located the problem, exit idebug and edit the source code to fix the bug. Recompile and run the program again.

__ 9. There is still something wrong with Part 3 of this program. The error here has more to do with C syntax than anything else. If you are not a C programmer, use idebug to look at other lines of the program and compare the syntax of the for statement in Part 2 with the for statement in Part 3. Once you discover the problem, exit from idebug and fix the problem in the source code. Recompile and test the program. The results should now match the desired results printed earlier in this lab.

__ 10. **Extra Credit for the C Programmer**

You may have noticed that the division done within this program is in integer values, thus truncating and not giving true results. Modify this program to print the results of the divisions as real (floating point) numbers.

END OF LAB

Exercise Instructions With Hints

This lab comes in a dbx version and a idebug version. Complete one of the two versions. If you have sufficient time (and appropriate hardware), you may want to complete the other version.

Note that depending on hardware availability, you may not be able to do the idebug version, as it requires a GUI environment (that is, some flavor of X-windows).

dbx Version

__ 1. Create a new directory named dbxlab under your home directory. Copy the file buggy.c from the directory specified by your instructor to the dbxlab you just created.

```
$ mkdir dbxlab
$ cd dbxlab
$ cp (directory supplied by instructor)/buggy.c buggy.c
```

__ 2. Compile the buggy.c program using the appropriate option flags for debugging and the -qcheck option to catch additional errors. Name the executable program buggy.

```
$ cc  -g -qcheck buggy.c -o buggy
```

__ 3. Run the buggy program. It should fail with a core dump. List the contents of your current directory (dbxlab) and note the existence of a core file. Do not attempt to cat the core file.

```
$ buggy
$ ls -l
```

__ 4. Invoke the dbx debugger on the executable program buggy. Remember that core is the expected core dump file name, so you need not supply it as an argument to the **dbx** command. The (dbx) prompt should appear along with the line of source code that caused the core dump. Note the line number.

```
$ dbx buggy
```

__ 5. Issue the **help dbx** subcommand and examine the list of available commands and topics.

```
    (dbx) help
```

__ 6. List the source code lines. Remember that dbx maintains a line pointer that can be moved simply by listing a specified set of lines. Refer to the dbx help list or the lecture notes for more information on listing lines.

```
    (dbx)  list 1,10
    (dbx)  list
    (dbx)  list 5
```

__ 7. While listing lines, locate a good place near the beginning of the source code file to place a breakpoint. Remember that breakpoints must be set at executable lines of code. Place a breakpoint at the line with the first printf statement.

```
    (dbx)  stop at  (number of line with first printf)
```

__ 8. Run the program from within dbx. When the first breakpoint is reached, examine the values of the x and y variables. Step through the program, examining the variables at each line. Can you locate the point and reason for the core dump?

```
    (dbx)  run
    (dbx)  print x
    (dbx)  print y
    (dbx)  step

   The problem is caused by the fact that the y variable is equal
   to 0 (zero), thus attempting divide-by-zero.
```

__ 9. If you have found the first bug, you will notice that it is caused by a variable with an undesirable value. Now run the program once again from within dbx. Try changing the value of this variable to 1 from within dbx, but before executing the line that caused the error. (Remember that this does not change the source code, only the value of a variable during this debugging session changes.) With the new value assigned to the problem variable, continue. Did this fix the first problem? From within dbx, edit the buggy.c file to fix the problem with the offending variable.

HINT: The offending variable lacks an initial value assignment. Assign this variable a starting value of 1. Recompile from within dbx. You need to be careful to name the output file to another name (not buggy). Since you are compiling from within dbx, and the executable is in use, the compiler will give an error if the output is given the same name. Exit dbx and then run the program. Another option is to exit from dbx first and then recompile with the same name for the output file. This time the program completes, but not with the desired results.

```
 (dbx)  run
 (dbx)  assign y=1
 (dbx)  cont
 (dbx)  edit

int y;

...to...

int y=1;
:wq
 (dbx)  quit
$ cc -g -qcheck buggy.c -o buggy
$ buggy
```

__ 10. Invoke dbx on the buggy program once again. Using breakpoints and the step technique, try to locate the bug that is causing the second part of the program to not print the desired results. Again, look carefully at variable values. Once you have located the problem, edit the source code to fix the bug, exit dbx, and recompile and test the program.

```
$ dbx buggy
(dbx) goto 15
(dbx) next
(dbx) next

While following the execution of buggy, dbx jumps
from Part 2 to Part 3, skipping the printf
statements. The problem is the line:

for ( x ; x<=1 ; x-- )

Since the value of x starts at 10, it fails the x<=1 test the
first time through the loop; thus, the loop ends. The line
should read:

for ( x ; x>=1 ; x-- )

The printf statement is also incorrect. It prints the value of
x, but is not passed the argument x. The statement has to be
changed from:

printf("The value of x is %d.\n");

...to...

printf("The value of x is %d.\n",x);

To fix the problem, do the following:

(dbx) edit

Change the line to what is shown above:

:wq
(dbx) quit
$ cc -g buggy.c -o buggy
$ buggy
```

__ 11. There is still something wrong with Part 3 of this program. The error here has more to do with C syntax than anything else. If you are not a C programmer, use dbx to list other lines of the program and compare the syntax of the for statement in Part 2 with the for statement in Part 3. Do you notice anything unusual? Edit the source code file to fix the bug. Recompile and test the program. The results should now match the desired results printed earlier in this lab.

```
In Part 3, the for statement should not end with a semicolon.
This causes the printf statement to only print after the "null
loop" ends. The printf statement is also incorrect. To fix the
problem, simply change the lines to:

for ( x ; x<=10; x++ )
        printf("The value of x is %d.\n",x);
```

__ 12. **Extra Credit for the C Programmer**

You may have noticed that the division done within this program is in integer values, thus truncating and not giving true results. Modify this program to print the results of the divisions as real (floating point) numbers.

```
Change the %d formats to %f formats in the printf statements
involving division. Cast the first variable of the division as
float. To fix the problem, simply change the lines to:

      printf("x divided by y is %f.\n",(float)x/y);
      printf("y divided by x is %f.\n",(float)y/x);
```

idebug Version

This version of the lab must be done from within the AIX GUI environment. idebug does not have an option to display global variables. Hence, use the buggy.c program with the x and y variables defined within the main module.

__ 1. Create a new directory named idebuglab under your home directory. Copy the file buggy.c from the directory specified by your instructor to the idebuglab directory you just created.

```
$ mkdir idebuglab
$ cd idebuglab
$ cp (directory supplied by instructor)/buggy.c buggy.c
```

__ 2. Compile the buggy.c program using the appropriate option flags for debugging and the -qcheck option to catch additional errors. Name the executable program buggy. Run the executable. It must result in a coredump.

```
$ cc -g -qcheck buggy.c -o buggy
$ ./buggy
```

__ 3. Start idebug with the buggy program.

```
$ idebug ./buggy &
```

__ 4. Continue the program. It should be stopped by a SIGTRAP signal (a window pops up, displaying the error message). Note which line the program stopped on (see the right-pointing arrow in the source pane of the idebug window).

```
Press F5 to continue the execution. The program will start and
then almost immediately stop. You will see a window pop up
displaying an error message indicating that the program has
stopped as a result of a SIGTRAP signal. Click OK to close the
pop up window. There will be a right-pointing arrow in the
source panel sub-window that occupies the right half of the
main idebug window. This arrow should be pointing at the line:

         printf("x divided by y is %d.\n",x/y);
```

__ 5. Set breakpoints at the first and last printf statements in the main module. Setting breakpoints at the last executing statement is important because once the execution is through, the output window closes. We will not be able to see the output of the program otherwise.

```
    Right-click on the line where breakpoint is to be set and
    select Set Breakpoint or press F9 after clicking on the line
    to set the breakpoint.
```

__ 6. Examine the x and y variables. Can you identify why the program is stopping?

```
Click on the Locals tab in the Value panes sub-window, which
will reveal that y is +0 and x is +10.
Dividing x by y is illegal because y is zero.
```

__ 7. Once you have found the first bug, you will discover that it is caused by a variable with an inappropriate value. Now, reload the program, and continue so that the program stops at the breakpoint. Change the variable with the inappropriate value to 1 from within idebug (remember that this does not change the source code, only the variable within this debugging session changes). With the new value assigned to the problem variable, continue. Did this fix the first problem? Exit from idebug and fix the error in the program (that is, make sure that the variable has an appropriate value before executing the statement which was failing). Recompile the program and run it again outside of the debugger. The program should complete but not with the desired results.

We want to change the value of y to something else. Anything other than 0 will do, so we will change it to 1. The program is currently stopped, ready to proceed with the division. Because it has already loaded y into a register, it does not do us any good to change the variable now. We will reload the program, change the variable and then allow it to continue.

Exit the debugger and invoke the program. Continue the program (press F5) so that it stops at the first breakpoint. Note that the right-pointing arrow is now on the first printf call in the program. Click on Locals tab in the Value panes. This will display the value of y. Right-click on y and select Edit from the menu. This pops up a text box. Replace the existing value of +0 and type in a new value of 1 (or any other non-zero value). Press the Enter key to update the variable's value. The Locals tab will now display the new value for y.

Press F5 to continue the program. The program will now run to completion. But the output (it will appear in the window that pops up, along with the debugger window) is incorrect. Exit from idebug.

```
$ vi buggy.c /* Fix the bug by assigning y = 1 */
$ cc -g -qcheck buggy.c -o buggy
$ ./buggy
```

__ 8. Invoke idebug on the buggy program again. Using breakpoints and idebug's **step** command, try to locate the bug that is causing the second part of the program to not print the correct results (note that the output of the program will appear in the output window). Again, look carefully at variable values. Once you have located the problem, exit idebug and edit the source code to fix the bug. Recompile and run the program again.

Start idebug again. Right click on the line in the ./buggy.c source window that reads:

```
printf("Now on to Part 2...\n");
```

Select Set Breakpoint from the menu. A red circle appears at the left corner indicating that a breakpoint is set. Press F5 to continue.

The program will stop on the printf statement that you set a breakpoint on. Press F8 and the arrow will advance to the for statement.

Press F8 again and the arrow will advance past the entire for loop to the printf that starts the last part of the program!

Clearly the problem is related to the for statement. A quick look at the local variables (click Locals tab in the Value panes of the main idebug window) reveals that x is +10. The test in the for loop is x<=1. Since x is +10, this is false and the loop is not executed. In order for x to count down to 1, like the desired results output says it should, this test should be x>=1.

Also, the printf statement following the for statement has an error. It is supposed to print the value of x, but an argument is not passed to it. The statement must be:

```
printf("The value of x is %d.\n",x);
```

Exit from idebug and fix the source code. Recompile the program and try it again. The output should now be correct for Part 2.

```
$ vi buggy.c /* Fix the bug as described above */
$ cc -g -qcheck buggy.c -o buggy
$ ./buggy
```

__ 9. There is still something wrong with Part 3 of this program. The error here has more to do with C syntax than anything else. If you are not a C programmer, use idebug to look at other lines of the program and compare the syntax of the for statement in Part 2 with the for statement in Part 3. Once you discover the problem, exit from idebug and fix the problem in the source code. Recompile and test the program. The results should now match the desired results printed earlier in this lab.

Invoke idebug on the buggy program. Set a breakpoint at the line that reads:

```
printf(“Starting Part 3...\n”);
```

Continue execution(press F5)until the breakpoint we just set is reached. Press F8 several times. You will notice that the execution does not enter into the loop. Instead, the program terminates.
There are actually two problems to fix here:

- The for statement should not end with a semi-colon.
- The printf statement should not have 5D in it. This should be %d instead.

Changing these two lines from:

```
for(x;x<=10;x++);
printf("The value of x is 5D.\n",x);
```

to

```
for(x;x<=10;x++)
printf("The value of x is %d.\n",x);
```

will correct both problems.

__ 10. **Extra Credit for the C Programmer**

You may have noticed that the division done within this program is in integer values, thus truncating and not giving true results. Modify this program to print the results of the divisions as real (floating point) numbers.

Change the %d formats to %f formats in the printf statements involving division. Cast the first variable of the division as float. Here are the changed lines:

```
printf("x divided by y is %f.\n",(float)x/y);
printf("y divided by x is %f.\n",(float)y/x);
```

END OF LAB

Exercise 3. The make and SCCS Facility

What This Exercise is About

This lab allows the student to explore the features of the AIX **make** command and the SCCS utility.

What You Should Be Able to Do

At the end of the lab, students should be able to:

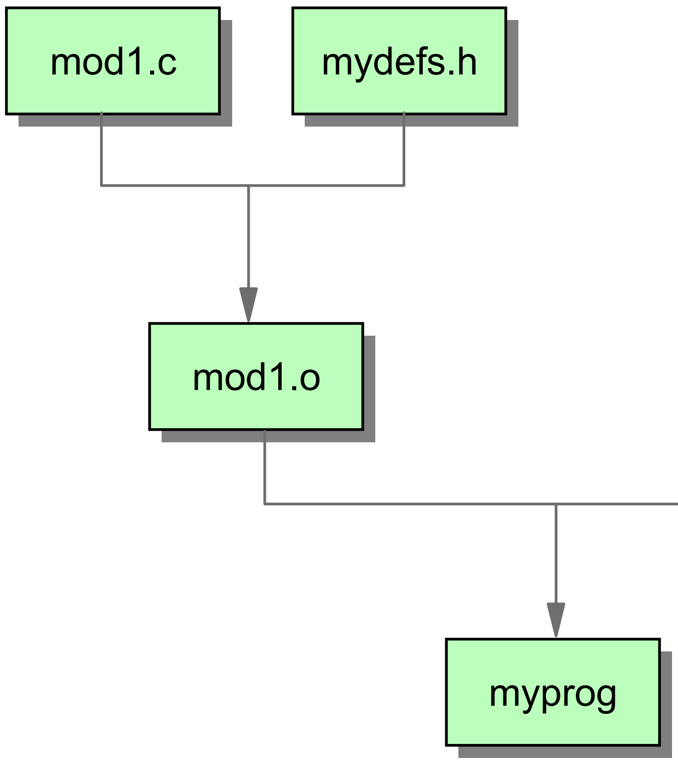
- Establish dependencies and actions in a makefile.
- Execute the **make** command with a variety of options.
- Use macros in makefiles.
- Use SCCS.

Required Materials

- A pSeries or RS/6000 system with AIX 5L Version 5.1.

Exercise Instructions

In Step 2 of Exercise 1 - Compiling Programs, you would have worked with three source files that were compiled and executed. The source files are mod1.c, mod2.c, and mydefs.h. The diagram below defines how the modules are compiled into a program.



__ 1. Compile these files into an executable called myprog.

__ 2. Build a makefile to manage the dependencies for the myprog program. Try executing it. (Remember to use tabs on the action lines.)

__ 3. Edit mod1.c. Change the last print statement to print something else. Save and exit. Based on the dependencies, what has to happen now to make myprog reflect the change?

____

Run **make** to see if you were right. Execute myprog to make sure that your changes are reflected.

__ 4. Make a copy of makefile and call it mymake. Add several comments to mymake. Include some additional commands, such as **echo** statements. Include as an action line under the myprog dependency line **rm *bogus***, where bogus is a file that should not exist in the current directory. Make a change to one of the source files and run **make** against the mymake file. Does make fail? If so, why?

____

Rerun **make** so that the error will not cause **make** to terminate. Next, edit your makefile and add a fake dependency to suppress the error messages and test it.

__ 5. Make a copy of mymake and call it mymakeopt. Change mymakeopt so that the **echo** actions are not output twice. Then, remove the **rm bogus** line from mymakeopt. Now, create a macro called OPTLIST. This is going to provide options to the compile line. Set OPTLIST to -O for optimization. Include the macro in the action lines associated with the mod1.o and mod2.o. What happens now when you execute **make** against mymakeopt?

____

__ 6. The reason for using macros is to make it easier for **make** actions to be customized on the command line. Re-execute **make**, overriding the OPTLIST macro on the command line with the -g option. Put the OPTLIST phrase in double quotes.

__ 7. Make a copy of mymakeopt and call it myclean. Edit myclean and add a cleanup pseudo target to remove the .o files. Execute **make** to do just the cleanup step.

__ 8. Bring the file mod1.c under the custody of SCCS.

__ 9. Extract the file mod1.c for editing.

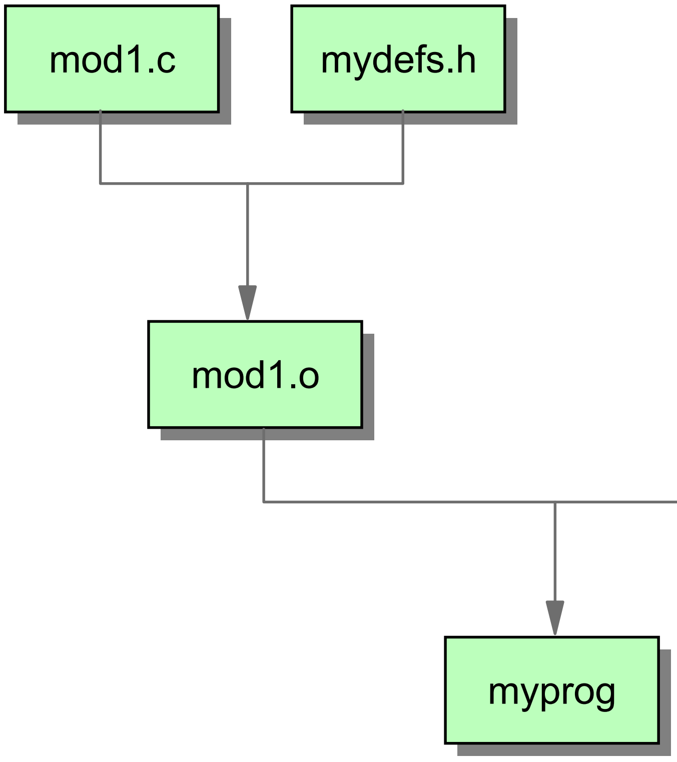
__ 10. Place the ID keywords in the first line to show the module name and the SID of the mod1.c file. Check this modified file.

__ 11. Extract a read-only copy of mod1.c to see how the ID keywords are expanded.

END OF LAB

Exercise Instructions With Hints

In Step 2 of Exercise 1 - Compiling Programs, you would have worked with three source files that were compiled and executed. The source files are mod1.c, mod2.c, and mydefs.h. The diagram below defines how the modules are compiled into a program.



__ 1. Compile these files into an executable called myprog.

```
$ cc  -o myprog mod1.c mod2.c
```

__ 2. Build a makefile to manage the dependencies for the myprog program. Try executing it. (Remember to use tabs on the action lines.)

```
$ vi makefile
myprog: mod1.o mod2.o
      cc -o myprog mod1.o mod2.o
mod1.o: mod1.c mydefs.h
      cc -c mod1.c
mod2.o: mod2.c
      cc -c mod2.c
$ make
        cc -c mod1.c
        cc -c mod2.c
        cc -o myprog mod1.o mod2.o
$ make
Target "myprog" is up to date.

Make sure that this works.
```

__ 3. Edit mod1.c. Change the last print statement to print something else. Save and exit. Based on the dependencies, what has to happen now to make myprog reflect the change?

—mod1.o and myprog would be built again.

Run **make** to see if you were right. Execute myprog to make sure that your changes are reflected.

```
$ vi mod1.c   # Change last printf string to "...something
else..."
$ make        # mod1.c needs recompile and relink
$ myprog      # Did the change take effect?

You should be able to see your changes when myprog is executed.
```

__ 4. Make a copy of makefile and call it mymake. Add several comments to mymake. Include some additional commands, such as **echo** statements. Include as an action line under the myprog dependency line **rm *bogus***, where bogus is a file that should not exist in the current directory. Make a change to one of the source files and run **make** against the mymake file. Does make fail? If so, why?

```
$ cp makefile mymake
$ vi mymake
# The mymake file builds the myprog program
myprog: mod1.o mod2.o
      rm bogus
      cc -o myprog mod1.o mod2.o
      echo "Today is `date`"
mod1.o: mod1.c mydefs.h   # This is a comment aaa
      cc -c mod1.c
      echo "I sure wish it was lunch time"
mod2.o: mod2.c
      cc -c mod2.c
$ vi mod1.c         # Make some change to the source
$ make -f mymake    # Did it succeed?

Make will fail with a error message as the rm command returns a
non-zero value.

$ make -i -f mymake # Ignore errors and continue.

Make will print the error messages but will continue to
success.

$ vi mymake
# Add the line .IGNORE: in the makefile
.IGNORE:

...

$ touch mod1.c
$ make -f mymake    # Did it succeed?

Make will print the error messages but will continue to
success.
```

Rerun **make** so that the error will not cause **make** to terminate. Next, edit your makefile and add a fake dependency to suppress the error messages and test it.

__ 5. Make a copy of **mymake** and call it **mymakeopt**. Change **mymakeopt** so that the echo actions are not output twice. Then, remove the **rm bogus** line from **mymakeopt**. Now, create a macro called **OPTLIST**. This is going to provide options to the compile line. Set **OPTLIST** to **-O** for optimization. Include the macro in the action lines associated with the **mod1.o** and **mod2.o**. What happens now when you execute **make** against **mymakeopt**?

```
$ vi mymakeopt
# The mymakeopt file builds the myprog program
OPTLIST = -O
myprog: mod1.o mod2.o
      cc -o myprog mod1.o mod2.o
      @echo "Today is `date`"  # or use .SILENT:
mod1.o: mod1.c mydefs.h   # This is a comment
      cc -c $(OPTLIST) mod1.c
      @echo "I sure wish it was lunch time"
mod2.o: mod2.c
      cc -c $(OPTLIST) mod2.c
$ touch mod1.c mod2.c
$ make  -f mymakeopt

The modules are compiled with optimization.
```

__ 6. Make a copy of mymake and call it mymakeopt. Change mymakeopt so that the **echo** actions are not output twice. Then, remove the **rm bogus** line from mymakeopt. Now, create a macro called OPTLIST. This is going to provide options to the compile line. Set OPTLIST to -O for optimization. Include the macro in the action lines associated with the mod1.o and mod2.o. What happens now when you execute **make** against mymakeopt?

```
$ touch mod1.c mod2.c
$ make -f mymakeopt "OPTLIST=-g"
```

__ 7. The reason for using macros is to make it easier for **make** actions to be customized on the command line. Re-execute **make**, overriding the OPTLIST macro on the command line with the -g option. Put the OPTLIST phrase in double quotes.

```
$ vi myclean
# The myclean file builds the myprog program
OPTLIST = -O
myprog: mod1.o mod2.o
      cc -o myprog mod1.o mod2.o
      @echo "Today is `date`"  # or use .SILENT:
mod1.o: mod1.c mydefs.h   # This is a comment
      cc -c $(OPTLIST) mod1.c
      @echo "I sure wish it was lunch time"
mod2.o: mod2.c
      cc -c $(OPTLIST) mod2.c
cleanup:
      -rm mod1.o mod2.o
      @echo "Object files have been removed"
$ make -f myclean cleanup
```

__ 8. Bring the file mod1.c under the custody of SCCS.

.

```
$ admin -imod1.c s.mod1.c
There are no SCCS identification keywords in the file. (cm7)
$ mv mod1.c mod1.c.orig
```

__ 9. Extract the file mod1.c for editing.

.

```
$ get -e s.mod1.c
1.1
new delta 1.2
19 lines
$ ls -l s.mod1.c mod1.c p.mod1.c
-rw-r--r--   1 guest   usr           377 Mar 25 10:14 mod1.c
-rw-r--r--   1 guest   usr            32 Mar 25 10:14 p.mod1.c
-r--r--r--   1 guest   usr           520 Mar 25 10:13 s.mod1.c
```

__ 10. Place the ID keywords in the first line to show the module name and the SID of the mod1.c file. Check this modified file.

```
$ vi mod1.c
static const char* SCCSID=”%M% %I%”;
...
$ delta s.mod1.c
Type comments, terminated with an End of File character
        or a blank line.
I made a small change
<hit enter>
1.2
1 inserted
0 deleted
19 unchanged
$ ls -l s.mod1.c mod1.c p.mod1.c
ls: 0653-341 The file mod1.c does not exist.
ls: 0653-341 The file p.mod1.c does not exist.
-r--r--r--   1 guest    usr            657 Mar 25 10:17 s.mod1.c
```

__ 11. Extract a read-only copy of mod1.c to see how the ID keywords are expanded.

.

```
$ get s.mod1.c
1.2
20 lines
$ head -1 mod1.c
static const char* SCCSID="mod1.c 1.2";
```

END OF LAB

Exercise 4. AIX Dynamic Binding and Shared Libraries

What This Exercise is About

This lab helps the student understand various aspects of dynamic binding and using shared libraries.

What You Should Be Able to Do

At the end of the lab, students should be able to:

- Dynamically bind a simple shared object to an application
- Create shared libraries and perform both static and dynamic binding to that library
- Use the dlopen() family of system calls to load a program module on the fly.

Introduction

The exercise deals with creating and using shared libraries.

Required Materials

- One pSeries or RS/6000 system with AIX 5L Version 5.1
- C for AIX

Exercise Instructions

__ 1. Create the following three short source files:

```
colors.c

red()
{
          printf("This is the red function.\n");
}

blue()
{
          printf("This is the blue function.\n");
}

green()
{
          printf("This is the green function.\n");
}

shapes.c

round()
{
           printf("This is the round function.\n");
}

square()
{
           printf("This is the square function.\n");
}

dims.c

area()
{
           printf("This is the area function.\n");
}

vol()
{
           printf("This is the vol function.\n");
}
```

__ 2. Compile the three files into object files (do not link them).

__ 3. Create the export file for the shared object you are about to create. It should include all the functions in your source files. If you specify the location for the exports file, make it a relative path name. The module will be called geometry.o and will be stored in the libgeolib.a library. Name the export file geometry.exp.

__ 4. Link the three object files into a shared object module named geometry.o. Specify the red() function as the entry point.

__ 5. Archive the object file into a library named libgeolib.a in your current directory.

__ 6. Create the following application source file named geoapp.c:

```
main()
{
        printf("This is main.\n");
        red();
        square();
        area();
        printf("Back in main.\n");
}
```

__ 7. Optionally, create the import file for geoapp, containing red, square, and area. Name it geoapp.imp. (Remember, since we will be binding to all shared objects, this step is really unnecessary.)

__ 8. Compile and link the application.

Note that because the geometry.o object was created as a shared object, it is not necessary to specify the imports for geoapp. The default is to automatically import all needed symbols from shared libraries.

__ 9. Try to run the geoapp program. It should run successfully. To see if the dynamic binding is running as expected, rename the library and run the application again. What happened?

____

__ 10. Actually, it is not necessary to create a shared library to perform dynamic binding. Create the following short source file named shrfoo.c:

```
foo()
{
        printf("This is the foo function\n");
}
```

__ 11. Create the following shrfoo.exp_imp file:

```
#!./shrfoo
foo
```

__ 12. Compile and link the file into a loadable object.

__ 13. Create the following application to call foo(). Call the source file foocaller.c:

```
main()
{
            printf("This is main calling foo...\n");
            foo();
            printf("Back to main.\n");
}
```

__ 14. Compile and link the application to the loadable object. Call the executable application foocaller. Use the same file that you used as the export file for shrfoo for the import file to foocaller.

__ 15. Test run the foocaller application. To demonstrate how the dynamic binding can work, modify the contents of the shrfoo.c file and recompile it into the shared/loadable object. Now rerun the foocaller application. The change should take effect without a need for relinking.

__ 16. Finally, we will create a module that can be loaded on the fly. Create the following program as lastapp.c:

```
#include <dlfcn.h>
#include <stdio.h>

main()
{
        void *lib_handle;
        int (*funcp)();
        char *error;

        lib_handle = dlopen("./loadmod", RTLD_NOW);
        if(error = dlerror()) {
                fprintf(stderr, "Error:%s \n",error);
                exit(1);
        }
        printf("Load was successful.\n");

        funcp = (int (*)()) dlsym(lib_handle, "func3");
        if(error = dlerror()) {
                fprintf(stderr, "Error:%s \n",error);
                exit(1);
        }
        (*funcp)();
```

```
                printf("\nUnloading loadmod.\n");

                dlclose(lib_handle);
                if(error = dlerror()) {
                        fprintf(stderr, "Error:%s \n",error);
                        exit(1);
                }
        }
```

__ 17. Compile the lastapp.c file:

__ 18. Create the following file as loadmod.c:

```
        func1(){
              printf("This is function 1.\n");
        }

        func2(){
              printf("This is function 2.\n");
        }

        func3(){
              printf("This is function 3.\n");
        }
```

__ 19. Compile the loadmod.c file to an object module.

__ 20. Link the loadmod.o file, using func2 as the entry point.

__ 21. Now test run the lastapp program.

END OF LAB

Exercise Instructions With Hints

__ 1. Create the following three short source files:

```
colors.c

red()
{
          printf("This is the red function.\n");
}

blue()
{
          printf("This is the blue function.\n");
}

green()
{
          printf("This is the green function.\n");
}

shapes.c

round()
{
           printf("This is the round function.\n");
}

square()
{
           printf("This is the square function.\n");
}

dims.c

area()
{
           printf("This is the area function.\n");
}

vol()
{
           printf("This is the vol function.\n");
}
```

__ 2. Compile the three files into object files (do not link them).

```
$ cc -c colors.c shapes.c dims.c
```

__ 3. Create the export file for the shared object you are about to create. It should include all the functions in your source files. If you specify the location for the exports file, make it a relative path name. The module will be called geometry.o and will be stored in the libgeolib.a library. Name the export file geometry.exp.

```
#!libgeolib(geometry.o)
red
blue
green
round
square
area
vol
```

__ 4. Link the three object files into a shared object module named geometry.o. Specify the red() function as the entry point.

```
$ ld -o geometry.o colors.o shapes.o dims.o \
> -bE:geometry.exp -bM:SRE -ered -lc
```

__ 5. Archive the object file into a library named libgeolib.a in your current directory.

```
$ ar -vq libgeolib.a geometry.o
```

__ 6. Create the following application source file named geoapp.c:

```
main()
{
   printf("This is main.\n");
   red();
   square();
   area();
   printf("Back in main.\n");
}
```

__ 7. Optionally, create the import file for geoapp, containing red, square, and area. Name it geoapp.imp. (Remember, since we will be binding to all shared objects, this step is really unnecessary.)

```
        #!libgeolib(geometry.o)
        red
        square
        area
```

__ 8. Compile and link the application.

```
  $ cc  -o geoapp geoapp.c  -L./ -lgeolib
```

Note that because the geometry.o object was created as a shared object, it is not necessary to specify the imports for geoapp. The default is to automatically import all needed symbols from shared libraries.

__ 9. Try to run the geoapp program. It should run successfully. To see if the dynamic binding is running as expected, rename the library and run the application again. What happened?

```
  $ geoapp
  Executes as expected
  $ mv libgeolib.a libmylib.a
  $ geoapp
  The program exits with the following error message:
  exec(): 0509-036 Cannot load program ./geoapp because of the
  following errors:
              0509-150   Dependent module libgeolib.a(geometry.o)
     could not be loaded.
              0509-022 Cannot load module libgeolib.a(geometry.o).
              0509-026 System error: A file or directory in the
     path name does not exist.
```

__ 10. Actually, it is not necessary to create a shared library to perform dynamic binding. Create the following short source file named shrfoo.c:

```
foo()
{
   printf("This is the foo function\n");
}
```

__ 11. Create the following shrfoo.exp_imp file:

```
#!./shrfoo
foo
```

__ 12. Compile and link the file into a loadable object.

```
$ cc -bE:shrfoo.exp_imp -o shrfoo shrfoo.c -efoo
```

__ 13. Create the following application to call foo(). Call the source file foocaller.c:

```
main()
{
            printf("This is main calling foo...\n");
            foo();
            printf("Back to main.\n");
}
```

__ 14. Compile and link the application to the loadable object. Call the executable application foocaller. Use the same file that you used as the export file for shrfoo for the import file to foocaller.

```
$ cc -bI:shrfoo.exp_imp -o foocaller foocaller.c
```

__ 15. Test run the foocaller application. To demonstrate how the dynamic binding can work, modify the contents of the shrfoo.c file and recompile it into the shared/loadable object. Now rerun the foocaller application. The change should take effect without a need for relinking.

```
$ foocaller
$ vi shrfoo.c
(add another printf line)
$ rm shrfoo
$ cc -bE:shrfoo.exp_imp -o shrfoo shrfoo.c -efoo
$ foocaller
(Two printf lines should be printed)
```

__ 16. Finally, we will create a module that can be loaded on the fly. Create the following program as lastapp.c:

```
#include <dlfcn.h>
#include <stdio.h>

main()
{
        void *lib_handle;
        int (*funcp)();
        char *error;

        lib_handle = dlopen("./loadmod", RTLD_NOW);
        if(error = dlerror())
        {
                fprintf(stderr, "Error:%s \n",error);
                exit(1);
        }
        printf("Load was successful.\n");

        funcp = (int (*)()) dlsym(lib_handle, "func3");
        if(error = dlerror())
        {
                fprintf(stderr, "Error:%s \n",error);
                exit(1);
        }
        (*funcp)();
        printf("\nUnloading loadmod.\n");

        dlclose(lib_handle);
        if(error = dlerror())
        {
                fprintf(stderr, "Error:%s \n",error);
                exit(1);
        }
}
```

__ 17. Compile the lastapp.c file:

```
$ cc -o lastapp lastapp.c
```

__ 18. Create the following file as loadmod.c:

```
func1(){
      printf("This is function 1.\n");
}

func2(){
      printf("This is function 2.\n");
}

func3(){
      printf("This is function 3.\n");
}
```

__ 19. Compile the loadmod.c file to an object module.

```
$ cc -c loadmod.c
```

__ 20. Link the loadmod.o file, using func2 as the entry point.

```
$ cc -o loadmod loadmod.o -e func2 -bexpall
```

__ 21. Now test run the lastapp program.

```
$ lastapp
Load was successful.
This is function 3.

Unloading loadmod.
$
```

END OF LAB

Exercise 5. AIX File and I/O System Calls

What This Exercise is About

This lab provides practice with AIX system calls used to access and manipulate files and file I/O.

What You Should Be Able to Do

At the end of the lab, students should be able to:

- Write simple C programs that create AIX data files.
- Use the shmat() system call to memory map AIX data files.

Introduction

The exercise deals with using file and I/O system calls.

Required Materials

- A pSeries or RS/6000 system with AIX 5L
- C for AIX

Exercise Instructions

__ 1. Write a C program that creates a file named mydata. Have the program loop five times, each time asking the user for a word. The words should be read in as strings, then written to the mydata file. Compile and run this program to create the mydata file. Check to see if all the input given to this program get recorded into the mydata file.

__ 2. Write a C program that opens the mydata file and creates a new file called mydata2. Have the program skip the first three characters in the file using the lseek() call, read only the fourth through tenth characters from the mydata file and write those characters to the mydata2 file. Compile and run this program. Verify the contents of the mydata2 file against the mydata file.

__ 3. Create a simple C program that counts the number of characters in an ASCII file. The name of the ASCII file should be given as an argument to the program upon execution. The program should first verify that the ASCII file exists and that the user has read access to the ASCII file. Compile and run this program. Verify the correctness of this program against the output from the **wc** command.

__ 4. Write a program to create a directory and to change the directory permissions to deny all access for group and others. Give full permissions on this directory for self. Create a symbolic link to this directory. Use the **ls** command to verify if the program works well.

__ 5. Enter and compile the writeit.c program from the lecture notes. Run the program so that it creates a file and writes 1,000,000 characters into it. Time the program using the AIX **time** command.

Enter and compile the mapit.c program from the lecture notes. Run the program so that it creates a file and writes 1,000,000 characters into it using memory mapping routines. Time the programs using the AIX **time** command.

Note the time difference between each program. Which program runs faster?

____

END OF LAB

Exercise Instructions With Hints

__ 1. Write a C program that creates a file named mydata. Have the program loop five times, each time asking the user for a word. The words should be read in as strings, then written to the mydata file. Compile and run this program to create the mydata file. Check to see if all the input given to this program get recorded into the mydata file.

```
/* rdwr.c */
/*   This program asks the user for five words and  */
/*   stores them in the file named "mydata".  */
/*   New lines are added. */
#include <fcntl.h>

main()
{
   int i, fd, rc;
   char word[32];

   if ((fd=open("./mydata",O_RDWR|O_CREAT,0666)) < 0 ) {
      perror("Could not open - See Instructor.\n");
      exit(1);
   }
   for(i=1; i<=5; i++) {
      printf("\n\nEnter a word: ");
      rc=read(0,word,sizeof(word));
      write(fd,word,rc);
   }
   close(fd);
}

$ cc rdwr.c -o rdwr
$ ./rdwr
$ cat mydata
```

The contents should be the same as the input given to *rdwr.*

__ 2. Write a C program that opens the mydata file and creates a new file called mydata2. Have the program skip the first three characters in the file using the lseek() call, read only the fourth through tenth characters from the mydata file and write those characters to the mydata2 file. Compile and run this program. Verify the contents of the mydata2 file against the mydata file.

```
/* lseektst.c */
/* This program opens the "mydata" file created in  */
/* the previous exercise and copies characters */
/* 4 through 10 to a file called "mydata2". */

#include <fcntl.h>
#include <unistd.h>

main()
{
   int i, fd1, fd2;
   char c;

   if ((fd1=open("./mydata",O_RDONLY)) < 0 ) {
      perror("Can't open mydata, see Instructor.\n");
      exit(1);
   }

   if ((fd2=open("./mydata2",O_WRONLY|O_CREAT,0666)) < 0) {
      perror("Can't open mydata2, see Instructor.\n");
      exit(2);
   }

   lseek(fd1,3,SEEK_SET);

   for(i=4; i<=10; i++) {
      read(fd1,&c,1);
      write(fd2,&c,1);
   }
   close(fd1);
   close(fd2);
}

$ cc lseektst.c -o lseektst
$ ./lseektst
$ vi mydata2
$ vi mydata
```

Characters 4 to 10 of the *mydata* file should be present in *mydata2* file.

__ 3. Create a simple C program that counts the number of characters in an ASCII file. The name of the ASCII file should be given as an argument to the program upon execution. The program should first verify that the ASCII file exists and that the user has read access to the ASCII file. Compile and run this program. Verify the correctness of this program against the output from the **wc** command.

```
/* count.c */
/* This program opens the file specified on the  */
/* command line and counts the number of  */
/* characters in the file. The results are  */
/* printed to standard out. */

#include <fcntl.h>

main(int argc, char *argv[])
{
   int fd, count = 0;
   char c;

   if (argc != 2) {
      printf("Usage -- %s filename\n",argv[0]);
      exit(1);
   }
   if ((fd=open(argv[1],O_RDONLY)) < 0 ) {
      perror("Can't open file, see Instructor.\n");
      exit(1);
   }
   while (read(fd,&c,1))
      count++ ;
   printf("The character count is %d.\n", count);
   close(fd);
}

$ cc count.c -o count
$ ./count count.c
$ wc -c count.c
```

The output from the *count* program and *wc* command should be equal.

__ 4. Write a program to create a directory and to change the directory permissions to deny all access for group and others. Give full permissions on this directory for self. Create a symbolic link to this directory. Use the **ls** command to verify if the program works well.

```
/* dirperms.c */
/* This program creates a directory */
/* and changes its permission to rwx------ */

#include <fcntl.h>

main(int argc, char *argv[])
{
   int fd, count;
   char c;

   if (argc != 3) {
      printf("Usage -- %s dirname linkname\n",argv[0]);
      exit(1);
   }
   if (mkdir(argv[1], 0700) == -1) {
      perror("Can't create directory.\n");
      exit(1);
   }
   if(symlink(argv[1], argv[2]) == -1) {
      perror("Can't create link.\n");
      exit(2);
   }
}

$ cc dirperms.c -o dirperms
$ ./dirperms mydir mylink
$ ls -ld mydir mylink
drwx------ 2 guest usr 512 Mar 15 16:47 mydir
lrwxrwxrwx 1 guest usr 5 Mar 15 16:47 mylink -> mydir
```

__ 5. Enter and compile the writeit.c program from the lecture notes. Run the program so that it creates a file and writes 1,000,000 characters into it. Time the program using the AIX **time** command.

Enter and compile the mapit.c program from the lecture notes. Run the program so that it creates a file and writes 1,000,000 characters into it using memory mapping routines. Time the programs using the AIX **time** command.

Note the time difference between each program. Which program runs faster?

```
/* writeit.c */
#include <fcntl.h>
#include <sys/types.h>
main()
{
   int i,fdw;
   char c='A';
   fdw=open("wfile",O_CREAT|O_RDWR, 0666);
   for (i=0;i<1000000;i++)
      write(fdw,&c,1);
   close(fdw);
   exit(0);
}

$ cc writeit.c -o writeit
$ time ./writeit
```

```
/* mapit.c */
#include <fcntl.h>
#include <sys/shm.h>
main()
{
   int i,fdw;
   char *ptr, *beg_ptr;
   fdw=open("mfile",O_CREAT|O_RDWR, 0666);
   ptr=shmat(fdw, 0, SHM_MAP);
   beg_ptr=ptr;
   for(i=0;i<1000000;i++)
      *ptr++='A';
   shmdt(beg_ptr);
   ftruncate(fdw,1000000);
   close(fdw);
   exit(0);
}

$ cc mapit.c -o mapit
$ time ./mapit
```

This program will run faster than the *writeit* program.

END OF LAB

Exercise 6. AIX Process Management System Calls

What This Exercise is About

This lab provides practice with AIX system calls that are used to manage processes.

What You Should Be Able to Do

At the end of the lab, students should be able to write simple C programs that utilize the AIX system calls that manage processes.

Introduction

This lab requires you to write four programs. On completion of all the four programs, you would have covered most of the process related system calls discussed in the unit.

The program in Step 4 is to change the priority of the process, for which you have to be a root user. You can see the priority change only if you are a root user; otherwise an error message is displayed.

Required Materials

- One pSeries (or RS/6000) system with the AIX 5L Version 5.1
- C for AIX

Exercise Instructions

__ 1. Log in to your assigned user ID. In your $HOME directory, create a directory named procmgmtlab. Change directory to procmgmtlab. Write a simple C program that displays its own process ID number as well as the PID of its parent. Have it also display its user ID (real or effective) and its group ID. Compile and run the program several times. Next, change the user to root and run the program. Set the set UID bit of the executable. Now logout from root and run the program as any other user.

Which process properties change and which ones remain the same?

____

__ 2. Alter the program from Step 1 to also display the process group ID, then fork a child process. Have the child process display its process group ID, then set its process group ID to its own PID. After the change, display the new process group ID of the child. Compile and execute the program.

__ 3. Write a program that reports its PID and PPID, then forks a child process that executes the first program you created in this lab. Have the parent process properly wait for the child process to exit. Compile and test the program.

__ 4. Write a C program that performs an endless loop. Within the loop, have the process sleep for 10,000 microseconds (refer to the online documentation on the usleep system call, or try the **man** command), then wake and report its priority value. Compile and run the program in the foreground. After a while, stop the program by using the Ctrl-C keystroke.

Now, change the program to loop 10 times, then check the UID to see if the user is root. Change the priority to 40 and print the new priority. Execute the program as your regular user ID and then as root.

END OF LAB

Exercise Instructions With Hints

__ 1. Log in to your assigned user ID. In your $HOME directory, create a directory named procmgmtlab. Change directory to procmgmtlab. Write a simple C program that displays its own process ID number as well as the PID of its parent. Have it also display its user ID (real or effective) and its group ID. Compile and run the program several times. Next, change the user to root and run the program. Set the set UID bit of the executable. Now logout from root and run the program as any other user.

Which process properties change and which ones remain the same?

____

```
$ mkdir procmgmtlab
$ cd procmgmtlab
$ vi disp_IDs.c

/* Program name - disp_IDs.c */
/*   This program displays its PID, PPID, RUID,    */
/*   EUID and GID   */
#include <stdio.h>
main()
{
   printf("My PID is %d.\n", getpid());
   printf("My PPID is %d.\n", getppid());
   printf("My RUID is %d.\n", getuid());
   printf("My EUID is %d.\n", geteuid());
   printf("My GID is %d.\n", getgid());
}
/* Program ends here */
```

Steps to complete the exercise:

1. `cc disp_IDs.c -o disp_IDs`
2. `su`
3. Run the program.
4. `chmod u+s disp_IDs`
5. Run the program as root and non-root users.

The parameters that change are PID, and EUID.

__ 2. Alter the program from Step 1 to also display the process group ID, then fork a child process. Have the child process display its process group ID, then set its process group ID to its own PID. After the change, display the new process group ID of the child. Compile and execute the program.

```
/* Program name - set_child_PGID.c */
/*   This program displays its PID, PPID, RUID,   */
/*   EUID , GID and PGID   */
#include <stdio.h>
main()
{
   printf("My PID is %d.\n", getpid());
   printf("My PPID is %d.\n", getppid());
   printf("My RUID is %d.\n", getuid());
   printf("My EUID is %d.\n", geteuid());
   printf("My GID is %d.\n", getgid());
   printf("My PGID is %d.\n", getpgrp());
   if (fork()==0) {
      printf ("The child PGID is currently %d.\n"),
             getpgrp());
      printf ("Changing the child process group ID.\n");
      setpgid(0,0);
      printf ("The child PGID is now %d.\n"), getpgrp());
   }
}

$ cc set_child_PGID.c -o set_child_PGID
```

__ 3. Write a program that reports its PID and PPID, then forks a child process that executes the first program you created in this lab. Have the parent process properly wait for the child process to exit. Compile and test the program.

```
/* Program name - demo_exec.c */
/*   This program displays its PID and PPID,  */
/*   then forks a new process and waits  */
#include <stdio.h>
main()
{
   printf("My PID is %d.\n", getpid());
   printf("My PPID is %d.\n", getppid());
   if (fork()==0) {
      printf ("Executing program 1...\n");
      execl("/home/teamxx/procmgmtlab/disp_IDs", "disp_IDs",0);
      printf(“Error executing program 1\n”);
      exit(1);
   }
   wait (NULL);
   printf ("Returned to parent...\n");
}

$ cc demo_exec.c -o demo_exec
```

__ 4. Write a C program that performs an endless loop. Within the loop, have the process sleep for 10,000 microseconds (refer to the online documentation on the usleep system call, or try the **man** command), then wake and report its priority value. Compile and run the program in the foreground. After a while, stop the program by using the Ctrl-C keystroke.

Now, change the program to loop 10 times, then check the UID to see if the user is root. Change the priority to 40 and print the new priority. Execute the program as your regular user ID and then as root.

```
/* Program name - check_pri1.c */
/* This program reports its priority every */
/* 10,000 microseconds */
#include <stdio.h>
main()
{
   while(1) {
      printf("My priority is %d.\n", getpri(0));
      usleep(10000);
   }
}

$ cc check_pri1.c -o check_pri1
```

```
/* Program name - check_pri2.c */
/*  This program reports its priority every 10,000  */
/*  microseconds, then attempts to change the priority */
#include <stdio.h>
main()
{
   int i;
   for (i=1;i<=10;i++) {
      printf("My priority is %d.\n", getpri(0));
      usleep(10000);
   }
   printf("Attempting to change the priority...\n");
   if (getuid()==0 || geteuid()==0) {
      setpri(0,40);
      printf("My priority is now %d.\n", getpri(0));
   }
   else
      printf("You do not have authority to change the
      priority.\n");

   printf("My priority is still %d.\n", getpri(0));
}

$ cc check_pri2.c -o check_pri2
```

END OF LAB

Exercise 7. AIX Signal Management System Calls

What This Exercise is About

This lab provides practice with the AIX system calls used to manage signals.

What You Should Be Able to Do

At the end of the lab, students should be able to:

- Write simple C programs that utilize the AIX system calls that manage signals.

Required Materials

- A pSeries or RS/6000 system with AIX 5L.
- C for AIX compiler

Exercise Instructions

__ 1. To do this exercise, you need either to be in X-Windows or have another telnet session to the same machine. Open a second window (or, if not in X-Windows, have a second ASCII terminal to provide the second session). Create a program that reports its process ID in a infinite loop. Have a sleep time of one second between each process ID reporting. Use the sigaction() system call to register a signal handler that ignores signals 2 (SIGINT) and signal 15 (SIGTERM). Do not forget to include the proper files. Examine the signal.h file for help.

Run the program in the first window. After a while, try to stop the program by using the <Ctrl>C keystroke. Did it stop the program? Try stopping the program by issuing the **kill** command from the second window. (Use the AIX online documentation to learn the syntax of the **kill** command.) If **kill** did not work, find a **kill** variation that cannot be trapped, thus killing this program.

__ 2. Modify the program in step Step 1 so that it prints the following message whenever a SIGINT or SIGTERM signal is received, then keeps running:

```
I hear you knocking but you can't come in!
```

Run the program in the first window. After a while, try to stop the program by using the <Ctrl>C keystroke. Does the program print any message? Kill this program before proceeding to the next question.

__ 3. Create a program that sends a SIGUSR1 signal to a process. Have it send the signal to the process with an ID that is provided by the user as an argument to the invocation of the program. Have it verify the existence of the desired process prior to sending the signal.

Create a program in the other window to accept the SIGUSR1 signal. Print a message that the signal was received, then terminate. Let this program print its process ID also. Run this program. Next, run the program that sends the SIGUSR1 signal. Pass the PID of the other program as an argument.

__ 4. Now we will create a program that emulates an alarm clock. To accomplish this, we will use the alarm() system call. The alarm() function causes the system to send the calling process a SIGALARM signal after the specified number of seconds. (Refer to the AIX online documentation for more details, if needed). Another function, pause(), takes no arguments and is used to suspend the calling process until the delivery of a signal whose action is to either execute a signal-catching function or terminate the process. pause() and alarm() can be combined to put a process to sleep for a specified number of seconds.

Write a program that utilizes these two system calls plus, your own alarm catching function, to print the current time, schedule an alarm for ten seconds, pause, and then print the new time. To print the time, you may use the **date** command.

END OF LAB

Exercise Instructions With Hints

__ 1. To do this exercise, you need either to be in X-Windows or have another telnet session to the same machine. Open a second window (or, if not in X-Windows, have a second ASCII terminal to provide the second session). Create a program that reports its process ID in a infinite loop. Have a sleep time of one second between each process ID reporting. Use the sigaction() system call to register a signal handler that ignores signals 2 (SIGINT) and signal 15 (SIGTERM). Do not forget to include the proper files. Examine the signal.h file for help.

Run the program in the first window. After a while, try to stop the program by using the <Ctrl>C keystroke. Did it stop the program? Try stopping the program by issuing the **kill** command from the second window. (Use the AIX online documentation to learn the syntax of the **kill** command.) If **kill** did not work, find a **kill** variation that cannot be trapped, thus killing this program.

```
/* ignore.c */
/* This program reports its PID every second */
/* It also ignores SIGINT and SIGTERM signals */
#include <signal.h>

main()
{
   struct sigaction mysig;
   mysig.sa_handler = SIG_IGN;
   sigemptyset(&mysig.sa_mask);
   mysig.sa_flags = 0;
   sigaction(SIGINT, &mysig,NULL);
   sigaction(SIGTERM, &mysig,NULL);
   while (1) {
      printf("My PID is %d.\n", getpid());
      sleep(1);
   }
}

$ cc -o ignore ignore.c
$ ./ignore
My PID is 22740.
^CMy PID is 22740.
...

From another window,
$ kill -9 22740
Output in first window:
[1] + Killed                    ./ignore

This program cannot be killed using the <Ctrl>C keystroke.
This program can be killed using the signal SIGKILL.
```

__ 2. Modify the program in step Step 1 so that it prints the following message whenever a SIGINT or SIGTERM signal is received, then keeps running:

```
I hear you knocking but you can't come in!
```

Run the program in the first window. After a while, try to stop the program by using the <Ctrl>C keystroke. Does the program print any message? Kill this program before proceeding to the next question.

```
/* knock.c */
/*This program reports its PID every second*/
/*It also catches SIGINT and SIGTERM signals.          */
#include <signal.h>

void deal_with_it(int dummy)
{
   printf("I hear you knocking but you can't come in!\n");
}

main()
{
   struct sigaction mysig;
   mysig.sa_handler = deal_with_it;
   mysig.sa_flags = 0;
   sigemptyset(&mysig.sa_mask);

   sigaction(SIGINT, &mysig,NULL);
   sigaction(SIGTERM, &mysig,NULL);
   while (1) {
      printf("My PID is %d.\n", getpid());
      sleep(1);
   }
}

$ ./knock
My PID is 19392.
My PID is 19392.
^CI hear you knocking but you can't come in!
...

This program prints a message when the <Ctrl>C keystroke is
used.
From another window,
$ kill -9 19392
Output in first window:
[1] + Killed                       ./knock
```

__ 3. Create a program that sends a SIGUSR1 signal to a process. Have it send the signal to the process with an ID that is provided by the user as an argument to the invocation of the program. Have it verify the existence of the desired process prior to sending the signal.

Create a program in the other window to accept the SIGUSR1 signal. Print a message that the signal was received, then terminate. Let this program print its process ID also. Run this program.

Run the program that sends the SIGUSR1 signal. Pass the PID of the other program as an argument.

```
/* getsig.c */
/* This program reports its PID */
/* It also catches SIGINT and SIGTERM signals. */

#include <signal.h>

void deal_with_it(int dummy)
{
   printf("I received a SIGUSR1 signal\n");
   exit(0);
}

main()
{
   struct sigaction mysig;

   mysig.sa_handler = deal_with_it;
   mysig.sa_flags = 0;
   sigemptyset(&mysig.sa_mask);
   sigaction(SIGUSR1, &mysig,NULL);
   printf("My PID is %d.\n", getpid());

   /* infinite loop */
   while (1) ;
}

$ cc -o getsig getsig.c
```

This program will stop after it receives a *SIGUSR1* signal from the other program.

```
/* sendsig.c */
/* This program sends a SIGUSR1 signal to */
/* the PID provided as arg 1 by */
/* the user. It first verifies the existence */
/* of the process. */
#include <signal.h>

main(int argc, char *argv[])
{
   int rv;
   int target_pid;

   if ( argc < 2 ) {
      printf("Usage: %s target_pid\n", argv[0]);
      exit(1);
   }

   target_pid = atoi(argv[1]);

   rv = kill(target_pid, 0);   /*  test existence  */
   if ( rv < 0 ) { /*  PID doesn't exist  */
      printf("PID %d does not exist.\n", target_pid);
      exit(2);
   }

   rv = kill(target_pid, SIGUSR1);
   if ( rv < 0 ) {
      printf("Unable to send signal.\n");
      printf("Check UID for permission.\n");
      exit(3);
   }
}

$ cc -o sendsig sendsig.c
$ ./getsig
My PID is 22756.
From another window:
$ ./sendsig 22756
The output in the first window is
I received a SIGUSR1 signal
```

__ 4. Now we will create a program that emulates an alarm clock. To accomplish this, we will use the alarm() system call. The alarm() function causes the system to send the calling process a SIGALARM signal after the specified number of seconds. (Refer to the AIX online documentation for more details, if needed). Another function, pause(), takes no arguments and is used to suspend the calling process until the delivery of a signal whose action is to either execute a signal-catching function or terminate the process. pause() and alarm() can be combined to put a process to sleep for a specified number of seconds.

Write a program that utilizes these two system calls plus, your own alarm catching function, to print the current time, schedule an alarm for ten seconds, pause, and then print the new time. To print the time, you may use the **date** command.

```
/* alarm.c */
/* This program emulates an alarm clock */

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void catch_alrm (int signo)
{
   printf(“Just got a SIGALRM signal.\n“);
   return;
}
main()
{
   struct sigaction mysig;
   mysig.sa_handler = catch_alrm;
   mysig.sa_flags = 0;
   sigemptyset(&mysig.sa_mask);
   sigaction(SIGALRM, &mysig, NULL); /* reg. alarm signal */
   system("date");
   printf("Setting our alarm clock...\n");
   alarm (10);   /*schedule alarm for 10 sec. from now*/
   pause ();    /* pause until signal received */
   system("date");
   return 0;
}

$ cc -o alarm alarm.c
$ ./alarm
Wed Mar 27 13:29:28 CST 2002
Setting our alarm clock...
Just got a SIGALRM signal.
Wed Mar 27 13:29:38 CST 2002
```

END OF LAB

Exercise 8. Interprocess Communications

What This Exercise is About

This lab gives the student an opportunity to experiment with one or more of the IPC mechanisms by writing or completing programs.

What You Should Be Able to Do

At the end of the lab, students should be able to:

- Write a program using unnamed pipes.
- Write a pair of simple programs that pass strings of data between them using shared memory.
- Write programs that use the semaphore "file lock" example shown in the lecture.
- Complete a client and server set of programs that use message queues to process requests for data base lookups.
- Experiment with the socket example shown in the lecture.

Introduction

The lab exercises require that the students write five programs. The exercises cover pipes, shared memory, semaphores, message queues, and sockets.

On completion of the exercises, the students will have revisited all the topics and system calls discussed in the lecture.

Required Materials

- One pSeries (or RS/6000) system with AIX 5L Version 5.1
- C for AIX

Exercise Instructions

__ 1. Login as any user and change to your home directory. Create a directory named ipclab. Change to the ipclab directory.

__ 2. **Pipes:** Write, compile, test, and debug a program that passes one phrase from the child program to the parent program. Use unnamed pipes to accomplish it.

__ 3. **Shared Memory:** Write, compile, test and debug two programs that use shared memory. Have the first program create and attach a shared memory segment, then write a simple message into the segment. Have the second program attach the shared memory segment, read the message left by the first program, then detach from and remove the shared memory segment.

Run the first program and let it conclude before running the second program. This will demonstrate how shared memory segments survive beyond the life of the process that creates them. The second process will destroy the shared memory segment.

Remember that both programs must use the same key.

As an extra challenge, do not tell the second program how many characters are in the message written by the first program. Use something in the message to terminate the reading of the message by the second program.

Feel free to use the shared memory program from the lecture notes as a model.

Note: The solution provided is more complex than the model in the lecture notes.

__ 4. **Semaphores:** Create a source file that contains the semaphore example for the file locking that was described in the lecture.

After creating the source file containing locker() and unlocker(), create two other programs, each in separate modules, that will use the locker() and unlocker() routines. This means that the above source file must be included as an argument on the command line for the compilation of each of the other two programs. The other programs should attempt to set a lock on some file, then open the file and write data into the file. If possible, test the two programs by running them simultaneously in different AIX windows.

Continue to experiment with the semaphore example by trying different flags for controlling the semop() call's reaction to failing to gain the lock. For example, if the lock is not available, wait for the lock. Or, if the lock is not available, return a failure message to the user. Use AIX online documentation to find out what flags are available for the semop() system call.

__ 5. **Message Queues:** Create, compile, and test a client and server model of the baseball team lookup example presented in the lecture. To help you, the following files have been placed in a directory on your system. Please check with the instructor for the location:

bbcli.c	The client program

bbsvr.c	The server program
bball.h	A header file that defines the record structure of each entry in the baseball database
bball.db	The database of cities and baseball teams

The bbcli.c and bbsvr.c files have certain important lines of code missing. These lines are marked by comments. Your job is to replace these comments with working code.

The following is the incomplete code of bbcli.c:

```
/* Program name - bbcli.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
bbmsg_t msgout, msgin;
int msgid;
msgout.mtype=1;
msgout.mpid=getpid();
if((msgid=msgget(MSGKEY,0))<0) {
         perror("Could not get queue...");exit(1);
}
/*  Ask user for city and assign it to msgout.mcity  */
if(msgsnd(msgid,&msgout,sizeof(msgout),IPC_NOWAIT)<0) {
         perror("Could not send message...");exit(1);
}
msgrcv(msgid,&msgin,sizeof(msgin),getpid(),0);
/*  Display the team name found in msgin.mteam  */
...
```

The following is the incomplete code of bbsvr.c:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
bbmsg_t msgout, msgin;
int msgid;
if((msgid=msgget(MSGKEY,IPC_CREAT|0660))<0) {
         perror("Could not create queue...");exit(1);
}
msgrcv(msgid,&msgin,sizeof(msgin),1,0);
msgout.mtype=msgin.mpid;
msgout.mpid=0;
strcpy(msgout.mcity,msgin.mcity);
/*  Lookup the baseball team for msgin.mcity and assign the
```

```
results to msgout.mteam  */
if(msgsnd(msgid,&msgout,sizeof(msgout),IPC_NOWAIT)<0)
         perror("Could not send message...");
...
```

Copy these files into a directory under your home directory. Note that the msgin and msgout buffers are already defined for you. For the client to send and the server to receive, the message buffer format is:

```
...
struct msgout {      /* for the client */
     mtyp_t mtype;      /* message type */
     short mpid;        /* client's PID */
     char mtext[20];      /* message text */
} mymsgout;
...
```

To save time, much of the client and server program I/O routines are already provided. Your job is to add the appropriate header files, key generation, queue generation, and message handling. Feel free to refer to the lecture notes for help.

Here is how it all should work:

The server program should be started first. It should create the message queue, open the database file, then enter an endless loop, where it waits for an incoming message request of type 1.

The server must use the msgin.mpid value of the client's PID from the incoming message, as the client's PID will be used by the server as the mtype value when sending a return message to the client. The return message will be sent via a msgout structure of type mymsgout.

The server then looks up the city value found in mymsgin.mtext in the baseball database. This code has been provided. The team name is then assigned to mymsgout.mtext and the return message is sent to the client. If no match was found for the city, "No Team" is returned to the client.

The client program needs to send its own PID value as part of the message string sent to the server. This is so that the server knows who to send the response back to. The getpid() call returns a process' PID, but in integer form.

The client program should start by accessing the message queue created by the server. If the queue cannot be accessed, print a message to the user to check that the server is up.

The client program then enters an "endless" loop, asking the user for a city. If the user enters the word "quit", the client program ends. Otherwise, the integer value of the client's PID and the city value are packaged into the mymsgout.mpid and mymsgout.mtext fields respectively and sent to the message queue as a message type 1.

The client then waits for a message on the queue of type equal to the client's PID. The response from the server contains the baseball team's name in the msgin.mtext field. The client should display this to the user, then loop to ask for another city.

Note: The following shell script can be used to generate the database file:

```
#!/usr/bin/ksh
# Shell script name - gen_db

echo "Pittsburgh\0          Pirates\0            " > bball.db
echo "Boston\0              Red Sox\0            " >> bball.db
echo "Chicago\0             Cubs\0               " >> bball.db
echo "Atlanta\0             Braves\0             " >> bball.db
echo "Los Angeles\0         Dogders\0            " >> bball.db
echo "Toronto\0             Blue Jays\0          " >> bball.db
echo "Houston\0             Astros\0             " >> bball.db
echo "END\0                 No Team\0            " >> bball.db
```

__ 6. **Sockets:**

Either create (or peek at the solution at this point!) two socket programs meant to work on a inter-CPU basis.

- They should use sock-stream to handle the transfer.
- The parent process should register itself at port 7000 on your PC. (If there are multiple people on your PC, then add your team ID number to the port number to make yours unique) (Try entering the **hostname** command to find out your CPU, and then **host *your_hostname*** to find out your IP address.)

END OF LAB

Exercise Instructions With Hints

__ 1. Login as any user and change to your home directory. Create a directory named ipclab. Change to the ipclab directory.

```
    $ cd $HOME
    $ mkdir ipclab
    $ cd ipclab
```

__ 2. **Pipes:** Write, compile, test, and debug a program that passes one phrase from the child program to the parent program. Use unnamed pipes to accomplish it.

```
/* Program name - unnamedpipe.c */
#include <sys/signal.h>
#include <stdlib.h>
#include <fcntl.h>

int p[2];

int main(void)
{
   int read_count;
   int pid;
   char parentbuf[25];
   char childbuf[25];

   struct sigaction mysig;
   sigset_t mymask;

   void end_pgm(int signo);

   mysig.sa_handler = end_pgm;
   sigfillset(&mymask);
   mysig.sa_mask = mymask;
   sigaction(SIGINT, &mysig, NULL);
   sigaction(SIGQUIT, &mysig, NULL);
   sigaction(SIGTERM, &mysig, NULL);

   pipe(p);
   pid = fork();
   if (pid == -1) {
      perror("parent: problems with fork"); exit(1);
   }
/* CONTINUED ON NEXT SCREEN */
/* unnamedpipe.c  continued */

   if (pid == 0) {
      close(p[0]);
      strcpy(childbuf, "Hi there from child!\n");
      if ((write(p[1], childbuf, sizeof(childbuf))) < 0){
          perror("child: problems with writing to pipe");
          exit(2);
      }
      close(p[1]);
      exit(EXIT_SUCCESS);
   }
   close (p[1]);
   for(;;) {
      read_count = read(p[0], parentbuf, sizeof(parentbuf));
      if (read_count > 0) {
          printf("processing request\n");
          printf("request is: %s\n", parentbuf);
      }
      else {
          sleep(1);
      }
   }
}

void end_pgm(int signo)
{
   close(p[0]);
   printf("Parent dying off... Signal %d received.\n", signo);
   exit(EXIT_SUCCESS);
}

$ cc unnamedpipe.c -o unnamedpipe
```

__ 3. **Shared Memory:** Write, compile, test and debug two programs that use shared memory. Have the first program create and attach a shared memory segment, then write a simple message into the segment. Have the second program attach the shared memory segment, read the message left by the first program, then detach from and remove the shared memory segment.

Run the first program and let it conclude before running the second program. This will demonstrate how shared memory segments survive beyond the life of the process that creates them. The second process will destroy the shared memory segment.

Remember that both programs must use the same key.

As an extra challenge, do not tell the second program how many characters are in the message written by the first program. Use something in the message to terminate the reading of the message by the second program.

Feel free to use the shared memory program from the lecture notes as a model.

Note: The solution provided is more complex than the model in the lecture notes.

```
/* Program name - shm1.c */

#include <sys/types.h>
#include <sys/shm.h>
#define SHMKEY 333333L
#define INFOSIZE 256000

main()
{
   int shmid;
   char mymsg[] = "Hi!!  Here's your message....\n";

   struct info {
      int type;
      int length;
      char msg [256];
   } *shmptr, *shmptr_beg;

   shmid= shmget(SHMKEY,INFOSIZE,IPC_CREAT|0660);
   if ((shmptr = (struct info *)shmat(shmid,0,0)) < 0)
      perror("probs with shmat");
   shmptr_beg= shmptr;

   shmptr->type = 1;
   shmptr->length = strlen(mymsg) + 1 +
     sizeof(shmptr->type) + sizeof(shmptr->length);
   strcpy(shmptr->msg,mymsg);

   shmdt(shmptr_beg);
   /* let shm2 delete the shared memory    */
}

$ cc shm1.c -o shm1
```

```
/* shm2.c */

#include <sys/types.h>
#include <sys/shm.h>
#define SHMKEY 333333L
#define INFOSIZE 256000

main()
{
   int shmid;

   struct shmid_ds mystat;
   struct info {
      int type;
      int length;
      char msg [256];
   } *shmptr, *shmptr_beg;

   if ((shmid = shmget(SHMKEY,INFOSIZE,0)) <0){
      perror("problems with shmget"); exit(1);
   }
   if ((shmptr = (struct info *)shmat(shmid,0,0)) < 0){
      perror("probs with shmat"); exit(1);
   }
   shmptr_beg= shmptr;

   printf("The message says:  \n\n%s\n", shmptr->msg);

   /* If we wanted, we could use the length field */
   /* to figure out how much to read.*/

   /* Note that if we would advance the pointer */
   /* (shmptr++), we would be going ahead one full */
   /* structure */

   /* We could also have used another field  */
   /* or separate control record to */
   /* figure out how many structures to read...OR */
   /* we could use semaphores....CONT. ON NEXT SCREEN */
/* CONTINUED */

   mystat.shm_nattch = 2;
   /* Make sure no messages on queue before it's */
   /* removed.... Remember, you have to be the   */
   /* owner to remove it.  You can set it from   */
   /* the original application with -            */

   /* shmctl(shmid,IPC_STAT,&mystat)              */
   /* mystat.shm_perm.uid = xxx;                  */
   /* shmctl(shmid,IPC_SET,&mystat)               */

   while(mystat.shm_nattch > 1)
   {
      sleep(1);
      shmctl(shmid,IPC_STAT,&mystat);
   }

   if ((shmdt(shmptr_beg)) < 0){
      perror("problems in detaching memory");exit(1);
   }
   if ((shmctl(shmid, IPC_RMID, NULL)) <0){
      perror("problems in removing shmid"); exit(1);
   }
}

$ cc shm2.c -o shm2
$ ./shm1
$ ./shm2
```

__ 4. **Semaphores:** Create a source file that contains the semaphore example for the file locking that was described in the lecture.

After creating the source file containing locker() and unlocker(), create two other programs, each in separate modules, that will use the locker() and unlocker() routines. This means that the above source file must be included as an argument on the command line for the compilation of each of the other two programs. The other programs should attempt to set a lock on some file, then open the file and write data into the file. If possible, test the two programs by running them simultaneously in different AIX windows.

Continue to experiment with the semaphore example by trying different flags for controlling the semop() call's reaction to failing to gain the lock. For example, if the lock is not available, wait for the lock. Or, if the lock is not available, return a failure message to the user. Use AIX online documentation to find out what flags are available for the semop() system call.

```
/* Program name - lockit.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

extern int semid;

locker(int num)
{
   static struct sembuf lock_it[2] = {
      0 , 0 , 0 ,
      0 , 1 , 0
   };

   lock_it[0].sem_num = num; /* sem number to test */
   lock_it[1].sem_num = num; /* sem number to lock */

   if ((semop(semid,&lock_it[0],2)) <0)
      return 1;
   else
      return 0;
};

unlocker (int num)
{
   static struct sembuf unlock_it[1] = {
      0 , -1 , 0
   };

   unlock_it[1].sem_num = num; /*sem # to unlock */

   if ((semop(semid,&unlock_it[0],1)) <0)
      return 1;
   else
      return 0;
}

$ cc -c lockit.c
```

```
/* Program name - sem1.c */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/signal.h>
#define SEMKEY 888888L

int semid;
extern int locker(int num);
extern int unlocker(int num);

/* Be sure to include lockit.c when creating this mod */
/* cc -c sem1.c;cc -c lockit.c */
/* cc -o sem1 sem1.o lockit.o */

main()
{
   sigset_t mymask;
   struct sigaction mysig;

   sigemptyset (&mymask);
   mysig.sa_handler = SIG_IGN;
   /* I don't care about cleanup since I'm the */
   /* one issuing the SIGUSR1 at the end */
   mysig.sa_flags = 0;
   mysig.sa_mask = mymask;
   sigaction (SIGUSR1,&mysig,NULL);
   /* prepare for SIGUSR1 before semctl done */
/* CONTINUED ON NEXT SCREEN ..............*/
/* CONTINUED */

   semid=semget(SEMKEY,1,IPC_CREAT|0660);
   if (locker(0)  !=  0) {
      perror("Problems locking semaphore"); exit(1);
   } else
      printf("Locked semaphore and pid is %d\n", getpid());

   if (fork() == 0) /* start up competing child in*/
      execl("./sem2","sem2",0);/* same process group */

   sleep(10);  /* simulate interim activity  */

   if (unlocker(0)  != 0) {
      perror("Problems unlocking semaphore"); exit(1);
   } else
      printf("Unlocked semaphore and pid is %d\n", getpid());

   sleep(4); /* Let sem2 get started in the lock process */

   killpg(getpid(), SIGUSR1); /* warn that you're coming */
   /* down to all procs in current process group */

   sleep(5);

   printf("Bringing down semaphore set....\n");

   if ((semctl(semid,0,IPC_RMID,NULL)) <0) {
      perror("problems removing semaphore"); exit(1);
   }
   /* repeat for all semaphores in set */

   exit(0);
}

$ cc -c sem1.c
$ cc -o sem1 sem1.o lockit.o
```

```
/* Program name - sem2.c */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/signal.h>
#define SEMKEY 888888L

int semid;
extern int locker(int num);
extern int unlocker(int num);
static void handler(int signo);

/* Be sure to include lockit.c when creating this mod.*/
/* cc -c sem2.c;cc -c lockit.c */
/* cc -o sem2 sem2.o lockit.o */

main()
{
   sigset_t mymask;
   struct sigaction mysig;

   sigfillset( &mymask);
   mysig.sa_handler = handler;
   mysig.sa_flags = 0;
   mysig.sa_mask = mymask;
   sigaction(SIGUSR1,&mysig,NULL);
    /* prepare for SIGUSR1 before semctl done */
   semid=semget(SEMKEY,1,IPC_CREAT|0660);
   if (locker(0)  !=  0) {
      perror("Problems locking semaphore");exit(1);
   }
   else
      printf("Locked semaphore and pid is %d\n", getpid());

   sleep(10);  /* simulate interim activity  */

/* CONTINUED ON NEXT SCREEN.............*/
/*sem2.c*/    /* CONTINUED */

      if (unlocker(0)  != 0) {
         perror("Problems unlocking semaphore");exit(1);
      }  else
         printf("Unlocked semaphore and pid is %d\n", getpid());

      printf("Exiting....\n");
      exit(0);
   }

   void handler(int signo)
   {
      printf("Doing cleanup activity and exiting....\n");
      exit(1);
   }

   $ cc -c sem2.c
   $ cc -o sem2 sem2.o lockit.o
```

__ 5. **Message Queues:** Create, compile, and test a client and server model of the baseball team lookup example presented in the lecture. To help you, the following files have been placed in a directory on your system. Please check with the instructor for the location:

bbcli.c	The client program
bbsvr.c	The server program
bball.h	A header file that defines the record structure of each entry in the baseball database
bball.db	The database of cities and baseball teams

The bbcli.c and bbsvr.c files have certain important lines of code missing. These lines are marked by comments. Your job is to replace these comments with working code.

The following is the incomplete code of bbcli.c:

```
/* Program name - bbcli.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
bbmsg_t msgout, msgin;
```

```
int msgid;
msgout.mtype=1;
msgout.mpid=getpid();
if((msgid=msgget(MSGKEY,0))<0) {
         perror("Could not get queue...");exit(1);
}
/*  Ask user for city and assign it to msgout.mcity  */
if(msgsnd(msgid,&msgout,sizeof(msgout),IPC_NOWAIT)<0) {
         perror("Could not send message...");exit(1);
}
msgrcv(msgid,&msgin,sizeof(msgin),getpid(),0);
/*  Display the team name found in msgin.mteam  */
...
```

The following is the incomplete code of bbsvr.c:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
bbmsg_t msgout, msgin;
int msgid;
if((msgid=msgget(MSGKEY,IPC_CREAT|0660))<0) {
         perror("Could not create queue...");exit(1);
}
msgrcv(msgid,&msgin,sizeof(msgin),1,0);
msgout.mtype=msgin.mpid;
msgout.mpid=0;
strcpy(msgout.mcity,msgin.mcity);
/*  Lookup the baseball team for msgin.mcity and assign the
results to msgout.mteam  */
if(msgsnd(msgid,&msgout,sizeof(msgout),IPC_NOWAIT)<0)
         perror("Could not send message...");
...
```

Copy these files into a directory under your home directory. Note that the msgin and msgout buffers are already defined for you. For the client to send and the server to receive, the message buffer format is:

```
...
struct msgout {      /* for the client */
     mtyp_t mtype;      /* message type */
     short mpid;        /* client's PID */
     char mtext[20];      /* message text */
} mymsgout;
...
```

To save time, much of the client and server program I/O routines are already provided. Your job is to add the appropriate header files, key generation, queue generation, and message handling. Feel free to refer to the lecture notes for help.

Here is how it all should work:

The server program should be started first. It should create the message queue, open the database file, then enter an endless loop, where it waits for an incoming message request of type 1.

The server must use the msgin.mpid value of the client's PID from the incoming message, as the client's PID will be used by the server as the mtype value when sending a return message to the client. The return message will be sent via a msgout structure of type mymsgout.

The server then looks up the city value found in mymsgin.mtext in the baseball database. This code has been provided. The team name is then assigned to mymsgout.mtext and the return message is sent to the client. If no match was found for the city, "No Team" is returned to the client.

The client program needs to send its own PID value as part of the message string sent to the server. This is so that the server knows who to send the response back to. The getpid() call returns a process' PID, but in integer form.

The client program should start by accessing the message queue created by the server. If the queue cannot be accessed, print a message to the user to check that the server is up.

The client program then enters an "endless" loop, asking the user for a city. If the user enters the word "quit", the client program ends. Otherwise, the integer value of the client's PID and the city value are packaged into the mymsgout.mpid and mymsgout.mtext fields respectively and sent to the message queue as a message type 1.

The client then waits for a message on the queue of type equal to the client's PID. The response from the server contains the baseball team's name in the msgin.mtext field. The client should display this to the user, then loop to ask for another city.

Note: The following shell script can be used to generate the database file:

```
#!/usr/bin/ksh
# Shell script name - gen_db

echo "Pittsburgh\0          Pirates\0             " > bball.db
echo "Boston\0              Red Sox\0             " >> bball.db
echo "Chicago\0             Cubs\0                " >> bball.db
echo "Atlanta\0             Braves\0              " >> bball.db
echo "Los Angeles\0         Dogders\0             " >> bball.db
echo "Toronto\0             Blue Jays\0           " >> bball.db
echo "Houston\0              Astros\0              " >> bball.db
echo "END\0                  No Team\0             " >> bball.db
```

```
/* Program name - bbcli_final.c */

/* This program asks the user for a city, then sends
a message across a message queue to a server.  The
server looks the city up in the bball.db database and
returns the name of the baseball team in that city.   */

#include <sys/mode.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include "bball.h"

main()
{
   struct msgout { int mtype; /* Outgoing message */
                   int mpid;
                  char mtext[20];} mymsgout;

   struct msgin  { int mtype; /* Incoming message */
                  char mtext[20];} mymsgin;

   char response[20];
   key_t mykey;
   int msgid;

   mykey=7654321L;

   if ((msgid=msgget(mykey,0))<0) {
      printf("Message queue is %d.\n",msgid);
      perror("Could not get message queue...");
      perror("Check to see if the server is running...");
      exit(1);
   }

/* CONTINUED ON NEXT SCREEN.............*/
/* bbcli_final.c CONTINUED */

   while (1) {
      printf("\nEnter a city:  ");
      gets(response);
      if (!(strcmp("quit",response))) {
          printf("Exiting...\n");
          exit(0);
      }

     mymsgout.mpid=getpid();
     strcpy(mymsgout.mtext,response);

     mymsgout.mtype=1;
     if ((msgsnd(msgid,&mymsgout,sizeof(mymsgout),0)) < 0 ) {
         perror("Could not send message...");
         exit(1);
      }

     printf("\nWaiting for server to answer...\n");
     if((msgrcv(msgid,&mymsgin,sizeof(mymsgin),getpid(),0))<=0)
     {
         perror("Could not receive message..."); exit(1);
     }

     printf("\nThe team name is:  %s\n",mymsgin.mtext);

   }    /*  closes while loop   */

}   /*  End of Program    */

$ cc bbcli_final.c -o bbcli_final
```

```
/* Program name - bbsvr_final.c   */

/*  This program takes requests for database lookups
    from a client via message queues. The server
    creates the message queue. Ther server looks
    up the city in the bball.db file and returns the
    discovered team name to the client via the message
    queue. The outgoing message type is equal to the
    client's PID, which the client sent in the request.
    The structure of the client's request message is:

                 int mpid;
                 char city[20];

  The bball.db record structure is defined in bball.h */


#include <stdio.h>
#include <sys/mode.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/file.h>
#include "bball.h"

main()
{
   struct msgin {   int mtype; /* Incoming message */
                    int mpid;
                    char mtext[20];} mymsgin;

   struct msgout {   int mtype; /* Outgoing message */
                     char mtext[20];} mymsgout;

 /* CONTINUED ON NEXT SCREEN............*/
/* bbsvr_final.c - CONTINUED */

   struct bbrec mybbrec;
   key_t mykey;
   int msgid;
   int fd;
   int i,n;
   char city[20];

   mykey=7654321L;

   if ((msgid=msgget(mykey,IPC_CREAT|S_IRUSR|S_IWUSR))<0)
   {
      printf("Message queue ID is %d.\n",msgid);
      perror("Could not create message queue...");
      exit(1);
   }

   printf("Message queue ID is %d.\n",msgid);

   if ((fd=open("./bball.db",O_RDONLY,0)) < 0 ) {
      perror("Could not open data base file...");
      exit(1);
   }
   while (1) {
      printf("Waiting for requests...\n");
      if (msgrcv(msgid,&mymsgin,256,1,0) <= 0) {
         perror("Could not receive messages...");
         exit(1);
      }

 /* CONTINUED ON NEXT SCREEN............*/
/* bbsvr_final.c - CONTINUED */


          strcpy(city,mymsgin.mtext);
          mymsgout.mtype=mymsgin.mpid;

          printf("Lookup for city:  %s\n",city);

          lseek(fd,0,SEEK_SET);
          while(read(fd,&mybbrec,sizeof(mybbrec))) {
             printf("Found city: %s   ---   team:  %s\n",
               mybbrec.city,mybbrec.team);
             if (!strcmp(mybbrec.city,city) ||
              !strcmp(mybbrec.city,"END")) {
                 strcpy(mymsgout.mtext,mybbrec.team);
                 printf("Returns: %s\n",mymsgout.mtext);
                 if
          (msgsnd(msgid,&mymsgout,sizeof(mymsgout),IPC_NOWAIT)<0)
                  {
                      perror("Could not send team name...");
                      exit(1);
                   }
                   printf("Message sent.\n");
                   break;
             } /* end of if */
          } /* end of while */
       }    /*  Closes while loop    */
   } /*    End of Program    */

   $ cc bbsvr_final.c -o bbsvr_final
   $ ./bbsvr_final &
   $ ./bbcli_final /* This will ask for an input. Enter a city
   name specified in the database file bball.db */
```

__ 6. **Sockets:**

Either create (or peek at the solution at this point!) two socket programs meant to work on a inter-CPU basis.

- They should use sock-stream to handle the transfer.

The parent process should register itself at port 7000 on your PC. (If there are multiple people on your PC, then add your team ID number to the port number to make yours unique) (Try entering the **hostname** command to find out your CPU, and then **host *your_hostname*** to find out your IP address.)

```
/* Program name - socket_srv.c */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/signal.h>
int sd;

void cleanup(int i)
{
   close(sd);
   exit(0);
}

main()
{
int sessionsd;
int rc, pid, i;
socklen_t clientlen;
char buffer[80];
char str[20];
struct sockaddr_in sockname, client;

signal(SIGUSR1,cleanup);
strcpy(buffer,"This is the message from the server\n");

memset( &sockname, 0, sizeof(sockname));
sockname.sin_family = AF_INET;
sockname.sin_port = htons((u_short )7001);
sockname.sin_addr.s_addr = htonl(INADDR_ANY);

/*obtain a socket descriptor and bind a name to it*/
sd=socket(AF_INET,SOCK_STREAM,0);
if ( sd == -1) {
   perror("server:socket");exit(1);
}
printf("Server socket is %d\n", sd);

rc = bind(sd, (struct sockaddr *)&sockname,sizeof(sockname));
if (rc == -1) {
   perror("server:bind");exit(2);
}

/* CONTINUED ON NEXT SCREEN */
/* socket_srv.c continued */

/*specify the number of connections requests that can be
queued*/
if (listen(sd,3)== -1) {
   perror("sever:listen");exit(3);
}

/*wait for a connection*/
clientlen = sizeof(&client);
for(;;) {
   sessionsd = accept(sd, (struct sockaddr *)&client,
   &clientlen);
   if (sessionsd == -1) {
      perror("server:accept");exit(4);
   }
   /*fork child to perform rest of actions*/

   pid=fork();
   if(pid == -1) {
      perror("server:fork");exit(6);
   }
   if(pid == 0)  {
      /*write a message to the session socket descriptor*/
           printf("Server sending....\n");
           rc = write(sessionsd,buffer,sizeof(buffer));
           if (rc == -1) {
                perror("server:send");exit(5);
           }
      close(sessionsd);
      exit(0);
      } /* end of if */
   } /* end of for */
}  /* end of program */

$ cc socket_srv.c -o socket_srv
```

```
/* socket_cli.c */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
main()
{
int rc;
int sd,sessionsd;
char buffer[80];
struct sockaddr_in toname;
memset(&toname, 0, sizeof(toname));
toname.sin_family = AF_INET;
toname.sin_port = htons((u_short )(7001));
toname.sin_addr.s_addr = inet_addr("10.19.98.12");
/* The IP address must be that on which the server is running */
/* Obtain a socket descriptor*/

sd=socket(AF_INET,SOCK_STREAM,0);
if ( sd == -1) {
   perror("client:socket");exit(1);
}

/*connect with a server process*/

rc = connect(sd,(struct sockaddr *)&toname,sizeof(toname));
if (rc == -1) {
   perror("client:connect");exit(2);
}

/*sleep to allow server to send a message*/

printf("Client ready to read from server...\n");
sleep(2);

/*read a message from server*/

rc = read(sd, buffer, sizeof(buffer));
if (rc == -1) {
   perror("client:read");exit(3);
}
printf("%s",buffer);

close(sd);
}
```

```
$ cc socket_cli.c -o socket_cli
$ ./socket_srv &
[2]     721106
Server socket is 3
$ ./socket_cli
Client ready to read from server...
Server sending....
This is the message from the server
```

END OF LAB

Exercise 9. Writing AIX Daemons

What This Exercise is About

The object of the lab is to write a program that generates a daemon process, then execute a simple application in that process.

What You Should Be Able to Do

At the end of the lab, students should be able to:

- Write a simple C daemon program.

Introduction

The lab exercise requires the students to write a daemon program. First, the daemon initialization process is to be done as discussed in the lecture. After the initialization process, the daemon executes its own code.

Required Materials

- One pSeries (or RS/6000) system with AIX 5L
- C for AIX

Exercise Instructions

__ 1. Create a directory named dmonlab under your home directory.

__ 2. Using the model presented in the lecture, create a simple "daemon launching" program. Name this program dstart and place it in the dmonlab directory.

This program should perform all the tasks necessary to set up a daemon environment, including signal handling and creation of a child process. The child process should then continue the task of creating the daemon environment by starting a new process group, closing all files, changing directories, and so on.

Once the environment is set for the daemon, the program should exec the following separate program module. Name this module mydaemon and place it in the dmonlab directory. Compile the program and name the executable as mydaemon.

```
main()
{
   while (1) {
         system("/usr/bin/date >>/home/teamxx/dmonlab/dtest");
         sleep(10);
   }
}

$ cc mydaemon.c -o mydaemon
```

__ 3. Execute the dstart program. You do not need to run it in the background because it dies shortly after kicking off the child daemon process. You should get your prompt back quickly. To see if the daemon is really running, use the **ps -ef** command. Look for your mydaemon process. If everything is running correctly there will be a "-" symbol in the TTY column of the **ps** output for your daemon process. This means that the daemon has no controlling terminal.

The real test is to log off, then log back on and see if your daemon is still running. Also, check to see that data is being written to the /home/youruserid/dmonlab/dtest file on a regular basis (every ten seconds).

Congratulations, you can now write daemon programs!

END OF LAB

Exercise Instructions With Hints

__ 1. Create a directory named **dmonlab** under your home directory.

```
$ cd $HOME
$ mkdir dmonlab
$ cd dmonlab
```

__ 2. Using the model presented in the lecture, create a simple "daemon launching" program. Name this program dstart and place it in the dmonlab directory.

This program should perform all the tasks necessary to set up a daemon environment, including signal handling and creation of a child process. The child process should then continue the task of creating the daemon environment by starting a new process group, closing all files, changing directories, and so on.

```
/* Daemon starter dstart.c */
#include <stdio.h>
#include <signal.h>
#define NUMFILEDES 2000
main()
{
   int fd;

   signal(SIGTTOU,SIG_IGN);
   signal(SIGTTIN,SIG_IGN);
   signal(SIGTSTP,SIG_IGN);

   if (fork() != 0)   /* Parent  */
   {
      sleep(10);
      exit(0);
   }

   setsid();  /* detach from process group and
   relinquish controlling terminal   */

   for (fd=0; fd<NUMFILEDES; fd++)
      close(fd);

   chdir("/");
   umask(0);

   execl("/home/teamxx/dmonlab/mydaemon","mydaemon",0);
   /* Set to appropriate dir  */
}

$ cc dstart.c -o dstart
```

Once the environment is set for the daemon, the program should exec the following separate program module. Name this module mydaemon and place it in the dmonlab directory. Compile the program and name the executable as mydaemon.

```
main()
{
   while (1) {
     system("/usr/bin/date >>/home/teamxx/dmonlab/dtest");
     sleep(10);
```

```
    }
}

$ cc mydaemon.c -o mydaemon
```

__ 3. Execute the dstart program. You do not need to run it in the background because it dies shortly after kicking off the child daemon process. You should get your prompt back . To see if the daemon is really running, use the **ps -ef** command. Look for your mydaemon process. If everything is running correctly there will be a "-" symbol in the TTY column of the **ps** output for your daemon process. This means that the daemon has no controlling terminal.

The real test is to log off, then log back on and see if your daemon is still running. Also, check to see that data is being written to the /home/teamxx/dmonlab/dtest file on a regular basis (every ten seconds).

Congratulations, you can now write daemon programs!

END OF LAB

Exercise 10. Remote Procedure Call (RPC) Lab

What This Exercise is About

This lab gives the students an opportunity to practice what they have learned about ONC RPC by writing a simple ONC RPC client.

What You Should Be Able to Do

At the end of the lab, students should have:

- An increased familiarity with the ONC RPC Language (RPCL).
- A better understanding of how to write ONC RPC applications in general and ONC RPC clients in particular.

Introduction

The exercises program is an extension of the example program illustrated in unit 12. You would be working on the the client-server program.

Required Materials

- One pSeries (or RS/6000) system with AIX 5L
- C for AIX

Exercise Instructions

This lab consists of writing a client for an ONC RPC application that implements a simple client-server style electronic mail server.

Although you are free to call your client program anything you want, in the following description the client is called mc (short for mail client). The existing server program is in /home/workshop/AU25/exercise/ex10 (you should probably make yourself a copy of the contents of this directory and work with the copy).

Note: This lab exercise refers to a number of directories that should exist on your machine. Ask you instructor where they are if you can not find them.

Your mail client (**mc**) should take command line parameters that instruct it to perform one of the following actions:

- Retrieve the next available e-mail message from the server for this user and print the contents on stdout.
- Send a message to a specified user (the body of the message is read from stdin until either 1 KB has been read or the end-of-file is reached).

User names within this mail system are simply AIX user names as returned by the getlogin() function.

Examples

mc next

prints the next message for the AIX user invoking the command (or an appropriate message if there are no more messages).

mc send username < msg

sends the message contained in the file msg to the AIX user username.

Note that neither of these commands specifies the name of the machine that the server is running on. Use whatever convention you like that allows the client program to know which machine is running the server.

Suggestion: Use the **hostname** command to get the name of your machine and hardcode this name into your client; this will allow you to borrow someone else's machine to see if your client can work with your server remotely. Just to be different (and to illustrate another approach), the solutions that you will find in /home/workshop/AU25/exercise/ex10/ use an environment variable RPCMAIL, which must be set to the name of the server's machine).

You are free to simplify or elaborate on the above description to whatever extent you like. Just keep in mind that the goal is to write a simple ONC RPC client, not to implement the world's greatest electronic mail facility!

The remainder of this exercise will take you through the process of writing the client in a step-by-step fashion. If you are quite comfortable with what was discussed in the lecture, then you might decide to ignore the instructions in the rest of this lab and implement the client without additional guidance. You might even want to try to write the entire application from scratch (this is likely to take more time than has been scheduled for this lab).

If you would like to try something slightly simpler than this lab, get a copy of the solution for the lab from /home/workshop/AU25/exercise/ex10/basic/mc.c and make the changes described by the first Optional Exercise below.

__ 1. Review the protocol description file rpcmail.x. Make sure that you understand the calling sequence for the next and send remote procedures. It might help to look at the implementation of these procedures in funcs.c.

__ 2. Compile the server (have a look at the Makefile for guidance) and start the server running (the Makefile creates a binary called **ms**). The mail server runs continuously, responding to requests from clients. You may have to start and restart it if you change a file that causes the **ms** binary to need to be relinked.

__ 3. Write the client program. Compile and link the client, putting the binary in a file called mc. **Hint:** One common mistake is to forget to pass the pointer to the CLIENT object returned by clnt_create() as the second parameter in each call to a remote procedure.

__ 4. Verify that your application works.

Optional Exercise

__ 1. Modify the application so that the client can ask the server to return a count of the number of messages which are outstanding for a particular user. This will involve adding an additional remote procedure to the rpcmail.x file, implementing the procedure in funcs.c and changing your client program to call the remote procedure upon request.

__ 2. Modify the application so that:

- The send request includes the user name of the sender and a subject line for the message.
- The next request returns the subject line, the name of the sender and the date and time that the message was sent.

If you really want to get fancy (and have a lot of extra time), use a different version number for the version of the protocol that supports this last set of changes and make sure that the same server binary simultaneously supports the old and the new versions of the protocol. If an old client sends a message which is received by a new client then the new client gets an empty subject line and a sender of "<unknown>"(because the old client had no way of providing the subject line or the sender's name). This is not actually that hard to do, although you have to be careful to make sure that you keep the two versions straight.

END OF LAB

Exercise Instructions With Hints

__ 1. Login as any user and change directory to your home. Create a directory by name rpclab. Change directory to rpclab. Copy the files rpcmail.x, Makefile and funcs.c from /home/workshop/AU25/exercise/ex10 to your current directory. Review the protocol description file rpcmail.x. Make sure that you understand the calling sequence for the next and send remote procedures. It might help to look at the implementation of these procedures in funcs.c.

```
$ cd
$ mkdir rpclab
$ cd /home/workshop/AU25/exercise/ex10
$ cp rpcmail.x Makefile funcs.c $HOME/rpclab
$ cd $HOME/rpclab
$ vi rpcmail.x /* Review the protocol description file */
```

__ 2. Compile the server (have a look at the Makefile for guidance) and start the server running (the Makefile creates a binary called ms).

```
$ make ms
```

__ 3. Write the client program.

```
/* The mail client main program - mc.c */
/* This file is in /home/workshop/q1125/solutions/rpclbasic/mc.c */

      #include <stdio.h>
      #include <stdlib.h>
      #include <unistd.h>
      #include <rpc/rpc.h>
      #include "rpcmail.h"

      CLIENT *cl;
      char *username;

      void show_usage()
      {
         fprintf(stderr,"Usage:\n");
         fprintf(stderr,"\tmc send _recipient_ < msg\n");
         fprintf(stderr,"\tmc next\n");
      }

      void donext()
      {
         char **msgp;
         char *msg;

         msgp = next_1(&username, cl);
         msg = *msgp;
         if ( strlen(msg) == 0 ) {
              fprintf(stderr,"no more messages\n");
         } else {
              printf("%s\n",msg);
         }
      }
```

```
void dosend(char *recipient)
{
   struct msg m;
   char **msgp;
   char buffer[MAX_BODY];
   char *msg, *nextp;
   int left;

   m.recipient = recipient;
   m.body = &buffer[0];
   nextp = &buffer[0];
   left = sizeof(buffer)-1;
   while (1) {
      if ( fgets(nextp,left,stdin) == NULL ) {
         if ( nextp == &buffer[0] {
            fprintf(stderr,"no message body - request
      ignored\n");
         } else {
            send_1(&m,cl);
            printf("%d byte message sent to \"%s\"\n",
      strlen(buffer), recipient);
            return;
         }
   }
   left -= strlen(nextp);
   nextp += strlen(nextp);
   }
}
main(int argc, char **argv)
{
   char *servername;

   if ( argc < 2 ) {
      show usage(); exit(1);
   }

   servername = getenv("RPCMAIL");
   if ( servername == NULL ) {
      fprintf(stderr,"Please set RPCMAIL environment variable to
   the name\n");
      fprintf(stderr,"of the machine that the rpcmail server is
   running on. \n");
      exit(1);
   }
```

```
    username = getlogin();
    if ( username == NULL ) {
        fprintf(stderr,"can't get your username - bye!\n");
        exit(1);
    }

    /* Create the client record */
    cl = clnt_create(servername, RPCMAIL, V1, "udp");
    if ( cl == NULL ) {
        /* Can't contact the server */
        clnt_pcreateerror(servername); exit(1);
    }

    if ( strcmp(argv[1],"next") == 0 ) {
        if ( argv[2] != NULL ) {
            fprintf(stderr,"unexpected parameter after
    \"next\"command\n");
            show_usage(); exit(1);
        }
        donext();
    } else if ( strcmp(argv[1],"send") == 0 ) {
        if ( argv[2] == NULL ) {
            fprintf(stderr,"missing recipient after
    \"send\"command\n");
            show_usage(); exit(1);
        }
        if ( argv[3] != NULL ) {
            fprintf(stderr, "unexpected parameter after
    \"send\" command's recipient \"%s\"\n", argv[2]);
            show_usage(); exit(1);
        }
        dosend(argv[2]);
    } else {
        fprintf(stderr,"\"%s\" is an invalid command\n",
    argv[1]);
        show_usage(); exit(1);
    }
    exit(0);
}
```

Compile and link the client, putting the binary in a file called **mc**.

```
$ make mc
```

Hint: One common mistake is to forget to pass the pointer to the CLIENT object returned by **clnt_create()** as the second parameter in each call to a remote procedure.

__ 4. Verify that your application works.

```
As indicated earlier, the RPCMAIL environment variable has to
be set to the IP address of the machine on which the server is
running.
$ export RPCMAIL='hostname'

Create a data file named msgfile with a message in it. This
file will be given as input to the client program.
$ cat msgfile
Hello, this is a message from teamxx

Start the server program as a background process.
$ ./ms &

Start the client program, with the appropriate input.
$ ./mc send root < ./msgfile
36 byte message sent to root

The data in msgfile is sent as a message to root.
Next login as root user, set the RPCMAIL variable to the
hostname of the server, and check if the message is received.
$ ./mc next
Hello, this is a message from teamxx
```

Optional Exercise with Hints

__ 1. Modify the application so that the client can ask the server to return a count of the number of messages which are outstanding for a particular user. This will involve adding an additional remote procedure to the rpcmail.x file, implementing the procedure in funcs.c and changing your client program to call the remote procedure upon request. A solution for this part of the lab is in the /home/workshop/AU25/exercise/ex10/opt1 directory.

__ 2. Modify the application so that:

- The send request includes the user name of the sender and a subject line for the message.
- The next request returns the subject line, the name of the sender and the date and time that the message was sent.

If you really want to get fancy (and have a lot of extra time), use a different version number for the version of the protocol that supports this last set of changes and make sure that the same server binary simultaneously supports the old and the new versions of the protocol. If an old client sends a message that is received by a new client, then the new client gets an empty subject line and a sender of "<unknown>"(because the old client had no way of providing the subject line or the sender's name). This is not actually that hard to do, although you have to be careful to make sure that you keep the two versions straight.

A solution for this entire set of optional exercises is in the /home/workshop/AU25/exercise/ex10/optall directory.

END OF LAB

Exercise 11. Threads: Getting Started

What This Exercise is About

This lab reinforces the most common techniques needed to write basic thread programs.

What You Should Be Able to Do

At the end of the lab, students should be able to:

- Create a simple multithreaded program.
- Be able to receive a return code from a thread.

Required Materials

- A pSeries or RS/6000 system with AIX 5L
- C for AIX compiler

Exercise Instructions

__ 1. Create a simple multithreaded program named multi.c with the following characteristics:

- The main thread should start up a second thread.
- The main thread should pass it a simple character string as an argument.
- The main thread's exit should not cause the second thread to fail.
- The second thread should receive and print out the argument passed from the main thread.
- The second thread's exit should not cause any other thread to fail.

__ 2. Copy the program from Step 1 into exitstatus.c and modify it as follows:

- The second thread should return a return code of 2.
- The main thread should receive that return code and print it out.

END OF LAB

Exercise Instructions With Hints

__ 1. Create a simple multithreaded program named multi.c with the following characteristics:

- The main thread should start up a second thread.
- The main thread should pass it a simple character string as an argument.
- The main thread's exit should not cause the second thread to fail.
- The second thread should receive and print out the argument passed from the main thread.
- The second thread's exit should not cause any other thread to fail.

```
/* Program name - multi.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *  callFunc(void * myarg)
{
   printf("Arg from main is: %s\n", (char *)myarg);
   pthread_exit(NULL);
}

main()
{
   pthread_t callThd;
   char myparm[] = "Main Thread Arg.\n";

   pthread_create(&callThd, NULL, callFunc, (void *)myparm);
   pthread_exit(NULL);
}

$ cc_r multi.c -o multi -lpthreads
$ ./multi
Arg from main is: Main Thread Arg.
```

__ 2. Copy the program from Step 1 into exitstatus.c and modify it as follows:

- The second thread should return a return code of 2.
- The main thread should receive that return code and print it out.

```
/* Program name - exitstatus.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *  callFunc(void * myarg)
{
   printf("Arg from main is: %s\n", (char *)myarg);
   pthread_exit((void *) 2);
}

main()
{
   pthread_t callThd;
   pthread_attr_t attr;
   char myparm[] = "Main Thread Arg.\n";
   int status;

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_UNDETACHED);
   pthread_create(&callThd, &attr, callFunc, (void *)myparm);

   pthread_join(callThd,(void * *) &status);
   printf("status from callThd is %d\n", status);

   pthread_exit(NULL);
}

$ cc_r exitstatus.c -o exitstatus -lpthreads
$ ./exitstatus
Arg from main is: Main Thread Arg.

status from callThd is 2
```

END OF LAB

Exercise 12. Threads: Worrying About Your Neighbor

What This Exercise is About

This lab reinforces the most common techniques needed to write basic thread programs.

What You Should Be Able to Do

At the end of the lab, students should be able to:

- Lock global variables using mutexes.
- Set and wait on a condition using condition variables.
- Handle cancellation of a thread.

Required Materials

- A pSeries or RS/6000 system with AIX 5L
- C for AIX compiler

Exercise Instructions

__ 1. Modify the program from Step 2 of Exercise 11 to use mutexes to guard a global variable "count" as follows:

- The main thread should both initialize and destroy the mutex.
- The main thread should try to increment the value of the global variable "count" using the mutex locking technique.
- The second thread should try to increment the value of the global variable "count" using the mutex locking technique.

__ 2. Modify the program from Step 1 to use condition variables to both set and wait on a condition as follows:

- The main thread should both initialize and destroy the condition variable and corresponding mutex.
- Initialize global variable "Ready" to 0. Use it as the boolean variable.
- The main thread should be set the condition. Have it sleep for three seconds after it creates the second thread before it sets the "Ready" variable to 1 and broadcasts the condition to all other threads.
- The second thread should wait until "Ready = 1". Use the pthread_cond_wait call if the boolean is not ready.
- The second thread should indicate to the user when it is able to continue on.

__ 3. Modify the program from Step 1 of Exercise 11 to try out cancellation procedures as follows:

- The main thread should attempt to cancel the second thread.
- The second thread should use the default enabled/deferred settings.
- The second thread should set up a cancellation point by using the pthread_testcancel call. The goal is to have cancellation occur at this point.
- The second thread should indicate, through printf statements, where the cancellation occurs. (that is, a printf after the pthread_testcancel should not appear on your screen).
- Watch out for timing on this one. Remember the parent thread can easily cancel the second thread if it has just begun, or is in a sleep type of condition.

END OF LAB

Exercise Instructions With Hints

__ 1. Modify the program from Step 2 of Exercise 11 to use mutexes to guard a global variable "count" as follows:

- The main thread should both initialize and destroy the mutex.
- The main thread should try to increment the value of the global variable "count" using the mutex locking technique.
- The second thread should try to increment the value of the global variable "count" using the mutex locking technique.

```
/* lockdata.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int count=0;
pthread_mutex_t m;

void *  callFunc(void * myarg)
{
   printf("Arg from main is: %s\n", (char *)myarg);

   pthread_mutex_lock(&m);
   count++;
   printf("callThd says count is %d\n", count);
   pthread_mutex_unlock( &m);

   pthread_exit((void *) 2);
}

main()
{
   pthread_t callThd;
   pthread_attr_t attr;
   char myparm[] = "Main Thread Arg.\n";
   int status;

   pthread_mutex_init(&m,NULL);
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_UNDETACHED);
   pthread_create(&callThd, &attr, callFunc, (void *)myparm);

   pthread_mutex_lock(&m);
   count++;
   printf("Parent says count is %d\n", count);
   pthread_mutex_unlock(&m);

   pthread_join(callThd,(void * *) &status);
   printf("status from callThd is %d\n", status);

   pthread_mutex_destroy(&m);
   pthread_exit(NULL);
}
```

```
$ cc_r lockdata.c -o lockdata -lpthreads
$ ./lockdata
Parent says count is 1
Arg from main is: Main Thread Arg.

callThd says count is 2
status from callThd is 2
```

__ 2. Modify the program from Step 1 to use condition variables to both set and wait on a condition as follows:

- The main thread should both initialize and destroy the condition variable and corresponding mutex.
- Initialize global variable "Ready" to 0. Use it as the boolean variable.
- The main thread should be set the condition. Have it sleep for three seconds after it creates the second thread before it sets the "Ready" variable to 1 and broadcasts the condition to all other threads.
- The second thread should wait until "Ready = 1". Use the pthread_cond_wait call if the boolean is not ready.
- The second thread should indicate to the user when it is able to continue on.

```
/* condvar.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int Ready=0;
pthread_cond_t cv;
pthread_mutex_t m;

void *  callFunc(void * myarg)
{
   printf("Arg from main is: %s\n", (char *)myarg);

   pthread_mutex_lock(&m);
   while(Ready != 1)
       pthread_cond_wait(&cv, &m);
   pthread_mutex_unlock(&m);
   printf("second thread is continuing on....\n");

   pthread_exit((void *) 2);
}

main()
{
   pthread_t callThd;
   pthread_attr_t attr;
   char myparm[] = "Main Thread Arg.\n";
   int status;

   pthread_mutex_init(&m, NULL);
   pthread_cond_init(&cv, NULL);

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_UNDETACHED);
   pthread_create(&callThd, &attr, callFunc, (void *)myparm);

   sleep(3); /* simulate work */

   pthread_mutex_lock(&m);
   pthread_cond_broadcast(&cv);
   Ready=1;
   pthread_mutex_unlock(&m);

   pthread_join(callThd,(void * *) &status);
   printf("status from callThd is %d\n", status);

   pthread_mutex_destroy(&m);
   pthread_cond_destroy(&cv);

   pthread_exit(NULL);
}
```

```
$ cc_r condvar.c -o condvar -lpthreads
$ ./condvar
Arg from main is: Main Thread Arg.

second thread is continuing on....
status from callThd is 2
```

__ 3. Modify the program from Step 1 of Exercise 11 to try out cancellation procedures as follows:

- The main thread should attempt to cancel the second thread.
- The second thread should use the default enabled/deferred settings.
- The second thread should set up a cancellation point by using the pthread_testcancel call. The goal is to have cancellation occur at this point.
- The second thread should indicate, through printf statements, where the cancellation occurs. (that is, a printf after the pthread_testcancel should not appear on your screen).
- Watch out for timing on this one. Remember the parent thread can easily cancel the second thread if it has just begun, or is in a sleep type of condition.

```
/* cancelpt.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *  callFunc(void * myarg)
{
   int i;

   printf("Arg from main is: %s\n", (char *)myarg);
   for(i=0;i<1000000;i++);

   pthread_testcancel();
   printf("I should never reach this statement.\n");

   pthread_exit(NULL);
}

main()
{
   pthread_t callThd;
   char myparm[] = "Main Thread Arg.\n";

   pthread_create(&callThd, NULL, callFunc, (void *)myparm);

   pthread_cancel(callThd);
   pthread_exit(NULL);
}

$ cc_r cancelpt.c -o cancelpt -lpthreads
$ ./cancelpt
Arg from main is: Main Thread Arg.

Before testcancel
```

END OF LAB