DeVRY UNIVERSITY

Course Supplement **Fall 2006**

Course Title:	Operating **Systems and Interfacing With Lab**
Course number:	CET421
Credit/Contact hour:	4-2-5
Course Dependency:	**Prerequisite:** CET375 **Required Co-requisite**: (none)
Instructor:	
FTP Site:	`ftp://n0gsg.no-ip.org`
	Note: Examples developed in class can be accessed here.
	Tom Wheeler twheeler@kc.devry.edu 816.941.0430 x5211

Description

See the CET421 syllabus for the description of this course. This document serves to document the laboratory exercises that are a portion of this course, as well as the policies that will be applied to the laboratory work.

Required Text: **Win32 System Services 3/e, Brain & Reeves; Prentice Hall**

Schedule of Experiments

Number	Description	Due Week #
1	**Win32 File Services**	**3**
2	**Registry APIs**	**5**
3	**Multithreaded Applications**	**7**
4	**Synchronization: Events**	**9**
5	**Deadlock, Mutexes, and the Dining Philosopher's Problem**	**11**
6	**Interprocess Communications by File Mapping**	**13**
7	**Win32 Services**	**14**

Laboratory Procedures

Students in the laboratory portion of CET421 will work individually. Reports must consist of the following parts (please pay attention to order):

All portions of the lab reports for CET421 (with the exception of raw data, which may be included in an appendix at the writer's discretion) must be created electronically. **No hand written work is acceptable**. Use the equation editor in Word (Insert -> Object -> Microsoft Equation 3.0) to type equations and formulas. Captured waveform data must be contained within the document file.

REPORT CONTENTS

1) COVER PAGE contains:

a) Your name
b) Your class and section (CET421 8DA)
c) Experiment Title
d) For: SR. PROFESSOR WHEELER
e) Due Date of report (Week # or date given in class)
f) Operational sign-off blank
g) Final sign-off blank.

2) INTRODUCTION - This will be a written explanation of what the experiment is designed to do (and what is expected as a learning outcome).

3) THEORY OF OPERATION – This will be a walk-through of the code written to complete the requirements of the project. Do not document each line of code! Instead, document each major activity your program takes as it completes its task.

Important: Remember that *you* are to be the author of all laboratory work; refer to the DeVry Academic Integrity policy in the course syllabus, and the student handbook. The laboratory reports are to be written by *you*. Do not just copy or paraphrase the text from the laboratory instructions, or laboratory example. Do not share your writing with other students. Do not "loan" your writing to a fellow student to "help" them write a report. The written analysis must reflect *your* ideas and interpretations.

4) PROGRAM LISTING – This will be a complete listing of the program. If you're writing MFC code, you can omit the code for your main window class (unless it is critical to understanding how the program works). All code must be properly commented: All methods/functions must have a comment header giving the method name, purpose, parameter list, return value, and a description of each parameter and return item value.

4) CONCLUSION – A written explanation of what can be concluded from the activities of the experiment. It must be based on the data collected, and it may also comment on topics such as the efficiency/effectiveness of various software methods used to achieve the experimental goals.

See the document "labex" (on the CET421 web page) for an example of expected writing practices.

Report Grading Criteria

The documentation for CET421 reports is evaluated by the following criteria:

- Mechanics - Correctness of spelling, punctuation, and grammar.
- Organization - Presentation of ideas in a logical order.
- Clarity - Minimization of the reader's workload.
- Appearance - Neatness and visual appeal of the work.

Reports are evaluated holistically. In this method, individual points are not added or subtracted to obtain a score. Instead, the report is compared to these criterion:

The A paper consists of the following:

Central Idea:
- Is clearly expressed, responds to the assignment, provides focus.
- Is explicitly and logically supported with concrete details and examples.

Structure:
- A plan of organization is given in which ideas are arranged in a clear, logical order.
- Ideas are clearly connected.

Development:
- Generalizations are supported or explained with concrete details.
- Smooth transitions are used between sentences and paragraphs.

Style:
- Varied sentence length and structure.
- Consistent and appropriate tone.

Mechanics:
- Grammar, punctuation, capitalization, spelling are correct.

A B paper consists of the following:

Central Idea:
- Is clearly expressed, responds to the assignment, provides focus.
- Is explicitly and logically supported with concrete details and examples.

Development:
- Concrete details usually given to support ideas.
- Transitions are given in most instances where needed.

Style:
- Contains some variation of sentence length and structure.
- Tone is consistent throughout.

Mechanics:
- No more than eight mechanical errors.

A C paper consists of the following:

Central Idea:

- May be slightly askew, but seems to be somewhat clear.

Structure:

- A clear construction is attempted, but does not measure up consistently, and ideas are usually connected via transitions.

Development:

- Writer has attempted to give enough information to support his/her ideas, but there are "holes" where the reader may be uncertain.

Style:

- The writer has attempted a few times to vary sentence length and structure, and tone shifts often.

Mechanics:

- No more than 10 mechanical errors.

A D paper consists of the following:

Central Idea:

- Is somewhat unclear, but is stated.

Structure:

- The ideas are somewhat "rambling" in nature, and few transitions are given.

Development:

- Many ideas have little concrete information for their support. Thus, they often fade into mere opinion rather than rather than expressing "facts."
- Few transitions are given.

Style:

- Leaves the reader feeling unsure of the writer's own attitude toward the topic.

Mechanics:

- Has more than 15 grammar, punctuation, spelling errors.

An F paper consists of the following:

Central idea is missing, and writing wanders from topic to topic without a clear focus.

Structure -- no clear structure -- becomes a jumble of ideas without a stated reason given for why it was written.

Development -- very little development or support given for any discernable ideas.

Mechanics -- writer evidences very little basic understanding of grammar, punctuation, or spelling skills. Many errors of each kind.

Experiment Descriptions:

Lab 1: Win32 File Services

This experiment is designed to familiarize you with Win32 file services. Proceed as follows:

1. Download the project files from `ftp://n0gsg.no-ip.org/CET421/Fall2005/SerializationTest/` into a folder on your computer. (Don't worry about copying the `Debug` folder.)

2. Read Chapter 2 of the Brain & Reevse text. It covers the Win32 file services you need.

3. Convert the application to Win32 file services. Tip: You need to use `CreateFile()` for both reading and writing files. Do *not* use `OpenFile()`; this method is deprecated!

   TIP: Don't forget to include `<windows.h>` where needed!

4. If you have time, enhance the application (provide file integrity checking, operational menu, and so on).

Lab 2: Registry APIs

This experiment will demonstrate how to manipulate system registry from software. Your program will create a shell association for a fictitious content-type called "wheelerjunk" contained in files with the extension "999."

1. Read the handout and using Regedit, study the structures found under HKEY_CLASSES_ROOT to gain understanding. (For example, look at how files with the extension "txt" are mapped to the *notepad* application).

2. Using the system registry APIs, write a program called "associate.exe" that will associate an application located at c:\mine.exe with the file extension "999."

3. Create a *console* application called "mine.exe" that types a text file to the screen. The filename is passed on the command line, as in:

   mine test.999

   TIP: The parameters passed to main, argc and argv[], contain the count of command line items and a pointer to each command line item.

   Store this application at the *root* folder of drive C: so that it will be consistent with the registry entries created in step 2.

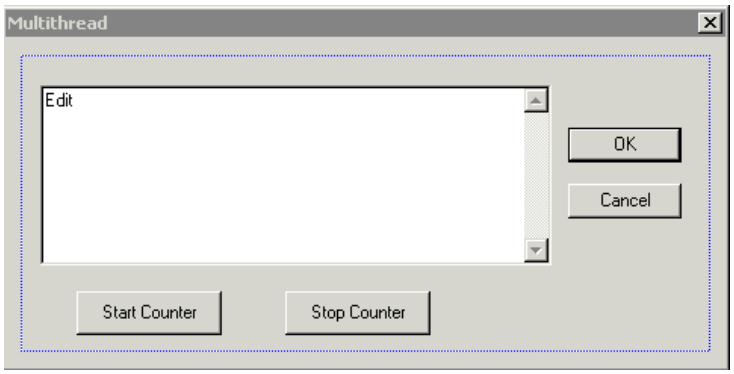
4. Create a test text file called "test.999" (using notepad); save this file to disk (NOT the desktop, however). Then run "associate.exe" to set up the registry. Double-clicking "test.999" from *Explorer* should cause the program "mine.exe" to run and display the text data.

TIP: Before writing a single line of code, use *Regedit* to manually manipulate the system registry and create the desired actions.

Lab 3: Multithreaded Applications

This experiment will demonstrate the power of multithreaded applications. It will perform counting as a background task while allowing other activities to take place.

1. Using Visual C (Studio or .NET), create a dialog application that looks like this:



   NOTE: Be sure you set the edit window to multiline, vertical scroll style.

2. Add threaded code that will cause a count to be accumulated in the edit window once per second once the Start Counter button has been pressed. When the Stop Counter button is pressed, the subthread is to terminate, again leaving the program in an idle state.

   Note: For the subthread to trigger an event in the dialog, it will need a pointer variable passed to the dialog object. Pass this as the parameter to the `AfxBeginThread()` method.

   Tip: Your thread procedure *must* have the prototype:

   UINT mythreadproc(LPVOID lpOwner);

   The *name* of the thread procedure is unimportant, however, the parameter types must match.

3. Include logic to prevent a mishap in case the Start Counter button is depressed multiple times.

4. If you have time, increase the subthread functionality. For example, you might have it interpret the edit box contents as a list of numbers to be summed (one per line), with the subthread appending the sum at the bottom of the box (this will result in an interesting display of "magic" numbers.)

Lab 4: Events

Events are often used to synchronize the operation of two (or more) processes. In this experiment you'll create an MFC-based program that is synchronized to the timer event in project #3. Two different processes will be synchronized using an event object. One process (your lab project #3) will be the source of the event triggers, and the second process will look for the trigger events to take place, time them, and report them.

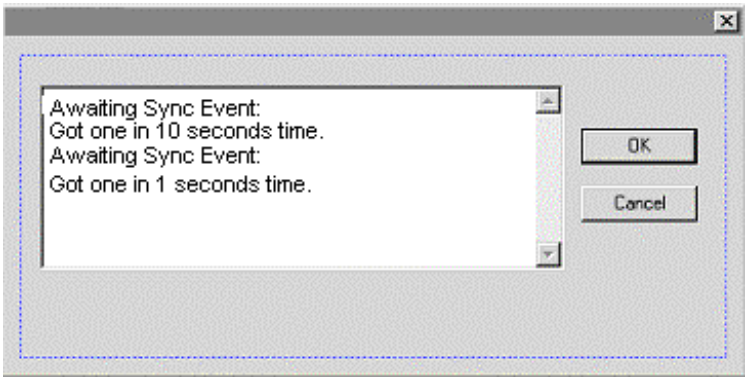
Modification needed for project #3:

Modify project #3 to create an Event object called "metclick" using the API method `CreateEvent()`. Be sure to give the event a name so that the second process can open it! When each count is accumulated in the window, use a Windows API call to set this event to a signaled state.

Program behavior:

The program will display a dialog box much like the one shown below. It will then start a new subthread that waits for the event "metclick" to go into the signaled state.

Use the `OpenEvent()` API call to open the "metclick" event. If the event doesn't exist, report the condition and give the user the option of continuing (again trying to open the event) or aborting (terminating the dialog). When the event becomes signaled, the subthread will cause the main dialog to report how long it took for the event to occur. The subthread will then wait for the next "metclick" event to take place.



The subthread must exit cleanly when the dialog is dismissed (use a completion flag variable as you did with project 3), and it must *not* wait indefinitely for the event to fire. It is suggested to use a waiting timeout period of no more than one second. Override the `OnDestroy()` method of `CDialog` to notify the thread when it should terminate.

In order to accumulate the time interval, you may use API calls to learn the system time, or you may simply accumulate the sum of timeout interval periods.

You've seen how a mutex can be used as a monitor for a data object. This monitor can be used to control access to the object by any number of threads. What happens if more than one thread attempts to enter a mutex-controlled monitor? You know that only one thread may possess a monitor at any time, so the other threads must wait for the active thread to exit the monitor. Of course, the waiting threads don't get any work done during this time!

What if a waiting thread *never* acquires the monitor? This thread will appear to be frozen, or *deadlocked*. This is a very common problem in multithreaded applications. In this experiment you will experiment with the existing code in the textbook that demonstrates the classical "Dining Philosophers Problem."

The objectives of this experiment are twofold:

- You will directly observe the effect of deadlock on a multithreaded program.
- You will implement, compare, and *document* at least two different solutions to the deadlock. The comparison will be quantitative.

Suggested Procedure:

1. Read Section 7.3 of the textbook (starting on page 193) to learn the background information. In particular, study Listings 7.10 and 7.11.

2. Enter and debug the program of Listing 7.10. Once you've got it running, set "numPhilosophers" to ten (10). **Comment this listing completely**.

3. Document the performance of this code. A good quantitative measure of "how good" the program is working is to calculate the average percentage of philosophers that get to eat at any particular time. (This directly correlates to throughput for each thread). You may find another measure that's even better; if so, use it!

   TIP: One excellent way of documenting performance is to write all the screen data to a file, then create a short C++ program to analyze the file content.

4. Apply the modifications of Listing 7.11. Use the same measure as in Step 3 to measure the performance. How much improvement was attained? **Show all calculations, formulas, and results clearly in your report**.

5. Try implementing another deadlock-breaking method (search the web for background). **Again, document the performance attained with this method, showing all calculations, formulas, and results**.

Report Tips:

1. Include a thorough discussion of how the program works. Discuss all modifications made to the software.

2. Make sure to include the quantitative measurements you've made.

3. After doing all of this, don't forget to draw conclusions about what you've done and observed!

Sometimes we need to send data between multiple threads and applications. You've seen that it's quite easy to communicate between threads in a single application; because Windows uses the apartment model for threading, the main process has access to all the memory allocated to each subthread. However, what if two different applications need to communicate? Windows provides several mechanisms:

- File Mapping
- Mailslots
- Named Pipes
- TCP/IP Communications

File mapping is the easiest of these to achieve. The concept is simple: A disk file (or virtual disk file) is mapped into a character array by the operating system. For a process to access the file, it merely asks the operating system to map the array to the disk file; thereafter, whenever the process writes to the array, the operating system automatically updates the file and any other mapped memory spaces in the system.

In this experiment you'll use file mapping to communicate between two processes. You'll create two different programs to do this.

The first program will be the *sender* of the data. It will ask the user to enter a message one line at a time. Each line of the message will be transmitted to any waiting application through a mapped file of your choice. The sender will need to fire an event to let the receiver know that the mapped file contents need to be read.

The second program will be the *receiver* of the data. It will merely wait for the event fired from the first program; when this event is received, it will read and display the data using the mapped file.

The second program should exit cleanly when the text "Bye" is typed into the first. It will also exit cleanly when no text has been received for 30 seconds. Use the timeout feature of the event system to accurately time the 30 second interval.

These programs can be built as either MFC GUIs or command-line applications.

Lab 7: Services

In this experiment you will build a simple Win32 service. There is no report required for this lab!

Your program will perform the following actions:

- It will sleep for ten seconds (use ten one-second `Sleep()` calls to make sure that the thread can exit cleanly in a reasonable time period.)
- Upon awakening, it will beep once, and transmit a UDP datagram to the flood broadcast address "255.255.255.255" on the LAN. This datagram will contain your full name in ASCII text, terminated with a zero '\0' byte.
- It will then repeat the process above indefinitely.

Your program *must* implement all the normal commands expected of the `Handler()` function for a Win32 service, and *must* exit cleanly when the service is stopped from the services control panel.

Sign Off

The instructor will give you a sign off for the lab when your broadcast packet is detected on the LAN. At this point, you will demonstrate that your service responds correctly to all commands from the services control panel.

Design Tips

- Be sure to link to "`ws2_32.lib`" and include <`winsock2.h`> in your project.
- Use the skeleton service that we developed in class as the foundation for your service. You can find this code (ServiceDemo.cpp) at the FTP site for the course (ftp://n0gsg.no-ip.org/cet421).
- Use a packet sniffer such as *Ethereal* (www.ethereal.com) if you want to verify emission of data onto a network.