

Operating Systems exercise – week 6

Deadlock 2.

1. Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is free from deadlock.
2. One method of deadlock prevention by denying the ‘wait for’ condition requires that processes must request all of the resources they will need before the system will allow them to compete for resources. The system grants resources on an ‘all or none’ basis. Discuss the advantages and disadvantages of this method.
3. What are the advantages and disadvantages of imposing a total ordering on the resources when attempting to prevent deadlock by denying the fourth necessary condition.
4. What is a safe state?

5. Consider the following snapshot of a system:

	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker’s algorithm:

- a) What is the content of the matrix Need?
- b) Is the system in a safe state?
- c) If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

6. Consider the following resource-allocation policy. Requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked, waiting for resources. If they have the desired resources, then these resources are taken away from them and are given to the requesting process. The vector of resources for which the waiting process is waiting is increased to include the resources that were taken away.

For example, consider a system with three resource types and the vector Available initialized to (4,2,2). If process P0 asks for (2,2,1), it gets them. If P1 asks for (1,0,1), it gets them. Then, if P0 asks for (0,0,1), it is blocked (resource not available). If P2 now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to P0 (since P0 is blocked). P0’s Allocation vector goes down to (1,2,1), and its Need vector goes up to (1,0,1)

- a) Can deadlock occur? If so, give an example. If not, which necessary condition cannot occur?
- b) Can indefinite blocking occur?

Simple Windows system programming.

1) Start the Windows virtual machine [Start, then VMware, then VMware Player, and select Windows XP virtual machine]

Remember to press CTRL+G to direct input to the virtual machine and CTRL+ALT to direct it to the host computer.

2) We are going to do some simple Windows system programming. In the virtual machine version of Windows there is a development environment for writing C/C++ programs called Dev C/C++.

So start Dev C/C++, go to Start and you will find an icon for Dev C/C++ in the most recently used list of programs. Click on it.

3) This will bring up Dev C/C++. Close the Tip of the Day. You will now see the interface for Dev C/C++.

4) Click on File menu then New and then Source File. This will then bring up a text editor screen where previously it was greyed out. You can enter program code in that text area. Entry is normal and the editor responds to cursor keys and the mouse as you would expect.

5) Enter the following program code (over page to keep it on one page to make it easier to copy). The program is a simple file copy program that uses Windows system calls to do the file copying.

There will be a lot of code that you do not understand, but we will go through it later.

```

/* Basic cp file copy program. Win32 Implementation. */
/* cp file1 file2: Copy file1 to file2. */

#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 256

int main (int argc, LPTSTR argv [])
{
    HANDLE hIn, hOut;
    DWORD nIn, nOut;
    CHAR Buffer [BUF_SIZE];
    if (argc != 3) {
        printf ("Usage: cp file1 file2\n");
        return 1;
    }
    hIn = CreateFile (argv [1], GENERIC_READ, FILE_SHARE_READ, NULL,
        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hIn == INVALID_HANDLE_VALUE) {
        printf ("Cannot open input file. Error: %x\n", GetLastError ());
        return 2;
    }

    hOut = CreateFile (argv [2], GENERIC_WRITE, 0, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hOut == INVALID_HANDLE_VALUE) {
        printf ("Cannot open output file. Error: %x\n", GetLastError ());
        return 3;
    }
    while (ReadFile (hIn, Buffer, BUF_SIZE, &nIn, NULL) && nIn > 0) {
        WriteFile (hOut, Buffer, nIn, &nOut, NULL);
        if (nIn != nOut) {
            printf ("Fatal write error: %x\n", GetLastError ());
            return 4;
        }
    }
    CloseHandle (hIn);
    CloseHandle (hOut);
    return 0;
}

```

6) Once you have entered the code, save it [File then Save]. When you are prompted for a location My Documents comes up by default. We want to save it to a different location. Go to C:drive, then Documents and Settings and then user1. You must give the program a name. We will call it FileCopy.c and change the Save File as Type to C source files (.c).

Save it.

7) We now want to compile the program we have typed in, so go to menu Execute and select Compile. This will compile the program. A small dialog box will come up showing the progress of compilation. Hopefully if all goes well once it has finished compiling it will say Done in the status field and 0 errors and 0 warnings. If this is the case then you can click on Close and you have successfully compiled the program.

If however, you get errors then the first line of code where an error was identified is highlighted. You will find the error messages in the window at the bottom. The lines show, line number of the error, File the error was found in (not relevant for us as all the compilation is in one file, but useful when building more complex projects) and then the actual error message. The error messages are not always helpful, sometimes just saying things like "Syntax error before...". As with most normal compilers you will need to deal with the first errors first as most of the later error listed will be caused by the earlier errors. So fix the errors in the order they are given. Fixing a few and then re-compiling.

Now we are almost ready to run the program.

8) The program copies a file and thus needs 2 command line parameters (name of file to be copied and name of file that is the copy). So to run it we could either go to the Windows command line (cmd.exe) and enter the command with the parameter names, or we could tell Dev C/C++ what the command line parameters will be and then run it from within Dev C/C++.

We are going to use the command line because it is easier for us later.

9) So start the command line cmd.exe (go to Start then Run and type cmd.exe). This will bring up the command line window and will be in the directory C:\Documents and Settings\user1

If type at the prompt,

```
dir
```

This is the directory listing command. You should find FileCopy.c and FileCopy.exe listed.

10) We need a file to copy. Well we could use FileCopy.c as the file to copy and produce the copy with a name like Test.c. It is up to you.

11) Now you can run the program as

```
FileCopy <file name of file to be copied> <filename of copy>
```

12) You can then check that the file has been copied by using

```
dir
```

To check the contents of the directory. You should find the copy there.

To verify that it is indeed a copy you can list the contents of the file by typing

type <name of copy>

e.g. it would be

type Test.c

if you called your copy Test.c.

13) Now we shall spend a bit of time understanding how the program works and especially the Windows system code. So look at the code and read the following.

1. **<windows.h>** contains all the windows function definitions and data type specifications.
2. **LPTSTR** is a type that is defined in windows.h. It represents a null-terminated string.
3. All Windows objects are identified by variables of type **HANDLE** (as we have seen they are held in the open handle table for a process). **hIn** and **hOut** are variables for the handles we are going to use for the file we are going to read from (**hIn**) and write to (**hOut**).
4. **DWORD** is a type that is defined in windows.h. It represents an unsigned 32 bit integer. The variables **nIn** and **nOut** are variables used to hold the number of bytes that are read in from a file (**nIn**) or written out from a file (**nOut**).
5. **CreateFile** – is a Windows system function that is used to both create and open files.

It returns – a handle to the open file object it has created or opened or **INVALID_HANDLE_VALUE** if open fails.

The arguments are (in order):

- i) **lpName** – pointer to a null-terminated string that is the name of the file
- ii) **dwAccess** – specifies read or write access – uses **GENERIC_READ** for reading, **GENERIC_WRITE** for writing and **GENERIC_READ | GENERIC_WRITE** for read/write
- iii) **dwShareMode** – determines whether file can be simultaneously opened by more than one process – use **FILE_SHARE_READ**
- iv) **lpSecurityAttributes** – points to a security attributes structure – but use **NULL**

v) dwCreate – specifies whether to create a new file or overwrite an existing – use OPEN_EXISTING – this will fail if the file for opening does not exist

vi) dwAttrsAndFlags – specifies file attributes and flags – use FILE_ATTRIBUTE_NORMAL

vii) hTemplateFile – ignore this – just set it to NULL

So in code we have:

```
hIn = CreateFile (argv [1], GENERIC_READ, FILE_SHARE_READ, NULL,  
                OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

And

```
hOut = CreateFile (argv [2], GENERIC_WRITE, 0, NULL,  
                  CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

Which opens the 2 files for reading (name in argv[1]) and writing (name in argv[2]).

6. After each file open the code checks for the error code to make sure the open has been successful.

7. Then we have a while loop in which the code repeatedly reads in bytes up to the maximum size of the buffer. The while condition reads in the bytes and checks the number of bytes read in. Provided the number of bytes is > 0 then the loop continues, but once we come to the end of the file (and number of bytes = 0) the loop ends. The reading of the file uses the **ReadFile** function. It reads from current file pointer position, updating the pointer.

The function returns – BOOL TRUE if read is successful (even if no bytes read because of an attempt to read past the end of the file), FALSE otherwise.

The arguments are (in order):

i) hFile – file handle of file being read

ii) lpBuffer – memory buffer for bytes read (typically an array)

iii) nNumberOfBytesToRead – the name says it all

iv) nNumberOfBytesRead – actual number of bytes read (can be 0 if file pointer is positioned at the end of the file)

v) lpOverlapped – points to a structure you will not use – so use NULL

So in code we have:

```
ReadFile (hIn, Buffer, BUF_SIZE, &nIn, NULL)
```

This inputs from file identified by HANDLE hIn, a maximum of BUF_SIZE bytes into the buffer Buffer, updating the byte count in nIn.

8. Within the loop the **WriteFile** function is used to output the bytes just copied in. WriteFile writes to current file pointer position, updating file pointer.

The function returns – BOOL TRUE if write is successful, FALSE otherwise.

The arguments are (in order):

- i) hFile – file handle of file being written to
- ii) lpBuffer – memory buffer for bytes to write (typically an array)
- iii) nNumberOfBytesToWrite – the name says it all
- iv) nNumberOfBytesWritten – actual number of bytes written
- v) lpOverlapped – points to a structure you will not use – so use NULL

So in code we have:

```
WriteFile (hOut, Buffer, nIn, &nOut, NULL)
```

This outputs to file identified by HANDLE hOut, nIn bytes from the buffer Buffer, updating the byte count in nOut.

9. The number of bytes read in and write out are checked to verify that the reading and writing are working correctly.

10. Finally the code closes the files using the **CloseHandle** function. It returns BOOL TRUE if successful and FALSE if not. The arguments of the function is simply the fileHandle that you got from the call to CreateFile.

So in code we have:

```
CloseHandle (hIn);  
CloseHandle (hOut);
```

14) Now we are in a position to modify the program above. We are going to change it so that now instead of copying a file from one file to another, the code will copy a file but output it to the console window. So it is a little like the DOS Type command.

To do this you need to know about one further function.

GetStdHandle – this returns a HANDLE and takes as an argument nStdHandle a DWORD that is the numerical value of the standard input/output stream. The permissible values are:

STD_INPUT_HANDLE – for keyboard input

STD_OUTPUT_HANDLE – for console output

STD_ERROR_HANDLE – error output (normally the console)

These identify which of the standard I/O values you need to use. You can now modify the code to send the file contents to the console screen.

a) Use WriteFile with the appropriate HANDLE you now get from GetStdHandle.

b) NOTE – YOU NEED TO DELETE (OR COMMENT OUT) THE CODE THAT CLOSES THE hOut handle. This is because it will close the console for output and so you won't get ANY further output to the console output.

c) You will have to change the number of arguments in the if statement that guards the usage prompt as the program will now be invoked with only one parameter, as

FileCopy <file to be copied to screen>.