

CSE451 Winter 2011 Project #1

Out: January 5, 2011

Due: January 12, 2011 (see grading criteria for late policy)

Objectives

The projects in this class involve building and modifying the Microsoft Windows Operating System. This is the first of four expected projects. It is mainly a familiarization exercise leading into the subsequent projects. The principal objectives for this first project are:

1. Brush up on your “C” programming skills
2. Learn to build and debug the Windows kernel.
3. Learn the flavor and nature of some of the NT system calls
4. Learn how to modify the kernel and keep track of system information between calls

Ancillary to all this is that you’ll learn some of the guiding principles used in the design of the Windows Operating System.

Getting Started

You will need to become familiar with two main areas. They are building the Windows kernel and using Virtual PC for testing and debugging. All of the sources and reference materials are located at `o:\cse\courses\cse451\11wi` and accessible from any machine in the lab. There are currently four subdirectories in the 11wi directory

1. **Project1** – The sources for the Windows kernel and sample test program.
2. **Reference Material** – Useful and historical reference material for this class
3. **Tools** – Tools needed for this project
4. **Virtual Machines** – Base virtual hard disk containing the Windows OS.

The first two projects are individual efforts. The last two will be done in groups. Project workspace is available on `o:\unix\projects\instr\CurrentQtr\cse451`. The TA’s will provide details assigning project space.

- Building your own kernel

Here are the steps for making a private copy of the Windows sources and building your private kernel.

1. **Get a private copy:** Copy the entire tree `o:\cse\courses\cse451\11wi\Project1` to a directory on `z:` or your local hard disk. For example from a cmd I would issue the instruction

```
xcopy /DECHKY o:\cse\courses\cse451\11wi\project1 z:\project1
```

2. **Start a wrk window:** Then from a cmd window run `\project1\wrk.cmd`. This will setup the path and aliases you need to build the kernel. You can also setup a shortcut on your desktop that issues the target command

```
%windir%\system32\cmd /K "z:\project1\wrk.cmd"
```

3. **Build the kernel:** In the wrk window cd down into the ntos directory and build the kernel

```
cd \project1\base\ntos  
nmake -nologo %arch%=
```

The resultant kernel is located in `\project1\base\ntos\build\exe` and it is called wrkx86.exe. There are also a symbol file for debugging and a readable memory map.

4. **Build the sample test program:** Then in the wrk window cd to the test directory and build the test program

```
cd \project1\test  
makeit
```

The resultant executable is called test.exe

From steps 3 and 4 you now have two executables that you want transfer over to the virtual hard disk containing your bootable windows system.

- Booting and debugging:

On each lab machine in "`\program files\Microsoft Virtual PC\images`" is a readonly file called ws03esp1.vhd containing a bootable Windows system. You will be creating what is called a Differencing disks that will essentially be the ws03esp1.vhd disk fronted by your private changes.

1. **Create cse451project1.vhd:** Without going into the complete step-by-step details you want to start Microsoft Virtual PC on the local machine and with the Virtual Disk Wizard create a new Virtual Hard Disk called cse451project1.vhd as a differencing disk whose parent is ws03esp1.vhd. On my system my new vhd is

```
%USERPROFILE%\My Documents\My Virtual Machines\cse451project1.vhd
```

2. **Create cse451project1.vmc:** Without going into the complete step-by-step details you need to create a virtual machine using the New Virtual Machine Wizard in Microsoft Virtual PC. Create a new Virtual Machine called `cse451project1.vmc` for Windows Server 2003, and specify `cse451project1.vhd` as your existing virtual hard disk.

3. **Setup COM2 on cse451project1.vmc:** Via the settings button for the virtual machine you need to specify a named pipe for COM2. Specify the named pipe as “`\\.\\pipe\\ws03com2`”
4. **Copy new kernel image to cse451project1.vhd:**
 - a. **Start cse451project1:** The vhd contains two bootable kernels. The default boot kernel is called “wrkx86” and the second choice is “Windows Server 2003 Enterprise”. In Virtual PC start cse451project1 and boot either choice.
 - b. **Drag and drop (i.e., copy) over the new kernel:** After the system fully booted copy the files wrkx86.exe and test.exe from the build machine to the test machine. In the test machine put wrkx86.exe into `c:\windows\system32`. Copy test.exe into the subdirectory `c:\test`.
 - c. **Shutdown the vmc:** The machine needs to be rebooted. Shut it down and go to step 5.
5. **Start cse451project1 and the debugger:** You are now ready to start the virtual machine and start windbg (kernel debugger). Use the start button on the Virtual PC console to start cse451project1 and immediately afterward type “winbag” in the wrk window. The winbag command alias will start windbg with the proper named pipe for debugging.
6. **Setup search paths in the debugger:** After windbg starts you will need to setup the search path for the kernel symbol, source, and image. The symbol and image path are both the same, they are “z:\project1\base\ntos\build\exe” The source path is “z:\project1\base\ntos”

You now should have a running system that you can login to and run the test program.

Your assignment:

Your assignment is to modify the windows kernel and a test program to monitor a small set of system calls (nee APIs). Here is some background information you will need to understand and complete this project. All user programs that need to utilize an NT system service (e.g., opening or writing to a file) enter into the kernel code through what are called “NT” calls. There are hundreds of different NT calls into the system; all begin with the two letter prefix “Nt”. The names of some of the API’s are NtCreateFile, NtOpenFile, NtClose, NtQuerySystemInformation, NtSetTimer, and so forth.

The “C” header files for these system calls are all declared in the directory “public\sdk\inc”. Here is a partial list of the files in the directory with a brief description of its contents.

```
ntdef.h      - basic definitions of common data types used by everyone
ntrtl.h     - runtime library for use by everyone in the kernel
```

```

ntstatus.h - all the possible return status values for system calls
bugcodes.h - system bug check codes
ntexapi.h - executive routines
ntioapi.h - I/O routines
ntkeapi.h - kernel routines
ntlpapi.h - local procedure calls routines
ntmmapi.h - memory management routines
ntobapi.h - object management routines
ntpsapi.h - process/thread handling routines
ntseapi.h - security routines

```

Your assignment is to modify the system to monitor the following APIs: NtCreateFile, NtOpenFile, NtReadFile, and NtWriteFile. You will find the sources for these routines in the directory base\ntos\io\iomgr. We want you to modify the code to collect for each API the number of times they have called since boot time and give a distribution of their return values.

Every Nt API function (that follows the rules) returns an NTSTATUS value. The value indicates success, or an error because of an invalid buffer, invalid request, privilege violation, and so forth. Look at the file ntstatus.h for a complete listing of possible return values. Each NTSTATUS value belongs to one of four categories (Success, Information, Warning, and Error). You only need to report on the classification for each return value. For example, on NtCreateFile you might report that was called 132 times 77 returned a success value, and 33 were errors, and 22 were warnings. You will find a set of macros in public\sdk\inc\ntdef.h that can help accomplish this.

```

//
// Generic test for success on any status value (non-negative numbers
// indicate success).
//
#define NT_SUCCESS(Status) ((NTSTATUS)(Status) >= 0)

//
// Generic test for information on any status value.
//
#define NT_INFORMATION(Status) ((ULONG)(Status) >> 30 == 1)

//
// Generic test for warning on any status value.
//
#define NT_WARNING(Status) ((ULONG)(Status) >> 30 == 2)

//
// Generic test for error on any status value.
//
#define NT_ERROR(Status) ((ULONG)(Status) >> 30 == 3)

```

For each API we want to know how many times it was called and the return values. In addition for NtReadFile and NtWriteFile we want you to keep track of the number of bytes read and written.

If you “grep” the sources for CSE451 you will find three files where we started to expand upon the already existing NtQuerySystemInformation API. This is skeletal code that the user program in the test directory uses to both send and retrieve information from the kernel. Your task is to understand the CSE451 additions to the kernel and the test program. And then modify both to report back the class API statistics. Feel free to add to or subtract from any of the CSE451 labeled code that you find in the sources, they were just place markers to help get you started.

How the test program formats the report is up to you. However the report must be easy to read. As the due date approaches we will provide you with a command script that will exercise these API calls. But here is an example of what your test program output can look like:

	Total						
API	Calls	Success	Info	Warn	Error	Bytes	
-----	-----	-----	-----	-----	-----		
NtCreateFile	132	77	0	33	22		
NtOpenFile	100	60	0	0	40		
NtReadFile	777	555	0	0	222	123,456	
NtWriteFile	44	22	11	0	11	234,456	

Extra Credit:

Extra credit will be awarded for reporting a more detailed listing of return values. Instead of just the broad classifications you will need to keep track of each value. For example NtOpenFile might return an error for trying to open a nonexistent file or a trying to open a directory as a file.

You will need to modify the kernel to keep track of all these return values. The catch is that there are a lot of possible return values (in ntstatus.h there are over 1000 status values) and no one really knows the actual subset of possible return values for any given API. So you will need to modify the code to learn what the returns values are as the system runs. You can assume that no API will have more than 32 different return values. This will let you statically allocate/define memory for storing your data in the kernel. Your test program will then print out the frequency of each individual return value.

Additional extra credit will be given for projects that correctly handle asynchronous I/O statistics. For example, consider the case where a user asks to read 512 bytes and at some later time (i.e., asynchronously) the read complete with only 32 bytes.

Turn-in:

Be prepared to turn in the following

1. Executables images of your test program, and the modified kernel.

2. Source code for your test program and the modules that you have modified and added to your kernel.

You will be submitting the source code and executables to Catalyst. Please separately submit test.exe, wrkx86.exe, and a zip of your source directory (After running 'nmake clean').

Grading Criteria:

Your project will be graded based on the following criteria:

E (0.0) - For turning in nothing and/or doing nothing

D (1.0) - For a project that is dysfunctional. It may not compile or it gives completely bogus answers.

C (2.0) - For a project that is barely functional. Or it might miss in a few cases but generally it is okay. Or you can sort of understand how the code works but it is not clear.

B (3.0) - For a project that is functionally complete. It is clean and does the job but without any real elegance. It is a meat and potatoes type solution. So a "B" simply accomplishes the objectives, but nothing fancy.

A (4.0) - Is for projects that go well beyond just being functional. They are works of art. We can look at an "A" project and it would stand high above the "B" project.

There is nothing wrong with a "B". It demonstrates clear mastery of the subject. Extra Credit that is well done will add a full letter grade to the assignment. There is a not a late penalty on this first assignment, however to be graded it must be turned in before the due date of project #2.

Addendum

In the Windows NT sources the use of "int" was frowned upon. Do you know why? Instead four basic data types formed the fundamental building blocks used in most of the data structures. The name of the four types are CHAR, SHORT, LONG, and LONGLONG; representing signed 8-bit, 16-bit, 32-bit, and 64-bit integers respectively. There are also corresponding unsigned versions UCHAR, USHORT, ULONG, ULONGLONG. And there are pointers to these eight types PCHAR, PCHAR, PSHORT, PUSHORT, PLONG, PULONG, PLONGLONG, PULONGLONG.

If you look in public\sdk\inc\ntdef.h you will find additional types, however these four represent the basic building blocks used in about 90% of the cases. In ntdef.h you will also find some commonly used "structs" for things like counted strings and linked lists, just to name two. This is not meant to be an exhaustive list, but hopefully it will help you get started understanding the code.