Porting Cilk to Windows NT

Rupinder Kataria
Advisor: Volker Strumpen
Yale University

Abstract

Cilk is a language for multithreaded parallel programming based on ANSI C. Cilk has been implemented on many Unix platforms already. The goal of this project is to port Cilk to Windows NT, an operating system with growing popularity. NT’s ease of installation and use, along with the large number of applications available provides an extensive potential user base for Cilk. In addition, this project will contribute to the effort of integrating Unix and NT environments.

1. Introduction

Cilk facilitates multithreaded parallel programming by providing a runtime system that is responsible for scheduling computations in a provably efficient manner. Programmers can focus on exploiting parallelism in their code. Cilk provides the programmer with two basic tools: the compiler (cilk2c) and the runtime system. Cilk2c translates a Cilk program into C code that is then compiled with gcc and linked with the runtime system [1].

The initial port of Cilk to NT was done under Cygwin [6], which provides a library to map a subset of Unix API (Application Programmer’s Interface) on top of the existing Win32 API. Cilk itself is a highly portable language; the few platform specific functionalities are localized to a few files in the runtime system and can be addressed through the use of Cygwin. The API provided by Cygwin allowed the runtime system to be ported to NT without extensive source code changes.

There are two implementations of Cilk available: process and threaded. The process version of Cilk is implemented with each Cilk “worker” as a separate process. Unlike the process version, the threaded implementation of Cilk employs operating system threads to create the Cilk “workers” [1]. Cilk utilizes a shared memory model to communicate between Cilk workers. Whereas the process version has a separate address space for each operating system process that must be shared through mapping, the thread implementation has one global address space that facilitates communication. Of course, the global address space hinders private storage for each Cilk worker, but this can be addressed through the use of private variables, as explained later. Each flavor of Cilk introduces a different set of problems in the porting process as discussed later.

Windows NT 4.0 is the target platform for this port of Cilk-5 (hereafter referred to as Cilk). Although the process version of Cilk is often available first on UNIX systems, problems with the process version on NT led to the threaded version of Cilk to be ported to NT first. NT also offers support for
private storage in each thread, and the lack of this private storage has prevented other ports of Cilk from including support for private variables in the thread implementation.

The rest of this paper is structured as follows. Section 2 provides background information on Cilk that is required in understanding the porting process. Next, section 3 briefly describes the environment under which this project was completed. Some of the challenges in porting Cilk to Windows NT are explained in Section 4. Section 5 summarizes the paper and offers some ideas for further improvements. Finally, a list of files, which were created or modified for this project, along with descriptions is included in the appendix.

2. Cilk Background

The Cilk language is a simple extension of C. The following three keywords are introduced in Cilk to allow parallelism in a C program: cilk, spawn, sync. Using the keyword spawn, a C procedure declared as type cilk can be spawned as a set of procedures executing in parallel. The procedures can be synchronized upon completion by using sync. The shared and private keywords are used as global type qualifiers of variables, where any access to a shared variable refers to the same memory location and private variables are local to each OS thread. Additional keywords in Cilk include abort and inlet [1].

Compilation of a Cilk program is accomplished in three simple stages. The Cilk source is first preprocessed using the C preprocessor. Next, the code is translated to C by using the cilk2c compiler. Finally, this C code is compiled and linked with the runtime system library [1]. This compilation process is modified slightly under NT to ensure that the program compiles successfully. These will be discussed in section 4.

The cilk2c compiler was modified only to allow the use of private variables in the threaded version of Cilk.

The runtime system contains several files that required extensive modifications. A set of architecture-specific files is provided for the existing ports. A file for Windows NT was added for this port, and a number of changes were made to the files provided in the appendix. Most of the early porting process concentrated on changing Unix system calls that were not supported by Cygwin. This required utilizing Win32 API system calls for mutex support, thread creation and synchronization, high-resolution timing, obtaining processor information, and Thread Local Storage (the private storage area provided for every thread in NT).

The process implementation of Cilk employs heavyweight operating-system processes to implement each “worker.” Cilk relies on using shared memory to communicate between Cilk threads. Specifically, Cilk uses the mmap [7] system call of Unix to share memory segments between processes at fixed locations [1].

The mmap system call under the Cygnus runtime system is not implemented correctly when the MAP_FIXED flag is used to specify the exact address (addr) at which the mapping should take place.
Execution of a program using the mmap system call with the MAP_FIXED flag invariably results in an ERROR_ACCESS_DENIED under Windows NT 4.0. Examination of the Cygnus sources did not reveal the problem, and as a result, the process version port of Cilk was canceled.

Native operating system threads are utilized in the creation of Cilk “workers” in the threaded version of Cilk. Although the shared memory model is provided for free through a global shared address space in each OS process, the threaded version of Cilk lacks the ability to use private global variables. These are variables that are global in a program but are private to each OS thread, i.e. each OS thread can set the private global variable to a value that is local to each thread. As a result, when different threads call the same function, and the function uses the global variable, the value it sees for the variable is the one allocated by the current OS thread.

Windows NT does support global private variables with Thread Local Storage (TLS), a private storage area for each thread [4]. The attached code (Threads3.cpp and Threads4.cpp) illustrates the system calls used under NT to utilize the private global variables. The thread local storage is initialized when a thread is created and can thereafter be accessed using the system calls shown in the sample code. The system calls used for accessing TLS are at least a factor of ten slower than accessing the TLS directly and their use will result in a dramatic reduction of the run-time system’s performance. A direct method of accessing this storage area is illustrated in section 4.

3. Porting Environment

This section describes the details of the environment under which the porting was done. Both the software and hardware utilized are listed here.

The target operating system for the port is Windows NT. Version 4.0 (build 1381) of the Workstation with Service Pack 3 was used for this project. NT was configured to run in multi-user mode. A dual processor machine was available for the duration of this project. NT workstation was used for the port because it can utilize up to two processors. Windows NT Server can support up to four processors and the Enterprise edition of Windows NT can scale to as many as eight processors. Although Windows NT can run on up to thirty-two processors, this requires a custom version to be designed by the hardware vendor [5].

Two sets of hardware were used. The initial port began on a single processor Compaq Armada 7800 laptop. This was an Intel Pentium II 266 MHz machine with 96 MB of RAM and a 5.0 Gig hard drive. After the initial port, the work was moved a Dell dual processor 400 MHz Pentium II computer. This machine had 128 MB of RAM and an 8.0 Gig hard drive. A third machine, identical to the second, was sometimes utilized for testing purposes when comparing performance results with the Linux version of Cilk on an Intel processor. The version of Linux used here was Redhat 5.2 with kernel 2.0.36. This was a uniprocessor kernel, so there was no SMP support available for the Linux version while testing.
Cygwin was used to expedite the porting process. The set of tools provided allowed much of the code in the runtime system to remain unchanged as a functional version of Cilk was developed under NT. The Cygwin library allows programmers to write code based on the Unix API and the underlying Win32 API. This allowed many of Unix system calls not implemented by Cygwin to be replaced with corresponding Win32 API calls that worked in conjunction with the existing code of the runtime system. Cygwin also provides a set of Unix programming tools, including gcc, flex, and bison; these were heavily utilized since the cilk2c compiler relies on these tools to compile itself and to aid in the translation process from Cilk to C.

4. Porting to NT

In porting the threaded version of Cilk to NT, there were four main areas of code that needed to be altered or rewritten: timing, locks, thread creation/yielding, and private variables. While the first two were exclusively located in the runtime system, the third required a number of changes to both the cilk2c compiler and the runtime system. A number of Win32 API prototypes that were used are listed here, and a chart containing their corresponding UNIX API prototypes is included in Appendix C.

4.1 Timers

Timing is very important in Cilk as it allows the programmer or user to measure the work and critical path of a given computation. The timers in Cilk permit users to obtain measurements of critical path, work, and parallelism in the code.

The timing measurements require a high-resolution timer, on the order of nanoseconds. The Win32 API provides the following prototype:

```c
VOID GetSystemTimeAsFileTime ( 
    LPFILETIME lpSystemTimeAsFileTime 
);
```

where LPFILETIME is a pointer to the following structure:

```c
typedef struct _FILETIME {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME;
```

This system call obtains a 64-bit value (divided in the structure into two 32-bit values) that represents the number of 100-nanosecond intervals since January 1, 1601[3]. The resulting values can then be packed into one 64-bit value that is used in measuring process time and wall clock time.

4.2 Locks

Since Cilk supports parallel programs working in a shared memory environment, locks are necessary to control access to certain objects. This ensures that only one worker is modifying an object at a given time. In addition to the memory locks that are provided at a low level, the Win32 API contains support for locks through mutexes.

 Mutexes are created by using the following Win32 API prototype:

```c
HANDLE CreateMutex ( 
    LPSECURITY_ATTRIBUTES lpMutexAttributes, 
    BOOL bInitialOwner, 
    LPCTSTR lpName 
);
```
This function will return a handle to a newly created mutex or NULL if an error occurs. The arguments specify security attributes, whether the calling thread will own the mutex, and an optional name for the mutex. Once the mutex is created, it can be waited on by using another Win32 API prototype:

```c
DWORD WaitForSingleObject(
    HANDLE hHandle,
    DWORD dwMilliseconds
);
```

The calling thread can wait on a mutex by specifying the handle to the mutex and the amount of time to wait; if INFINITE is specified for the time, the thread waits until the mutex is obtained. The return value of this function specifies whether the object being waited on was signaled, abandoned, or if the timeout parameter expired before the object was signaled.

A thread that owns the mutex can release it by simply calling:

```c
BOOL ReleaseMutex(
    HANDLE hMutex
);
```

and specifying the handle to the mutex to be released [3].

In conjunction, these three calls can be used to implement the mutex functionality that all flavors of Unix provide.

### 4.3 Thread Functionality

Windows NT contains a rich API for multithreaded programming. The system calls for thread programming that were replaced with the corresponding Win32 API calls were localized in the architecture-specific file for NT.

Cilk normally creates one thread per processor that is utilized for each computation. Threads are created easily under NT with:

```c
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    DWORD dwStackSize  
    LPTHREADSTART_ROUTINE lpStartAddress,  
    LPVOID lpParameter,  
    DWORD dwCreationFlags  
    LPDWORD lpThreadId
);
```

The argument `lpStartAddress` represents the address of the function where the thread will begin execution.

The priority of a running thread can be changed with a combination of the following 3 Win32 API prototypes:

```c
HANDLE GetCurrentThread(VOID);

int GetThreadPriority(  
    HANDLE hThread
);

BOOL SetThreadPriority(  
    HANDLE hThread  
    int nPriority
);
```

The first call will return a handle to the currently executing thread. The second and third calls will get and set the priority of the indicated thread according to a set of predefined constants, ranging from `THREAD_PRIORITY_IDLE` to `THREAD_PRIORITY_TIME_CRITICAL`.

Once a thread has been created, it is sometimes necessary to wait on the thread until it has completed its task(s). Two Win32 API functions offer this functionality. One is the `WaitForSingleObject` function whose
The prototype is listed above, and the other is described below:

\[
\text{DWORD \text{WaitForMultipleObjects}(}
\begin{align*}
&\text{DWORD \text{nCount},} \\
&\text{CONST \text{HANDLE \*lpHandles} } \\
&\text{BOOL \text{fWaitAll},} \\
&\text{DWORD \text{dwMilliseconds}}
\end{align*}
\); \\
\]

This is similar to \text{WaitForSingleObject} except an array of handles is passed to the function along with an integer representing the number of handles and a boolean flag indicating whether to wait for all of the objects indicated or to return as soon as one object is signaled. The return values are similar to the other function as well [3].

The following prototype is useful in relinquishing or yielding the remainder of a thread’s time slice to any other thread of equal priority that is ready to run:

\[
\text{VOID \text{Sleep} (DWORD \text{dwMilliseconds});}
\]

The thread will sleep for the indicated number of milliseconds, or if a value of 0 is specified, the thread relinquishes the remainder of its time slice [3].

Windows NT provides TLS to allow threads to contain thread private variables. When a thread is created, an array of 64 32-bit values are allocated, initialized to 0, and attached to the thread for private storage [4]. This storage are can be accessed with four API functions:

\[
\begin{align*}
&\text{DWORD \text{TlsAlloc} (VOID);} \\
&\text{BOOL \text{TlsSetValue(}} \\
&\begin{align*}
&\text{DWORD \text{dwTlsIndex},} \\
&\text{LPVOID \text{lpTlsValue}}
\end{align*}
\); \\
&\text{BOOL \text{TlsFree(}} \\
&\begin{align*}
&\text{DWORD \text{dwTlsIndex}}
\end{align*}
\);
\]

Although the space for TLS is pre-allocated with the creation of each thread, \text{TlsAlloc} should be used to “reserve” a slot in the array. The next two functions can be used to set and get values in the array, respectively. Finally, the slot in the array is freed when it is no longer needed.

Error checking was eliminated from these system calls in order to make them extremely efficient, but the cost incurred for each system call is still very large, relative to accessing the TLS array directly, when used in a runtime system, such as Cilk’s. The TLS array is located at a fixed offset from the beginning of the Thread Information Block (TIB), also known as the Thread Environment Block (TEB). The TEB stores all information local to a thread in NT [5]. This constant offset, not the one used below, was added to the address of the TEB, which can be obtained by loading it from the FS register on an Intel CPU with a line of inline assembly code:

\[
\begin{align*}
&\text{asm volatile ("movl \%fs:0x18,\%eax;}
\text{movl \%eax,\%0"
}; \\
&\text{"=g" (pTIB));}
\end{align*}
\]

The TEB contains a self-referential pointer at an offset of 0x18. This offset is added to the value of the FS register and copied to the EAX register and then to the variable pTIB. The TLS array can now be written to and read from without incurring the overhead of a system call that can be ten times as slow as directly accessing this array of private storage.
This slowdown is demonstrated in one of the sample programs in the appendix.

4.4 Private Variables

All global variables are shared by default because the OS-threaded version of Cilk operates in the address space of a single OS process. Any accesses to such a variable will refer to the same memory location. A global variable that is declared as private “reserves” a location in each thread, and only the owning thread can modify the value of the variable. Variables that are private to a thread can be beneficial when communicating between a Cilk thread and C functions that are called from the Cilk threads [1].

The implementation of private global variables on NT relies on thread local storage (TLS), the private area pre-allocated for each thread by the operating system. Windows NT is the first operating system to have an OS-thread distribution of Cilk that supports private global variables. The use of private global variables in NT is limited to static use, that is, any private global variable declaration is assumed to be static and can only be used in the file in which it was declared. The implementation described below clarifies this limitation.

Those private global variables that are declared in the Cilk runtime system are not limited by the static rule. They are global to the entire program. This set of variables is pre-declared in the runtime system as a structure called Cilk_private_variables. A pointer to this structure is initialized at the creation of each thread and references the thread local storage area for the running thread.

The global pointer is placed in the EBX register of the Intel x86 processor; thus, each processor has a copy of the pointer and can swap each thread’s copy in and out of the register as context switches occur.

The thread local storage area of each NT thread consists of 64 DWORD slots, where a DWORD is defined to be an unsigned int, which is currently four bytes on the Intel Pentium II platform. This allows a total of 256 bytes to be used for private variables in each Cilk program. Of the 256 bytes, 152 bytes are currently used by the Cilk_private_variables structure that contains all the private variables needed by the runtime system.

The cilk2c compiler only translates one Cilk file at a time and this prevents a global analysis of a Cilk program if more than one Cilk file is used. As a result, references to a global variable from one Cilk file to a private global variable declared in another file cannot be handled cleanly. Ideally, each global private variable would be placed directly in TLS and all references to it thereafter would find this private storage. The low number of general-purpose registers available and the limited size of TLS prevent the use of this approach.

The cilk2c compiler has been modified to collect all references to private variables and place them in a new uniquely named structure. This is done for each Cilk source file containing private global variable declarations. If the private variables are not declared static, the user is warned. The declarations are removed from the Cilk source, and the newly created structure
is placed in a header file to be included in the C file generated by cilk2c.

Any reference to the declared variables is replaced with a macro to access the variable through the structure created by the cilk2c compiler. For example, the global private variable `count` is replaced with the `PVS` macro that contains a randomly generated structure name:

\[ PVS(Private\_vars\_926969742, \text{count}) \]

The definition of the macro is inserted into the generated C source file. The function of the macro is discussed later in this section.

The allocation and initialization of each structure is important so that it references a valid location before any members of the structure are accessed. This is accomplished by generating an initialization function for each Cilk file. The initialization function is called after the creation of each thread in the runtime system. This allows each thread to execute the initialization procedure once per each source file containing references to private variables. Once a thread has initialized a structure, it sets the private variable `isFirst` to "1", a flag that will prevent the thread from executing the initialization more than once per file.

The initialization procedure allocates a block of storage for the global private variable structure local to the current C source file and places a reference to this allocated storage at the next available slot in the thread local storage for the current thread. An offset to the first open slot is calculated by adding an offset to the address at the top of the TLS. This offset is stored as a private variable that is part of the `Cilk_private_vars` structure. The diagram below illustrates this operation:

![Diagram](image)

**Figure 1**

Each file contains a static TLS_offset array that holds an offset into the TLS array of every thread. The offset points to the slot in TLS that contains a pointer to the storage area that holds the structure of private variables for the current file. The gray areas represent allocated storage, with CPV referring to the Cilk_private_variables structure that resides in every thread.

During the initialization, the offset to the currently allocated slot in TLS is also stored in a static array, `TLS_offset[]`, (referenced by the thread number) so that it can be used to find the address of the allocated storage in the future.

Access to the private variables is granted through a macro:

```c
#define PVS(strct, var)\n(/**(strct = (int) Cilk_private_vars +\nTLS_offset[USE_PRIVATE(_Self)])####var)```

The macro expands to a set of two instructions. First, `TLS_offset[]` is accessed to locate the offset to the slot in TLS that contains the needed pointer.
The index into the array is the integer id of the current thread, \_Self, which private to every thread. This offset is added to a pointer, Cilk\_private\_vars, which references the first slot of the TLS array. The resulting sum, indicating the slot in TLS that points to the structure of private variables, is stored in a pointer. This pointer can be dereferenced to access the thread private variables stored in the structure.

### 5. Conclusion and Future Work

The basic Cilk-5 environment has been ported to Windows NT 4.0 under the Cygwin runtime system. The process version of Cilk presented some perplexing issues, but NT’s threading model proved to be a good match for an NT port of Cilk. Sufficiently precise timing measurements are possible through the use of NT’s high-resolution timers (100 ns). Support is also provided for private global variables by utilizing the Thread Local Storage of OS threads in NT. Though the use of private global variables has its limitations, Windows NT is the first operating system to have support for private global variables in the thread distribution of Cilk.

This project represents only the first step to a complete port of Cilk to NT that is fine-tuned for this operating system. The Cygwin dependencies should be removed to allow native operation of Cilk on NT. Initial work on this has begun with a functional version of cilk2c compiled under the Microsoft Visual C++ environment without any Cygwin dependencies. The runtime system presents a number of challenges in achieving this goal, however.

The limitation of private variables in this port is due to the nature of cilk2c. Extending the cilk2c source-to-source translator to perform a global analysis on all input files can resolve this restriction among others.

One important tool that is included with Cilk-5 but not ported to NT is the Nondeterminator. This is a debugging tool that aids in the detection and location of data-race bugs. This tool, described in detail in [1] should also be ported to NT to assist users in writing Cilk code that does not contain data races.
References

   http://supertech.lcs.mit.edu/cilk


   http://sourceware.cygnus.com/cygwin/

Appendix A – Programmer’s guide

This appendix is a reference for programmer’s wishing to build the binaries from the Cilk source or modify the Cilk source. Most of the information in this section is included in the README file that is distributed with the sources.

The source directory contains the following directories/files:

cilk2c/ : source code of the cilk2c
         preprocessor
rts/  : source code of the Cilk runtime
        system
lib/  : source code for some library
       routines
examples/ : some example Cilk code
doc/   : original Cilk-5.2 manual (no
        NT specific info)
Makefile : top-level Makefile
Makefile.common : macros for use by
                Cilk Makefiles

This distribution is set up for a Windows NT 4.0 system and an Intel Pentium processor. If you have any problems with this distribution, please contact volker.strumpen@yale.edu.

INSTALLATION

Prerequisites:

1. Windows NT 4.0
2. Cygwin B20.1 (www.cygnsus.com)
3. Perl (there may be several ports for Win32; ActivePerl is available at
   www.activestate.com)
4. The following prototype must be inserted (at line 1570, or following the prototype of
   GetSystemTime) into the CommonFunction.h header file in the include/Windows32
   subdirectory of the Cygwin distribution:

   VOID GetSystemTimeAsFileTime (LPFILETIME lpSystemTimeAsFileTime);

   The prototype for GetSystemTimeAsFileTime function was not included in the Cygwin
   headers for the Win32 API and is required to collect high-resolution timing measurements.
This initial port of Cilk to Windows NT was done under the Cygwin environment. The port was completed under version 20.1 of Cygwin and consists of the cilk2c compiler and runtime system. A number of examples have also been provided for testing purposes.

The default shell (sh.exe) is assumed to be available in /bin for some of the scripts that are executed during compilation.

Building the NT distribution

Step 1: Startup Cygwin and its bash shell

Step 2: From the main Cilk directory, typing “make” should compile the entire distribution. This includes the cilk2c compiler in the cilk2c directory, the runtime system in the rts directory, some library routines in the lib directory, and all of the examples in the examples directory.

Step 3: Play with the examples. There are some examples in the 'examples' directory. Additional information on the options available with each example are provided in the Cilk manual. Also, some example programs may provide additional information if executed with a "-h" flag.

PARSER HACK

The Microsoft Windows header file “windows.h” is included in the runtime system to allow successful compilation under NT. This file is included in the architecture-specific header file for NT (arch-nt-threads.h) and the auto-generated file containing the definitions of some Cilk structures (cilk-segments.h). Although the runtime system compiles without any problems using gcc, the cilk2c parser complains of parsing errors in some of the header files included through windows.h. Rather than rewrite the cilk2c parser to deal with these, the following hack was used. The header file windows.h is included when compiling the runtime system. Through the use of a preprocessor directive, this file is not included when a Cilk source file is being translated to C. Instead, only a subset of the definitions in the Windows headers are inserted for this stage of compilation. Once the C file is generated, a Perl script is executed by the Makefile to remove these definitions and reinser the Windows headers. The inclusion of the Windows headers is controlled with CILK_COMPILEghtRTS, a flag defined in the Makefile.

GENERATED HEADER FILES

During the compilation of a Cilk program, a number of header files may be generated if there are private variables included in the Cilk source. These files are safe to remove once the program has been compiled. They will be overwritten during the next
compilation phase. The files are named by attaching one of two prefixes (_private_vars.h or private_vars_init.h) to the name of the Cilk input file. The files contain definitions and functions needed for private variable support during compilation of the C code.
Appendix B – Description of attached file

The files described here are divided into three sections: testing, rts, cilk2c. The testing files were used in learning about NT’s features throughout the porting process. The rts and cilk2c files are part of the Cilk runtime system and the cilk2c compiler, respectively.

**mmap.c (testing)**
This file was an attempt to understand how the mmap system call works under Cygwin. Mostly written by Professor Strumpen, this file tries to map a shared memory segment using mmap as explained earlier. The call is successful if the MAP_SHARED flag is used but if the address to be mapped is specified using MAP_FIXED, the call is unsuccessful for unknown reasons thus far. The only other addition made to the file was the Win32 API function GetSystemInfo, which is used to retrieve the system memory allocation granularity for memory alignment:

```c
VOID GetSystemInfo(
    LPSYSTEM_INFO lpSystemInfo
);

typedef struct _SYSTEM_INFO {
    union {
        DWORD dwOemId;
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    },
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    WORD wProcessorLevel;
    WORD wProcessorRevision;
} SYSTEM_INFO;
```

The GetSystemInfo function is used several times elsewhere as well since it is important in collecting important information about the system architecture for the Cilk run-time system.

**mmap.cc (testing)**
Taken from the Cygwin sources, this file demonstrates the implementation of the mmap system call under Windows NT for the Cygwin runtime system. Specifically, the Win32 API functions CreateFileMapping and MapViewOfFileEx illustrated. In addition, if any of these API calls fail, the results of the GetLastError function are printed with the corresponding NT error code. These error codes are what allowed the determination of the actual cause behind the “incorrect file permissions” error reported by Cygwin.

**Threads1.cpp (testing)**
This was the first experimental file to understand basic thread creation and synchronization API calls under Windows NT. This simple program demonstrates the CreateThread function to create four separate threads, each of which increments the same index of a global array while printing the results and terminates when the counter reaches 100. The mutex synchronization is accomplished by storing “handles” to each of the threads in an array and passing this array to the WaitForMultipleObjects function.
Threads3.cpp (testing)
The Thread Local Storage is utilized fully here through the creation of three threads. Before the threads are spawned, the global TLS index is allocated using TlsAlloc. Once TlsAlloc is called, any thread that is currently executing in the process as well as any threads that may be created in the future will have the corresponding slot in the TLS array reserved. However, one must be careful to only use indices that are allocated by TlsAlloc as there is no error checking implemented to ensure that these system calls are as fast as possible. For example, if an index is used that was not allocated through TlsAlloc, a call to TlsSetValue will overwrite any existing value in that slot. The remaining API functions TlsGetValue and TlsFree are also demonstrated in this test file. Each thread, once spawned, sets a value in its TLS array at g_dwTlsIndex, gets the value, prints it, and then calls the same function to demonstrate that calling the same function from different threads will return the correct values for each thread.

Threads4.cpp (testing)
Most of the code in this file is taken from the May 1996 issue of Microsoft Systems Journal, which explores the Thread Information Block and how to access it directly. The code is also attached. Unfortunately, the offsets given by the original code were not correct for the TLS array and as a result examining the assembly code was required to obtain the proper offset as explained in section 4.3. Once the offset was found, the proper inline assembly code is used to arrive at the start of the TLS array and then values from 1 to 64 are written in the array and printed. As the code in the first thread demonstrates, each of these values can then be read back using the system call TlsGetValue and it returns the same values that were set “manually.” Thus, it is possible to interchange the system calls with direct read/write operations of the TLS array as long as one keeps track of which indices are being used.

Threads11.cpp (testing)
This file and the next were utilized in collecting timing information for accessing thread local storage. The code here writes an integer to each of the 64 TLS slots 10,000,000 times. High-resolution timing from the Win32 API is used in a stopwatch fashion to determine the running time of the entire loop.

Threads12.cpp (testing)
Similar to the last file, the code contained here writes data to each of the 64 slots in TLS within a long loop. Instead of using the system calls here, the data is written directly as explained in 4.3. The timing procedure used here is identical to that in Threads11.cpp. As shown in the paper, the results indicate that directly accessing TLS is more than ten times faster than using the system calls.

showtib.cpp (testing)
This is the original code that illustrates the internals of the Thread Information Block as written in the column “Under The Hood” in the Microsoft Systems Journal, May 1996.

cilk.c (rts)
This file is a part of the runtime system and includes the entry point for a Cilk
program. The code to initialize the 
_Cilk_private_vars_ structure is located here. A reference to the TLS area is
placed into the pointer for this structure. 
Support for other private variables is
also included here through the
initialization of two private variables.
_TLS_current_offset_ contains the offset
from the top of TLS to where the next
open slot is available. The variable
 isFirst_ is utilized during the private
variable initialization described in 4.4.
This variable allows the thread to
determine if it has allocated space for the
private variables in a given file or not.

arch-nt-threads.h (rts)
The architecture specific options are
mostly described in this file and the next.
This file is modeled after the Solaris
architecture file. The highlighted
changes in the file illustrate some of the
definitions that are used from the
Windows header files. In addition,
function calls to retrieve information
about the machine and high-resolution
timing are also included here.

arch-nt-threads.c (rts)
Similar to the last file, the code here
features some of the Win32 API calls
utilized in the porting process. The main
functions used here are for operating
system locks (mutexes) and for thread
management. The mutex functions
allow creation of mutexes along with
waiting on a given mutex and releasing
it once the thread is finished. Thread
management functions include creation,
join, release, and yield mechanisms.
Also, thread priority can be determined
and changed through Win32 API

private-vars.c (cilk2c)
Most functions required for supporting
private variables are contained here.
Routines are included for adding include
directives for files that will contain
definitions and procedures that are
required for allocating storage for
private variables and initializing the
structure that contains private variables
in each generated source file. Also, the
declaration of each private variable
structure is added to the generated C
source.

output.c (cilk2c)
This file has been modified with some
routines for additional private variable
support. One function generates the
header file that will contain the structure
of private variables. The other also
creates a header file, but this one
contains the initialization function for
the private variable structure used in the
current C source file.

transform.c (cilk2c)
The transformation procedure has been
altered to allow usage of private
variables. The highlighted changes
show that all nodes declared as
_T_PRIVATE_ are removed from the list
and the list of nodes is inserted into a
newly created structure that is output to
a separate header file in output.c.
Furthermore, every reference to a private
variable is replaced with a macro that
expands (as explained in 4.4) to access
the variable in the newly created
structure. Finally, the routines in
private-vars.c and output.c are called to
include the information required to
compile successfully.
### Appendix C – Win32 API and corresponding UNIX system calls

<table>
<thead>
<tr>
<th>Win32 API System Call</th>
<th>Solaris System Call</th>
<th>POSIX System Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetSystemTimeAsFileTime</td>
<td>gethrvtme, gethrtme</td>
<td>*</td>
</tr>
<tr>
<td>CreateMutex</td>
<td>musex_init</td>
<td>pthread_mutex_init</td>
</tr>
<tr>
<td>WaitForSingleObject</td>
<td>mutex_lock</td>
<td>pthread_mutex_lock</td>
</tr>
<tr>
<td>WaitForMultipleObjects</td>
<td>thr_join</td>
<td>pthread_join</td>
</tr>
<tr>
<td>ReleaseMutex</td>
<td>musex_unlock</td>
<td>pthread_mutex_unlock</td>
</tr>
<tr>
<td>CreateThread</td>
<td>thr_create</td>
<td>pthread_create</td>
</tr>
<tr>
<td>GetCurrentThread</td>
<td>_lwp_self</td>
<td>*</td>
</tr>
<tr>
<td>GetThreadPriority</td>
<td>priocntl</td>
<td>*</td>
</tr>
<tr>
<td>SetThreadPriority</td>
<td>priocntl</td>
<td>*</td>
</tr>
<tr>
<td>Sleep</td>
<td>thr_yield</td>
<td>*</td>
</tr>
</tbody>
</table>

* No equivalent POSIX function is available