

COMP 421: Lab # 1

Threads, Concurrency, and Operating System Device Drivers

Important Dates:

- In class, Thursday, January 26, 2012: Lab # 1 handed out
- 11:59PM, Monday, February 13, 2012: Lab # 1 due

1. Project Overview

Through this assignment, you will be able to gain practical experience in programming concurrent systems using threads with monitors for synchronization. You will also gain experience with the design and operation of device drivers in operating systems. In particular, in this project you will implement a terminal device driver using a Mesa-style monitor for synchronization of interrupts and concurrent driver requests from different user threads.

To simplify the project, we will use a software terminal emulation rather than working directly with real terminal hardware. This emulation is designed to give you the flavor of developing a real device driver, without as much complexity and frustration. We will also use a simplified threads library instead of the standard POSIX “Pthreads” library. We hope that by avoiding the overly large and complex Pthreads API, you will better be able to focus on the project itself.

2. Specifics

This project must be done individually. The second and third projects in the class will be done in groups of 2 students, but you must do this one by yourself.

This section discusses a few pieces of important information related to the overall project.

2.1. Environment

- Your solution must be implemented in C or C++.
- This project must be done on the CLEAR Linux systems at Rice. Specifically, all compiling, linking, and executing for the project will only work on CLEAR.
- Your use of threads must be limited to the package provided for the project, and to only those features described in this document. You may not directly use Pthreads or any other threads package or other threads or synchronization mechanisms other than the threads package provided for this project.
- Your terminal device driver must support the simulated terminal hardware as described in this document.
- You should create a new, separate directory to work in for this project. All work for the project should be done in this directory. Anything you want us to grade *must* be in this directory (or its subdirectories) when you turn in your project.

2.2. Form

- Your terminal device driver must compile into a single object file called `montty.o`.
- **You may not create any threads that are simply a part of your terminal device driver.** The only threads you may create should be the users of your terminal device driver (those that call procedures such as `ReadTerminal` and `WriteTerminal` that are part of the terminal device driver API described in Section 6.2).

2.3. Grading

Grading for this project will consider the completeness, correctness, design, and implementation of your solution:

- We will evaluate the correct operation of your terminal device driver, including through a series of private test programs. Since you will not know in advance what our private test programs do, you cannot write and test your code only to pass our private tests; you will need to focus on all aspects of correctness.
- Because of the nature of this assignment, design is important, and projects that do not correctly follow the required paradigms (particularly for structuring a Mesa monitor, as described in Section 8.2) will be severely penalized.
- Performance is also important in this project, and projects that perform badly due to synchronization issues will encounter design or implementation penalties, as appropriate.
- We will consider the design and implementation of your solution to help you improve. It is essential that you comment your code to the extent necessary to make it understandable for grading.

3. What is a Terminal Device Driver?

A device driver is a module within an operating system that manages some particular external device (or specific type of device) and encapsulates the messy details of interacting with the device's hardware. Encapsulation is particularly necessary for devices, because there is a great variety of them and complexity in dealing with them. It is easier to provide many small modules that make all devices of a general type look the same, than to add support for each new device in many places within the operating system or within application programs that use these devices. I/O devices are also particularly messy because they operate at very different speeds. This makes the procedures of hardware interaction baroque as well as diverse, and exposes the OS device driver to some insidious concurrency constraints.

Device drivers are generally structured into a “top half” and a “bottom half.” The top half deals with reacting to the application programs that request service from the I/O device, such as “read” and “write” system calls that may be issued by the program. The bottom half deals with reacting to the hardware, such as servicing interrupts from the device hardware. The top and bottom halves coordinate their work by sharing data structures within the device driver.

To work correctly, this coordination must use some form of synchronization in controlling access to those shared data structures. Also, device drivers usually must implement some sort of synchronization with user programs in order to keep different concurrent requests from different programs from interfering with each other.

In a real operating system, interrupt synchronization for a device driver is typically provided by the interrupt priority levels supported by the hardware and occasionally also by disabling interrupts for a short

time and then re-enabling them. *To simplify the interrupt synchronization in this project, though, and to provide you experience using monitors, the interrupt synchronization in this project will be provided solely through a monitor, rather than through hardware interrupt priorities and enabling and disabling interrupts.* In particular, the entire terminal device driver must be structured as a *single* Mesa-style monitor. Each time an interrupt from the hardware occurs, this translates in this project into an *entry procedure* in the monitor being called. Likewise, each time a user thread makes some request of the terminal device driver, such as to write some output on the terminal or to read some input from the terminal, this translates in this project into the thread calling an *entry procedure* of the terminal driver monitor.

4. The Terminal Hardware

The terminals in this project are simulated on top of `xterm` windows running on your display using X11 window System. The terminals support both input and output, with all data being sent and/or received one character at a time. The hardware supports up to a maximum of `MAX_NUM_TERMINALS` terminals, a value defined in the `hardware.h` header file. Each terminal is identified by a unique *terminal number*, ranging from 0 to `MAX_NUM_TERMINALS - 1`.

Each terminal's hardware displays each character written into the *output data register* for that terminal, but does not do so instantaneously. While the hardware is busy transmitting the character to the terminal, a new character must not be written into the output data register. When the transmission of the character from the output data register to the terminal is complete, the terminal hardware raises a *transmit interrupt*.

Each character typed on the keyboard of a terminal is deposited into the *input data register* for that terminal. When the hardware places a new character in this register, it raises a *receive interrupt*. In real-life hardware, if another character is typed on the terminal keyboard, it is placed into the input data register for the terminal regardless of whether or not the previous character in the input data register has been read by the device driver. If the previous character has not yet been read by the driver, that is, if the driver failed to respond sufficiently quickly to the previous receive interrupt, that previous character would be lost without having ever been seen by the device driver, since the input data register can hold only a single character. Our simulated terminal hardware behaves the same way, except that our "hardware" will also print an error message when this happens, letting you know that the newly typed character has overwritten the previous character in the input data register.

It is in general not possible (even in a real device driver with real hardware) to guarantee that the device driver can always respond to one receive interrupt and read the character from the input data register, before the user could type another character on the terminal's keyboard. But a well designed device driver should be able to complete handling each receive interrupt *very* quickly, making it very difficult for the user to type faster than the device driver can respond to each interrupt. With our simulated hardware, if you type very quickly, you may be able to generate this error message, but at normal typing speeds, this message should never occur if you have structured your device driver correctly.

5. The Hardware Emulation API

The API for the simulated terminal hardware is proved as part of the provided library and requires the inclusion of `hardware.h`. You should include this file as:

```
#include <hardware.h>
```

This file defines all of the function prototypes and other definitions for interacting with the hardware. The following functions are available:

- `void WriteDataRegister(int term, char c)`
 This hardware operation places the character `c` in the output data register of the terminal identified by the terminal number `term`. On any error, prints an error message on `stderr` and terminates the program.
- `char ReadDataRegister(int term)`
 This hardware operation reads (and returns) the current contents of the input data register of the terminal identified by `term`. On any error, prints an error message on `stderr` and terminates the program.
- `int InitHardware(int term)`
 This hardware operation initializes the terminal identified by `term`. It must be called once and only once before calling any of the other hardware procedures on that terminal. Returns 0 on success or -1 on any error.
- `int HardwareInputSpeed(int term, int msec)`
 This hardware operation may (optionally) be used to set the average delay associated with the reception of each character from the terminal identified by `term`. The parameter `msec` specifies the average number of milliseconds of delay between the time at which a character is typed on the terminal and the time at which the character is available on the input data register. The actual delay is not predictable (depends on physics) but its statistical distribution is. `HardwareInputSpeed` returns the previous value of this delay. On any error, prints an error message on `stderr` and terminates the program. If `msec` is equal to `NO_CHANGE`, this function returns the speed value without changing it. This function is not required for any reason, but you may find it useful in testing; by varying device speeds, you may be able to ferret out schedule-dependent bugs.
- `int HardwareOutputSpeed(int term, int msec)`
 This hardware operation may (optionally) be used to set the average delay associated with the transmission of each character to the terminal identified terminal. The parameter `msec` specifies the average number of milliseconds of delay between the time at which a character is written to the output data register and the time at which the output data register again becomes available for writing another character. `HardwareOutputSpeed` returns the previous value of this delay. On any error, prints an error message on `stderr` and terminates the program. If `msec` is equal to `NO_CHANGE`, this function returns the speed value without changing it. This function is not required for any reason, but you may find it useful in testing; by varying device speeds, you may be able to ferret out schedule-dependent bugs.

6. Required Terminal Device Driver Procedures

The device driver you write will need to service interrupts from the hardware as well as requests from the programs (threads) executing in the terminal. *The procedures in this section must be written by you*, and are called either from the interrupt dispatcher or from a user thread that requests an operation on the device.

6.1. Interrupt Handlers

As mentioned above, when the transmission of a character to a terminal completes, the terminal controller hardware signals a *transmit interrupt*. Similarly, when the receipt of a new character from a keyboard

completes, the terminal controller hardware signals a *receive interrupt*. In your terminal device driver, you must write a separate procedure to handle each of these two types of interrupts. Specifically, your terminal driver must provide the following interrupt handlers:

- `void ReceiveInterrupt(int term)`

This procedure is called by the hardware once for each character typed on the keyboard of the terminal identified by terminal number `term`, after that character has been placed in the input data register of that terminal. The character that caused the interrupt should be read from the input data register using the `ReadDataRegister()` operation. *You must write your own `ReceiveInterrupt` handler routine.* Within your receive interrupt handler, you should not block the current thread, since further calls to the receive interrupt handler for subsequent receive interrupts cannot be done until the current call to the receive interrupt handler returns.

- `void TransmitInterrupt(int term)`

This procedure is called by the hardware once for each character written to the output data register for the terminal identified by terminal number `term`, after the character has been completely transmitted to this terminal. After executing a `WriteDataRegister()` operation, you must assume that the output data register for that terminal is busy with the transmission until you receive the corresponding transmit interrupt and your `TransmitInterrupt()` procedure is called with the same terminal number. *You must write your own `TransmitInterrupt` handler routine.*

6.2. Device Driver API for User Threads

User threads in this project communicate with the terminal using procedures similar to the `read()` and `write()` system calls in Unix. *This section describes the functions that you must implement for handling these user requests.* You should `#include` the definitions of these functions from `terminals.h` as:

```
#include <terminals.h>
```

All procedures below, unless otherwise noted, should return 0 for success or -1 in case of any error.

- `int WriteTerminal(int term, char *buf, int buflen)`

This call should write to terminal number `term`, `buflen` characters from the buffer that starts at address `buf`. The characters must be transmitted by your terminal device driver to the terminal one at a time by calling `WriteDataRegister()` for each character. Your driver must block the calling thread until the transmission of the last character of the buffer is completed. This function should return the number of characters written (`buflen`), or -1 in case of any error. Your terminal device driver should not impose any limit on the maximum length of the buffer.

- `int ReadTerminal(int term, char *buf, int buflen)`

This call should read characters from terminal number `term`, placing each into the buffer beginning at address `buf`, until either `buflen` characters have been read or a newline (`'\n'`) has been read. The characters must be received by your terminal device driver, one at a time. Your driver should block the calling thread until this call can be completed. This function should return the number of characters read, or -1 in case of any error. Note that the `ReadTerminal` procedure should *not* place a null character at the end of the buffer.

- `int InitTerminal(int term)`

This procedure should be called once and only once by applications before any calls to the terminal device driver procedures defined above are called for terminal `term`. Among other things, your `InitTerminal` procedure must initialize the terminal controller hardware for this terminal by calling the `InitHardware` operation for terminal number `term`. Remember you may not create any threads that are simply a part of your terminal device driver. The only threads you may create should be the users of your terminal device driver (those that call procedures such as `ReadTerminal` and `WriteTerminal` that are part of the terminal device driver API described in Section 6.2). This means that you may *not* create any new threads in your `InitTerminal` routine.

- `int TerminalDriverStatistics(struct termstat *stats)`

This procedure returns a *consistent* snapshot of the I/O statistics for *all* terminals at once. The argument `stats` is a pointer to an array of `struct termstat` structures. The size of this array should be dimensioned at `MAX_NUM_TERMINALS` (that is, there should be space where this pointer points, for `MAX_NUM_TERMINALS` total `struct termstat` structures). The definition of a `struct termstat` is given in the `terminals.h` file for this project. For each terminal, this structure records the total number of characters received from and sent to the hardware for that terminal, and the total number of characters received from and returned to user programs for that terminal. The “in” and “out” counts in the `termstat` structure are defined from the point of view of the device driver. In particular:

- Each time your driver does a successful `ReadDataRegister` call for some terminal, the `tty_in` count for that terminal should increase by 1.
- Each time your driver does a successful `WriteDataRegister` call for some terminal, the `tty_out` count for that terminal should increase by 1.
- Each time some user thread does a valid `WriteTerminal` call for some terminal (i.e., your driver does not initially return an error and not process the request), the `user_in` count for that terminal should increase by `len`, the length of the buffer written by that request.
- Each time some user thread does a successful `ReadTerminal` call for some terminal, the `user_out` count for that terminal should increase by the number of characters returned by that request.

The I/O statistics for terminal number *i* are recorded in entry *i* in this array. If terminal number *i* has not yet been initialized with `InitTerminal`, all I/O statistics for this terminal should be recorded as -1 in the results returned by `TerminalDriverStatistics` (initialize the statistics for all terminals to -1 in `InitTerminalDriver` below).

- `int InitTerminalDriver()`

This procedure must be called once and only once by any application program before any other calls to terminal driver procedures, even before any calls to `InitTerminal` are made. This procedure should perform any specific overall initialization that your terminal device driver needs. This overall initialization should include initializing the terminal device driver monitor itself. Remember you may not create any threads that are simply a part of your terminal device driver. The only threads you may create should be the users of your terminal device driver (those that call procedures such as `ReadTerminal` and `WriteTerminal` that are part of the terminal device driver API described in Section 6.2). This means that you may *not* create any new threads in your `InitTerminalDriver` routine.

7. Terminal Behavior

7.1. Character Processing

Terminal device drivers typically do much more than transfer characters between memory and the terminal hardware. They also process characters in a myriad of ways before (on reading) or after (on writing) the application program sees those characters (for example, see “man `termios`” if you want to get a taste of the many settable parameters in Unix/Linux). In this project, you need not implement a complete set of these operations that Unix performs, but you must implement some. Specifically, you must carry out the character processing described in this section.

Any newline (`'\n'`) character output to the terminal by the user program must be transmitted to the device as the sequence of the two characters `'\r'` and `'\n'`, in that order. To the hardware, the `'\r'` character is known as a carriage return, and moves the cursor to the beginning of the line; similarly, to the hardware, the `'\n'` character is known as a linefeed, and moves the cursor down one line. The C/C++ language changes the meaning of `'\n'` (e.g., in a `printf`) to do both cursor movements as a single operation for programmer convenience, although this does confuse the issue a bit.

On input, the terminal hardware provides a carriage return (`'\r'` in C) when you hit the “Enter” key at the keyboard. This carriage return should be converted on input to a single “newline” character (`'\n'`) by your device driver. This conversion is again necessary because the meaning of `'\n'` in C/C++ is the same as the terminal meanings of both `'\n'` and `'\r'`.

All characters input from the terminal must be “echoed” back to the terminal. This allows the user typing on the keyboard to see each character as it is typed. In order for the terminal to appear responsive, the echoed characters should be transmitted back to the terminal at the earliest opportunity, regardless of what the user threads are doing or have done in the recent past. In particular *no application output characters should go to the terminal between the time a character is typed at the terminal and the time that input character is echoed to the terminal.*

Special processing is required when you receive either a “backspace” character (`'\b'`) or a “delete” character (`'\177'`) from the terminal. When either of these two characters is typed, you should remove in memory the last character from the current input line (if any). The current input line consists of those characters you have received from the terminal that have not yet been terminated as a line of input by the receipt of a newline character (`'\n'`, after the processing described above). If the current input line is empty, you should ignore the backspace or delete character. If the current input line is nonempty, you should remove the last character in the line. In no case should you include the backspace or delete character itself in the input line that is returned to a user program calling the `ReadTerminal` procedure.

In processing a backspace or delete character as described above, if you remove a character from the current input line in memory (because the current line was not empty), you should also echo to the terminal the three-character sequence of “backspace” (`'\b'`) followed by “space” (`' '`) followed by “backspace” (`'\b'`). In no case should you echo the delete or backspace character itself to the terminal. Echoing this three-character sequence is necessary because the effect of the user typing `'\b'` and `'\177'` should be to remove a character from the screen, which the terminal hardware can do only with the sequence above: the first character (backspace) moves the cursor one space to the left, the second character (space) replaces on the screen the character that was there with a space, and the third character (backspace) again moves the cursor one space to the left. This sequence will result in the last character on the screen appearing to be “erased.” If, instead, the current input line was already empty when the backspace or delete character was typed, you should ignore the backspace or delete character and should not echo anything for this character (some of you may want to ring the bell, as some terminals do; you can do this by sending `'\7'` to the terminal).

7.2. Line-Oriented Input

You will notice that the specification of `ReadTerminal()` considers the input as consisting of lines of text delimited with `'\n'`, but that, at the same time, the terminal hardware lets you read you a single character every time it raises the receive interrupt. Something similar can be said for `WriteTerminal()` and the transmit interrupt.

This implies that your terminal device driver must implement the illusion (abstraction) of a line-oriented terminal on top of a character-oriented hardware device. It is usually true, as it is in this case, that the diverse pieces of the operating system implement abstractions that are not directly or completely supported by the hardware.

Implementing this line-oriented behavior in the terminal device driver presents some complications that may not be obvious at first. Consider the following application code:

```
int len1, len2, len3;

char buf1[2];
char buf2[10];
char buf3[10];
char buf4[10];

len1 = ReadTerminal(0, buf1, 2);
len2 = ReadTerminal(0, buf2, 10);
len3 = ReadTerminal(0, buf3, 10);
len1 = ReadTerminal(0, buf4, 10);
```

and suppose the user types the following on the keyboard of terminal number 0:

```
Hello\b\b\b\bi\nUniverse\b\b\b\b\b\b\bWorld\nGood bye
```

From the specification of `ReadTerminal` and the character processing that the driver is to perform, after the application code executes, the variables have the following values:

```
len1 = 2
buf1 = "Hi"

len2 = 1
buf2 = "\n"

len3 = 6
buf3 = "World\n"
```

In addition, the calling thread will be blocked on the last `ReadTerminal()` until the user types a `'\n'` or `'\r'`. Think about why this is the case. You may come to realize that you need some buffering inside your terminal device driver.

Note, by the way, that although the contents of `buf1`, `buf2`, and `buf3` are shown above as they would be written for typical C/C++ strings, in reality *there is no null character at the end of each* as would normally be the case in C or C++. `ReadTerminal` should *not* put a null character at the end of the buffer when it returns.

7.3. Terminal Sharing Discipline

When used from different threads, the terminal functions can be called concurrently. Furthermore, program-driven output also occurs concurrently with the output from the echoing of input characters. You should decide what behavior is reasonable in the presence of concurrency, but here are some minimum standards.

When a mix of two or more `WriteTerminal` calls occur concurrently to the same terminal, their outputs must not be interleaved on the screen. For example, if two `WriteTerminal` calls are currently made to terminal number 1, you must not output some of the characters from one `WriteTerminal` call, then some from the other `WriteTerminal` call, followed by more from the first `WriteTerminal` call; instead, you must finish outputting *all* of the characters from the first `WriteTerminal` call, and only then begin outputting the characters from the second `WriteTerminal` call.

When, instead, a mix of two or more `WriteTerminal` calls occur concurrently to *different* terminals, the characters being output to each different terminal may be interleaved in time between the different terminals, as long as the characters from each individual `WriteTerminal` call are still output in the order they occur in the buffer that was an argument to that call. For example, if one `WriteTerminal` call is made to terminal number 1 and, concurrently, another `WriteTerminal` call is made to terminal number 2, one *possible* correct execution would be to output the first character from the first `WriteTerminal` call to terminal number 1, then output the first character from the second `WriteTerminal` call to terminal number 2, and then the next character from the first `WriteTerminal` call to terminal number 1, followed by the next character from the second `WriteTerminal` call to terminal number 2, and so on.

The echoed input characters must be displayed at the earliest opportunity, as previously mentioned. They can (indeed, sometimes must) be mingled, on the screen, with the output from one or more `WriteTerminal` calls. For example, half-way through outputting the characters from one call to `WriteTerminal`, if the user types a key on the keyboard, that typed character must be echoed right away, before the rest of the characters from that call to `WriteTerminal`.

Multi-character sequences resulting from the character processing described above must be displayed without other interleaved output characters on that terminal, even if, as in the case of the echo stream, the stream itself can be interleaved. This is to preserve the display effect of these sequences.

If there is more than one `ReadTerminal` call waiting to read characters from a terminal, the contents of each of their buffers must be filled by characters typed sequentially at the keyboard. That is, input characters should go to a single `ReadTerminal` call until that `ReadTerminal` returns, and then on to the other `ReadTerminal` call. For example, concurrent `ReadTerminal` calls reading from the same terminal should *not* alternate the characters read from that terminal.

Your terminal device driver should implement line-oriented terminals. For this project, this means that no data is returned for a `ReadTerminal` call until a newline has been received from that terminal. This does not necessarily imply that the whole line is returned to the `ReadTerminal` call: only as many characters as requested should be returned, and the rest remain available for the next `ReadTerminal` call, even if this call is made by a different user thread. If a `ReadTerminal` call asks for fewer characters than the current length of the next input line, the `ReadTerminal` call must still be blocked until the input line ends.

7.4. Device Driver Implementation Constraints

Device drivers are critical parts of the code of the operating system: they are very stressed by concurrency and operate at a very low level inside the system. Notably, they are often invoked from hardware interrupts, which typically preempt any other system activity, including other important OS functions. They are also restricted, for reasons that will become clear later in the course, to using only small portions of well-defined

“pinned” memory. Because of this, *your device driver should not block when running in an interrupt handler.*

Your device driver should allocate and use only a fixed amount of memory for its internal data structures. In the event that a sequence of terminal operations should require the driver to use more memory, the last operation of the sequence should either not be carried out or should be blocked, depending on whether the operation is executed from within an interrupt handler or not. In particular, it is all right to drop input characters if you run out of buffer space to store them because characters are coming in faster, on the average, than they can be consumed by the applications. However, it is also unreasonable to require applications to have a pending `ReadTerminal` call at the time every single character arrives. You thus must use buffers to deal with bursts and temporarily “absent” applications, but the buffers must be of finite size.

When adding one character to such a buffer would overflow the buffer, and the driver cannot block waiting for the buffer to drain (e.g., if called by an interrupt) you may choose a course of action. You may drop the character as stated above, and you might, depending on the situation, try to output a bell (`'\7'`) to the terminal. The bell is not required, but you should implement some reasonable behavior for all cases.

You may not create any threads that are simply a part of your terminal device driver. The only threads you may create should be the users of your terminal device driver (those that call procedures such as `ReadTerminal` and `WriteTerminal` that are part of the terminal device driver API described in Section 6.2.

8. Using Threads and Concurrency

8.1. The Threads Package

The threads package you will use in this project is simplified in order to be easy to use and to have only the procedures you may need in the project. In order to use the threads package, you must include the function prototypes and definitions for the library as:

```
#include <threads.h>
```

You will also need to explicitly link in the threads library routines when you link your program. This is done automatically when you use the provided Makefile to link your code, as described in Section 10. (In case you are curious, the threads library routines, as well as the terminal hardware emulation routines, are part of the `liblab1.a` archive linked in by the provided Makefile.)

The threads package provides the following functions (you may not need all of these in the project, but they are provided for completeness):

- `thread_id_t ThreadCreate(void (*func)(void *), void *arg)`

The `ThreadCreate()` function starts a new thread that begins executing the procedure `func`. The `func` procedure should have a prototype of

```
void func(void *)
```

This means that `func` is a procedure that takes a generic pointer as an argument and does not return anything. The argument is specified via `ThreadCreate()`'s `arg` argument. This argument can be cast to the appropriate type, as needed. Upon success, `ThreadCreate()` returns the thread identifier of the new thread. Upon failure, `ThreadCreate()` prints an error message on `stderr` and terminates the program.

Remember, you may *not* create any threads that are simply a part of your terminal device driver. The *only* threads you may create should be the users of your terminal device driver (those that call

procedures such as `ReadTerminal` and `WriteTerminal` that are part of the terminal device driver API described in Section 6.2).

- `void ThreadWait(thread_id_t th)`

The `ThreadWait()` function waits for the thread identified by the thread identifier `th` to have exited; if that thread has not yet exited, the calling thread is blocked until that thread exits. On any failure, `ThreadWait()` prints an error message on `stderr` and terminates the program.

- `void ThreadWaitAll(void)`

The `ThreadWaitAll()` function waits for *all* threads previously created by `ThreadCreate()` to have exited. On any failure, `ThreadWaitAll()` prints an error message on `stderr` and terminates the program. In a typical user program, the `main()` thread should call `ThreadWaitAll()` before exiting (e.g., before “falling off” the bottom of `main()`).

- `void Declare_Monitor_Entry_Procedure(void)`

`Declare_Monitor_Entry_Procedure()` is used to declare that the current procedure is an entry procedure of the monitor. This call must be used only at the *beginning* of each C or C++ procedure that implements an entry procedure of the monitor. It acquires the mutual exclusion of the monitor, and when this entry procedure returns, the mutual exclusion for the monitor is then automatically released. Note that the threads package for this project supports only a single monitor, so all entry procedures defined by `Declare_Monitor_Entry_Procedure()` are implicitly part of the same monitor. On any failure, `Declare_Monitor_Entry_Procedure()` prints an error message on `stderr` and terminates the program.

- `cond_id_t CondCreate(void)`

The `CondCreate()` call creates a new conditional variable and returns the condition variable id of the new condition variable. On any failure, `CondCreate()` prints an error message on `stderr` and terminates the program.

- `void CondWait(cond_id_t cv)`

The `CondWait()` call performs the condition variable wait operation on the condition variable identified by the condition variable identifier `cv`. In particular, this call blocks until the same condition variable is signaled using `CondSignal()`, but only one of possibly several waiting threads will be unblocked at that point. `CondWait()` releases the mutual exclusion of the monitor before blocking, to allow entry into other entry procedures protected of the monitor, and then reacquires this mutual exclusion before returning after being woken up by a signal operation. `CondWait()` may only be called while inside the monitor. On any failure, `CondWait()` prints an error message on `stderr` and terminates the program.

- `void CondSignal(cond_id_t cv)`

The `CondSignal()` call performs the condition variable signal operation on the condition variable identified by condition variable identifier `cv`. It causes one thread currently waiting for this condition variable to become unblocked, ready for execution. The waiting thread will not actually execute until later, after the calling thread has exited the monitor or itself become blocked by waiting on some condition variable. `CondSignal()` may only be called while inside the monitor. On any failure, `CondSignal()` prints an error message on `stderr` and terminates the program.

- `void CondDestroy(cond_id_t cv)`

The `CondDestroy()` call deactivates the condition variable identified by `cv`. Any subsequent calls to `CondSignal()` or `CondWait()` on this condition variable will result in an error. On any failure, `CondDestroy()` prints an error message on `stderr` and terminates the program.

8.2. Using the Mesa Monitor Paradigm

As you know from the lectures, a monitor is a synchronization primitive that is similar to an “object” in the “object-oriented” sense, with “private” variables that can only be accessed through public “methods” known as entry procedures provided by the monitor. Monitors automatically provide mutual exclusion and also provide operations for concurrent threads to coordinate through condition variables.

Although the C and C++ languages do not provide specific support for monitors, it is possible to use monitors and to follow the monitor paradigm in C or C++ using the threads library procedures defined above. However, doing so requires a little self-discipline in how the threads library calls are used.

An example of how to correctly write a monitor in C or C++ is provided on CLEAR in the directory

```
/clear/courses/comp421/lab1/philosophers/
```

This is an implementation of the classic Dining Philosophers problem, as described in class and in the textbook. In particular, the file `philosopher.c` in this directory contains *all* of the source code for the monitor itself. For convenience (and to encourage you to *read* this code to see how to write a monitor), the complete code for this implementation of the Dining Philosophers problem is reproduced in the Appendix to this handout.

In order to use the monitor paradigm to provide mutually exclusive access to a set of shared variables, you must use the threads library procedures in this project as shown in this Dining Philosophers example and as detailed below:

- A monitor is a collection of C or C++ procedures and *all* shared variables controlled by those procedures.
- You should put all of the code for a monitor into a *single* file of C or C++ source code. You should also put an initialization procedure for your monitor in that one source file. Nothing other than the monitor should be included in that same file. For example, as noted above, all of the source code for the example Dining Philosophers monitor is in the single file `philosopher.c`. The initialization procedure there is `init_philosophers()`.
- You should create any condition variables to be used in the monitor in this initialization procedure.
- Some of the C or C++ procedures in this file will be entry procedures of the monitor, and some will be normal internal procedures called only from within the monitor. Each entry procedure should be defined as `extern`, and each internal procedure should be defined as `static`. The example monitor has two entry procedures, called `pickup_forks()` and `putdown_forks()`, and both are declared as `extern`. The example has one internal procedure, called `test_forks()`, and it is declared as `static`.
- At the *beginning* of each entry procedure, you should declare this procedure to be an entry procedure by calling `Declare_Monitor_Entry_Procedure()`. This procedure acquires the mutual exclusion of the monitor and arranges for the mutual exclusion to be automatically released when this entry procedure returns.
- Each condition variable should be used only from within the single monitor.

Please be sure to follow these rules in writing any monitor. The example monitor in `philosopher.c` shows clearly how to do this.

8.3. User Programs

In this project, each user program is composed of a set of threads that may call any of the procedures `WriteTerminal()`, `ReadTerminal()`, `TerminalDriverStatistics()`, `HardwareInputSpeed()`, and `HardwareOutputSpeed()` at will, but only after the procedure `InitTerminal()` has been called once (and only once) for each terminal used. In addition, some user program must call `InitTerminalDriver()` once before any other calls to any other terminal device driver procedures (including calls to `InitTerminal()`). The one call to `InitTerminalDriver()` and the calls to `InitTerminal()` (one for each terminal you will use) should be done by your `main()` procedure, described next.

When a user program begins execution, only a single thread is executing. This thread begins executing at the normal `main()` routine of the C or C++ program; this thread is called the “boot” thread. When the boot thread begins execution, neither the terminal hardware nor your terminal device driver are active or initialized yet. The boot thread should call `InitTerminalDriver()` and `InitTerminal()`, as described above. The boot thread should also create any other threads you will use in your program.

The boot thread otherwise acts like any other thread, except that when it ends (e.g., by calling `exit()` or by “falling off” the bottom of `main()`), the entire program, including *all* other threads, is terminated. If you want to wait for all other threads to have completed normally before ending `main()`, you can call the `ThreadWaitAll()` routine provided as part of the threads package (Section 8.1).

Please also note that your `main()` procedure is not part of the terminal device driver but is instead part of the user program that uses the terminal device driver. Thus, any code related to your device driver, including the initialization of any data structures you define in your driver, should *not* be part of or depend on any particular `main()` routine: your terminal device driver should be able to work with any `main()` routine and any mix of application threads started by that `main()` routine. In particular, when we test your terminal device driver, we will replace the `main()` routine with one of our own. Because of this, you should put the code for any `main()` you use in a source file separate from the file you use to hold the source code for your device driver itself. We will provide several sample test programs on CLEAR in the directory `/clear/courses/comp421/lab1/samples`. You should also write a number of your own test programs to test your device driver. The sample test programs we will provide are intended only as examples and will not test all features of your device driver (and we will use different test programs when *grading* your project).

9. Running Your Terminal Device Drivers

When you finally have a version of your terminal device driver compiled and you are ready to test it, you simply run your executable file that has been linked by the provided Makefile (Section 10).

This executable file, as linked with the provided library (done automatically by the provided Makefile), will try to open a new `xterm` on your display whenever `InitHardware` is called for some new terminal. This `xterm` will emulate this terminal device, with which the hardware procedures (e.g., `WriteDataRegister`) interact.

Any output from your program (written to `stdout` and `stderr` via, e.g., `printf` or `fprintf`) will go to the Linux shell from which you invoked the executable file (in the same way as normal output from normal C or C++ programs).

When using the X11 Window System to support each `xterm` on CLEAR, you will need to log in to CLEAR using `ssh`; each `xterm` window will then display on your local screen. For this to work correctly,

you will need to tell your local `ssh` client program to allow X11 Window System operations to be forwarded through the `ssh` connection. However, different `ssh` client programs may do this in different ways; if you are using one of the common Unix/Linux/cygwin command line-based `ssh` clients, you probably want to use the `-X` option on the `ssh` command line to do this.

Remember that in all cases, you must be logged into a CLEAR machine before you can compile, link, or run your terminal device driver.

10. Building Your Project

A template Makefile is available for your use, that will automatically compile your terminal device driver into `montty.o`. This template Makefile will also automatically make any user test programs that you want to test your terminal driver with; these test programs will automatically be linked with `montty.o` and with the necessary libraries needed to make the terminal hardware emulation and the provided threads package work.

This template Makefile is available on CLEAR at

```
/clear/courses/comp421/lab1/samples/Makefile.template
```

You should make a copy of this file into your own directory that you use for the project. You should name your copy of this file just `Makefile`.

This template Makefile will work with either `gcc` or with `g++`. The Makefile is initially set up to use `gcc`. To instead use `g++`, edit your copy of `Makefile` and comment out the `LANG = gcc` line (add a `#` at the beginning of the line) and uncomment the following `LANG = g++` line (remove the `#` at the beginning of that line).

The only thing else you should need to modify in your `Makefile` is the line that begins “`TEST =`”. This line defines a list of test programs that will automatically be compiled and linked, that you want to test your terminal device driver with. You should edit this list to be a list of your *own* test programs (the example initially given in the `Makefile` is to make test programs named `test1`, `test2`, and `test3`).

For each user test program in this `TEST` list, the Makefile will make the program out of a single correspondingly named source file. In addition to this single source file for each test program, each program will also be linked with your device driver. For example, the Makefile will make `test1` out of `test1.c`, if you have a file named `test1.c` in this directory.

Any `#include` files written by you and included by your programs should be located in your work directory and should be `#included` with double quotes; that is, you should include them using

```
#include "foobar.h"
```

Any include files that we provide that you use should be `#included` using

```
#include <foobar.h>
```

*You should not copy any of our include files into your own directory. Using this form of `#include` with `<...>` will automatically get them (in their most recent version, in case any revisions become necessary) from the common `comp421` directory for this project. Likewise, *you should not copy any of our library files to your own directory.**

11. Turning in Your Project

Your project is due by *11:59PM on Monday, February 13, 2012*. Before turning in your project, please create a file named “`README`” in the same directory as the rest of your files for the project. In this file,

please describe anything you think it would be helpful for the TAs to know in grading your project. This might, for example, mean describing unusual details of your algorithms or data structures, and/or describing the testing you have done and what parts of the project you think work (or don't work).

To submit your project for grading when you are ready, please perform the following two steps:

- First, on CLEAR, change your current directory to the directory where your files for the project for grading are located. For example, use the “cd” command to change your current directory. When you run the submission program, it will submit everything in (and below) your current directory, including all files and all subdirectories (and everything in them, too).
- Second, on CLEAR, run the submission program

```
/clear/courses/comp421/lab1/bin/lab1submit
```

This program will check with you that you are in the correct directory that you want to submit for grading, and finally, will normally just print “SUCCESS” when your submission is complete. If you get any error messages in running `lab1submit`, please let me know.

12. Honor Code Policy

All assignments in the course are conducted under the Rice Honor Code. For programming assignments such as this one, students are encouraged to talk to each other, the TAs, the instructor, or anyone else about the assignment. This assistance, though, is limited to discussion of the problem; *each student must produce their own solution*. Consulting another student's solution (even from a previous COMP 421 class) is prohibited, and submitted solutions may not be copied from any source.

13. Suggested Plan of Attack

There is no need for you to do things in any order. However, some parts of this assignment will take longer than you think. This is not because you will have to write a lot of code but because some of the bugs you will encounter will be reluctant to show themselves. One of many reasonable plans to attack this project is as follows:

- While working on the project, remember to check the class web page regularly. Any announcements such as clarifications or corrections will be posted there.
- Read this complete handout. Read it again. Make sure you understand the bit about the top and bottom halves of device drivers. Think about how they should interact. Realize that the top and the bottom halves operate asynchronously, with the top half being called by user threads and the bottom half being called by a different thread for each type of interrupt. The granularity of the objects that the top and bottom half work on are also different (lines versus characters).
- Remember that you may *not* create any threads that are simply a part of your terminal device driver. The only threads you may create should be the users of your terminal device driver (those that call procedures such as `ReadTerminal` and `WriteTerminal` that are part of the terminal device driver API described in Section 6.2).
- Decide where buffers and other shared structures will be needed and which functions will interact with them. Draw out your design on paper. Show the buffers and other data structures, as well as the procedures that will access them.

- Identify the synchronization problems. Think about the behavior of each terminal, both input and output. Also think about the behavior of the `TerminalDriverStatistics` call, which requires a *consistent* picture of the state of *all* terminals at once.
- Identify the critical section. Add notations indicating these to your diagram.
- Create an initial version of your terminal driver monitor that can successfully echo characters to the terminal, without having the `ReceiveInterrupt` handler block on the `TransmitInterrupt` handler (you will need to have at least *some* version of `ReceiveInterrupt` and `TransmitInterrupt` defined in your code in order to be able to link and run even this initial version of your device driver). Don't worry yet about any input or output character processing; for now, just echo each character "as is." *Test and debug* this version of your device driver using your own simple test programs before proceeding with the rest of the project. You should be able to type one character at a time and see it echo back onto the screen.
- Add support for `WriteTerminal` to your device driver. Make sure that the echo has priority over `WriteTerminal` output and that two or more concurrent `WriteTerminal` calls do not interfere with each other. Again, don't worry yet about any input or output character processing; for now, just output each character "as is." *Test and debug* this version of your driver using your own simple test programs before proceeding with the rest of the project.
- Add support for `ReadTerminal` into your driver. Make sure that echoing of characters to the screen is not affected by an application's failure to call `ReadTerminal`. Again, *test and debug* this version before proceeding with the rest of the project.
- Think about where best to introduce input and output character processing. Realize that the processing required for echo, output, and input, are different, but that they are not entirely dissimilar either.
- Add character processing.
- Thoroughly test what you have written so far. Try writing different test programs to exercise and stress different features of your terminal driver.
- Test everything some more.
- When you think you are ready, turn in your project by running the `lab1submit` program as described in Section 11.
- Relax. You are done with the project. Congratulations.

Appendix: Example of How to Write a Monitor (Dining Philosophers)

This appendix shows an example of how to use the threads package provided for this project, including how to create threads and how to write and use a monitor. In particular, this appendix shows a solution to the Dining Philosophers problem. The solution is divided into two files, as shown below.

File "philmain.c"

The file "philmain.c" contains the `main()` routine that creates one thread for each of the five philosophers, each executing the `do_phil()` routine (note that this Dining Philosophers program does not use the terminal hardware and so does not call `InitTerminalDriver()` or `InitTerminal()`). The

do_phil routine in this file then executes the behavior of each philosopher, showing how the philosopher threads call the two entry procedures of the monitor: pickup_forks() and putdown_forks().

```
#include <stdio.h>
#include <stdlib.h>
#include "threads.h"

void
do_phil(void *arg)
{
    int i = (int)(long)arg;

    printf("do_phil %d\n", i);

    while (1) {
        pickup_forks(i);
        printf("eating %d\n", i);
        putdown_forks(i);
        printf("thinking %d\n", i);
    }
}

int
main(int argc, char **argv)
{
    int i;

    init_philosophers();

    for (i = 0; i < 5; i++) {
        ThreadCreate(do_phil, (void *) (long)i);
    }

    ThreadWaitAll();

    exit(0);
}
```

File “philosopher.c”

The file “philosopher.c” is the monitor itself. Each of the two entry procedures, `pickup_forks()` and `putdown_forks()`, is actually made into an entry procedure by calling `Declare_Monitor_Entry_Procedure()` at the beginning of that procedure.

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

/*
 * The state of each of the 5 philosophers: either THINKING, HUNGRY,
 * or EATING.
 *
 * This is declared 'static' so it can't be seen outside this .c file.
 */
static int state[5];
#define THINKING    0
#define HUNGRY     1
#define EATING     2

/*
 * A condition variable for each philosopher to wait on. Declared
 * 'static' as with all variables that should only be seen inside
 * this monitor.
 */
static cond_id_t philcond[5];

#define LEFTPHIL    ((i+1) % 5)    /* the philosopher to i's left */
#define RIGHTPHIL  ((i+4) % 5)    /* the philosopher to i's right */

static test_forks(int);

/*
 * Pick up both of the forks for philosopher i.  Waits until
 * both forks are available.
 *
 * This is an entry procedure for the monitor, so:
 * - it is an 'extern' function (any function not defined 'static'
 *   is automatically extern, according to the C language; and
 * - it acquires the mutual exclusion of the monitor at the top
 *   and releases it at the bottom.
 */
extern
pickup_forks(int i)
{
    /*
     * You MUST use Declare_Monitor_Entry_Procedure() at the
     * beginning of EACH and EVERY entry procedure of your monitor.
     * You MUST NOT use Declare_Monitor_Entry_Procedure() anywhere
     * else.  This call acquires the mutual exclusion of the monitor
     * and arranges for the mutual exclusion to be automatically
     * released when this entry procedure returns.
     */
    Declare_Monitor_Entry_Procedure();
}
```

```

    state[i] = HUNGRY;
    test_forks(i);
    while (state[i] != EATING)
        CondWait(philcond[i]);
}

/*
 * Put down both of the forks for philosopher i.  If this allows either
 * the philosopher to our left or to our right to begin eating, let them
 * eat.
 *
 * As with pickup_forks, this is also an entry procedure of the monitor.
 */
extern
putdown_forks(int i)
{
    /*
     * You MUST use Declare_Monitor_Entry_Procedure() at the
     * beginning of EACH and EVERY entry procedure of your monitor.
     * You MUST NOT use Declare_Monitor_Entry_Procedure() anywhere
     * else.  This call acquires the mutual exclusion of the monitor
     * and arranges for the mutual exclusion to be automatically
     * released when this entry procedure returns.
     */
    Declare_Monitor_Entry_Procedure();

    state[i] = THINKING;
    test_forks(LEFTPHIL);
    test_forks(RIGHTPHIL);
}

/*
 * Test whether philosopher i can begin eating.  If so, move him
 * to EATING state and signal him (in case he is waiting).
 *
 * This is an *internal* (non-entry) procedure of the monitor.  Thus,
 * it is a 'static' function, making this function name not known
 * outside this .c file, so it can't be called from outside the
 * monitor.  An internal procedure should be called only from a
 * monitor entry procedure (or from another internal procedure of
 * the monitor).
 */
static
test_forks(int i)
{
    if (state[LEFTPHIL] != EATING &&
        state[i] == HUNGRY &&
        state[RIGHTPHIL] != EATING) {
        state[i] = EATING;
        CondSignal(philcond[i]);
    }
}

```

```
/*
 * Initialize the Dining Philosophers monitor.
 *
 * This procedure should be called *once* when the whole program starts
 * running. It creates the condition variables and initializes the
 * shared variables used inside the monitor.
 */
init_philosophers()
{
    int i;

    for (i = 0; i < 5; i++) {
        state[i] = THINKING;
        philcond[i] = CondCreate();
    }
}
```