RICE

Spring 2024

Lab 1

Threads, Concurrency, and Operating System Device Drivers

COMP 421/ELEC 421, COMP 521/ELEC 552

Important Dates:

- Thursday, January 25, 2024: Lab 1 begins
- Monday, February 12, 2024: Lab 1 due by 11:59 PM

1 Project Overview

Through this assignment, you will be able to gain practical experience in programming concurrent systems using *threads* with a *monitor* 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 that share an address space.

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.

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

This project is conducted under the Rice Honor Code, a code that you pledged to honor when you matriculated at Rice. See, in particular, Section 14.

2 Specifics

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

2.1 Environment

- Your solution must be implemented in the C programming language (e.g., *not* in C++ or other programming languages).
- 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.
- For the terminal hardware simulation to work correctly, you must be running a local X11 Windows server program on your own local computer, and must log in to CLEAR using `ssh` in a way that allows X Window operations to be forwarded back to your local X Windows server program through this `ssh` connection. This use of X Windows and `ssh` is explained more fully in Section 9.
- 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

- The source code for your terminal device driver must be in a single source file named `montty.c`, and your device driver must compile into a single object file named `montty.o`. This name is short for "monitor" and "terminal" ("tty" is actually short for "teletypewriter," a very old and still very common name for a terminal).
- You may use only a total of *one* monitor in your solution (e.g., *not* one monitor per terminal).
- All of the procedures defined in this project description in Section 6.1 (`ReceiveInterrupt` and `TransmitInterrupt`) and in Section 6.2 (`WriteTerminal`, `ReadTerminal`, `InitTerminal`, `TerminalDriverStatistics`, and `InitTerminalDriver`) *must all be entry procedures of your monitor*. Your monitor must have *no* other entry procedures.
- **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). You may *not* call the `ThreadCreate` function anywhere within your terminal driver.
- Your terminal device driver must guarantee that no deadlock is possible and must generally also ensure that no starvation is possible as well. The one case in this project where starvation may be allowed is due simply to the use of Mesa monitor semantics rather than Hoare semantics. Specifically, before most condition variable waits, a Hoare monitor only requires an `if` whereas a Mesa monitor generally requires a `while`. This use of a `while` is generally required for correctness but may itself be the source of possible starvation, if *every* time the process is awoken from a signal, the process ends of going around the loop and waiting again on the same condition variable.
- For students taking COMP 521 or ELEC 552, the *graduate student* version of this course, you must also write and submit a report as defined more fully below in Section 11.

2.3 Grading

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

- Your submitted project must comply with the form requirements described above in Section 2.2.
- 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 possible concurrency problems.

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" 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 largely 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 a monitor, 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 another *entry procedure* of the terminal driver monitor.

4 The Terminal Hardware

The terminals in this project are each simulated using windows your local display using X11 Window System. X Windows makes it possible for your program running on CLEAR to open windows on your local display to simulate each hardware terminal. 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 `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 `NUM_TERMINALS-1`.

Each terminal's hardware displays on that terminal's screen each character written into the *output data register* for that terminal, but does not do so instantaneously. While the hardware is busy transmitting one character to the terminal for display, a new character must not be written by the driver into the output data register, as the output data register in the hardware can hold only a single character at a time. When the transmission of the character from the output data register to the terminal is complete, the terminal hardware causes 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 causes a *receive interrupt*. Like the output data register, the input data register in the hardware can hold only a single character at a time. 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, again, the input data register can hold only a single character. Our simulated terminal hardware behaves in the same way, except that the "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.

The *hardware* of each terminal operates entirely independently. For example, the hardware's handling of input of characters typed on one terminal is independent of the input of characters typed on any other terminal, and the hardware's handling of output of characters from your device driver to one terminal is independent of the output of characters to any other terminal. Each terminal in hardware has its own input data register and output data register, independent of each other terminal. And the hardware of each terminal causes receive interrupts and transmit interrupts, as needed, completely independent of any other terminal.

Note that some keys on your actual keyboard will generate more than one input character from pressing that keyboard key only once. For example, on most keyboards, when you press the "up arrow" key once, the keyboard will generate the three-character sequence "`\33[A`". That is an "escape" character (`'\33'`), followed by an open square bracket character (`[`), and followed by a capital A character (`A`). The effect from the keyboard hardware is exactly the same as if you had pressed each of these three other individual keys on the keyboard once each. Your terminal device driver will thus get a sequence of three `ReceiveInterrupt` calls, once for each of these three other characters, all resulting from the single key press of the "up arrow" key on the keyboard, exactly the same as if you had indeed instead typed each of these three other characters once each. For each `ReceiveInterrupt` call, your device driver should just handle that character as for any other `ReceiveInterrupt` call. The fact that these three input characters all came from a single press of the "up arrow" key is irrelevant; there is nothing special you need to do because of that fact.

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 the include file `hardware.h`. You should include this file in your source 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, in this project, this function 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, in this project, this function 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.

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) using the terminal. *All of the procedures in this section must be written by you.* These procedures are called either from the hardware as an interrupt handler or from a user thread requesting an operation on a terminal.

6.1 Interrupt Handlers

As mentioned above, when the transmission of a character to a terminal completes, the terminal hardware signals a *transmit interrupt*. Similarly, when the receipt of a new character from a keyboard completes, the terminal 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. In this project, these two interrupt handlers must have specific procedure names, whereas in a real (complete) operating system, the procedure name would be irrelevant, as each interrupt handler would be called simply by a pointer to that procedure being found at the right subscript within the interrupt vector table. Specifically, for this project, *you must write the following two interrupt handler procedures as part of your terminal device driver:*

- `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 for that terminal. The character that caused the interrupt should be read by your device driver 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 (i.e., wait on a condition variable), 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 of that character until you receive the corresponding transmit interrupt and your `TransmitInterrupt()` procedure is called with this 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()` kernel calls in Unix/Linux. In addition to the two interrupt handler procedures defined above in Section 6.1 that you must implement, *this section describes the additional functions, which you must implement for handling these user requests.* You should `#include` the definitions of these functions from the include file `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.

  Other than as noted in Section 7.1, *all* output characters should be sent to the terminal exactly as-is (e.g., without worrying if the character is any special type of character such as a "control character").

  Your driver must block the calling thread until the transmission of the last character of the buffer is completed (including receiving a `TransmitInterrupt` following the last character); this call should not return until then. This function should then return as its result the number of characters written (`buflen`), or `-1` in case of any error. If `buflen` is 0, this call should return immediately, and if there are no errors, the return value should be 0 in this case. Your terminal device driver should not impose any limit on the maximum length of the user's buffer (`buflen`).

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

  This call should copy characters typed from terminal number `term`, placing each into the buffer beginning at address `buf`. As you copy characters into this buffer, continue until either `buflen` characters have been copied into `buf` or a newline (`'\n'`) has been copied into `buf` (whichever occurs first), but note that, as described in Section 7.2, only characters from input lines that have been terminated by a newline character in the input buffer can be returned. Your driver should block the calling thread until this call can be completed.

  Other than as noted in Section 7.1, *all* input characters should be returned to the calling thread exactly as-is (e.g., without worrying if the character is any special type of character such as a "control character" or if the user typed one of the cursor or "arrow" keys).

  This function should return as its result the number of characters copied, or `-1` in case of any error. If `buflen` is 0, this call should return immediately, and if there are no errors, the return value should be 0 in this case. Note that the `ReadTerminal` procedure should *not* place a null character (`'\0'`) at the end of the buffer.

- `int InitTerminal(int term)`

  This procedure should be called once and only once before any other call 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. Note that it is an error to initialize the same terminal more than once or to attempt to perform any other terminal device driver API calls on a terminal before that terminal has been initialized.

- `int TerminalDriverStatistics(struct termstat *stats)`

  This procedure returns a *consistent* snapshot of the I/O statistics for *all* terminals all at once. The description "consistent" here means "all taken at the same time, under the same mutual exclusion of the monitor." Thus, none of the statistics can change within the device driver while you are copying out to the user's buffer the terminal driver's statistics. The argument `stats` must be a pointer to an array of `struct termstat` structures. The size of this array should be `NUM_TERMINALS` number of entries (that is, there should be space where this pointer points, for `NUM_TERMINALS` total `struct termstat` structures). The calling thread must have allocated/created the memory that this pointer points to before calling `TerminalDriverStatistics`. The definition of a `struct termstat` is given in the `terminals.h` include 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, since that terminal was initialized by `InitTerminal`. 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 has completed 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 have been increased by `len`, the length of the buffer written by that request.
  - Each time some user thread has completed a successful `ReadTerminal` call for some terminal, the `user_out` count for that terminal should have increased 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`. Immediately after initialization with `InitTerminal`, all I/O statistics for this terminal should be recorded as `0` in the results returned by `TerminalDriverStatistics`.

- `int InitTerminalDriver()`

  This procedure must be called once and only once 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, try the command "`man termios`" if you want to get a taste of the many settable parameters for terminal character processing in Unix/Linux). In this project, you need not implement a complete set of these operations that Unix/Linux performs, but you must implement some of them. Specifically, you must carry out the character processing described in this section as part of your terminal device driver.

All characters *input from* the terminal must in general be "echoed" back to the terminal when they are typed (actually, when you get them from the input data register). This echoing 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 user application output characters from* `WriteTerminal` *should go to the terminal between the time a character is typed at the terminal (and you get a receive interrupt) and the time that input character is echoed to the terminal*. Any characters being output for echoing should be given priority over any characters being output by user application thread calls to `WriteTerminal`. When echoing most typed characters, you simply output the character that was typed, but some typed characters require special processing and should result instead in echoing a multi-character sequence, rather than literally echoing the typed character itself, as described directly below; as described further in Section 7.3, such multi-character sequences resulting from this character processing *must* be echoed without other interleaved output characters on that terminal.

On input, the terminal hardware provides a "carriage return" (`'\r'` in C) character when you hit the "Enter" key on the keyboard. This carriage return should be converted on input to a single "newline" character (`'\n'`) by your device driver (looking ultimately to user threads as if the user had typed `'\n'` rather than `'\r'`). The echo for this typed single character should consist of the special two-character sequence `'\r'` and `'\n'`, in that order. To the hardware, the `'\r'` character is known as a "carriage return," and when output moves the cursor to the beginning of the current displayed line; similarly, to the hardware, the `'\n'` character is known as a "linefeed," and when output moves the cursor down one line on the display.

Special processing is also 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 (internal to your device driver) the last character (if any) from the current input line. 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 in memory is empty, you should ignore the backspace or delete character from the terminal. If the current input line in memory is nonempty, you should remove the last character in the line. In no case should you include the backspace or the delete character itself in the input line that is returned to a user thread calling the `ReadTerminal` function.

Table 1: Summary of Special Output Character Processing

Next Character from `WriteTerminal`	**Output to Terminal**
`\n`	`\r\n`

In the 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* then echo to the terminal the special 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 having typed backspace (`'\b'`) or delete (`'\177'`) should be to remove the previous character visibly 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 (moving the cursor again one space to the right), and the third character (backspace) finally moves the cursor back one space to the left. This output 3-character 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 typed backspace or delete character and should not echo anything for this character. You optionally may want to ring the bell (sometimes a beep) in this case, as some systems do; you can do this by sending `'\a'` to the terminal (although whether or not this actually results in an audible bell sound depends on the configuration of the computer from which you are running your project).

As a special case, when you receive a character from a terminal that should normally be added to the input buffer for that terminal but that input buffer is currently full (no room in that input buffer for this new character), then you should simply discard (drop) that new input character; you should add *nothing* to the input buffer for that terminal. And, since you did not add the new input character to the input buffer, you likewise should *not* echo that new character to the terminal. Optionally, in this case you can instead echo a `'\a'` character to the terminal to cause the terminal bell to ring or beep (again, whether or not this actually results in an audible bell or beep sound depends on the configuration of the computer from which you are running your project).

On output for characters from a call to `WriteTerminal`, any newline (`'\n'`) character must be transmitted to the terminal hardware as the sequence of the two characters `'\r'` and `'\n'`, in that order. As described above, to the terminal hardware, the `'\r'` character moves the cursor to the beginning of the line, and the `'\n'` character moves the cursor down one line. The C language, instead, defines the meaning of the single character `'\n'` on output to do both cursor movements in a single character, so a `'\n'` from a user program must be converted to `'\r'` followed by `'\n'` when sent to the hardware.

All other characters for which no special character processing is described above should be treated normally, exactly the same as for any other character.

To summarize, Table 1 shows the major aspects for special *output* character processing for characters being output to the terminal from `WriteTerminal`. And Table 2 shows the major aspects for special *input* character processing for characters received from the terminal from `ReadDataRegister`. *Please refer to the description above for complete details.*

Table 2: Summary of Special Input Character Processing

Character from `ReadDataRegister`	**Add to Input Buffer**	**Echo to Terminal**
`\r`	`\n`	`\r\n`
`\n`	`\n`	`\r\n`
`\b` *or* `\177`	*Remove* one character if the current line in the input buffer is not empty	`\b \b` if you removed a character; or optional `\a` otherwise
any character, but input buffer is full	*nothing*	*nothing*; or optional `\a`

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 your driver read you a single character every time it causes a receive interrupt. Something similar can be said for `WriteTerminal()` and the transmit interrupt.

This behavior means 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. For example, 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
```

Each "`\b`" above represents the user typing a single backspace character on the keyboard. After handling each of each of these backspace characters (as described above in Section 7.1), the resulting contents then in the input buffer for terminal 0 should be:

```
Hi\nWorld\nGood bye
```

Now, consider the execution of the following application code:

```
int len1, len2, len3, len4;

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);
len4 = ReadTerminal(0, buf4, 10);
```

From the specification of `ReadTerminal` and the character processing that the driver must perform, after the application code executes, the variables have the following values (although note that the buffers are *not* terminated by a null character as suggested in the C notation used below):

```
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()` call until the user types a `'\n'` or `'\r'`. Think about why this is the case.

As mentioned above, although the contents of `buf1`, `buf2`, and `buf3` are shown above as they would be written for typical C strings, in reality *there is no null character at the end of each* as would normally be the case in 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 various terminal functions can be called concurrently. Furthermore, user 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 standard requirements.

When a mix of two or more `WriteTerminal` calls occur concurrently to the same terminal, the output characters from these `WriteTerminal` calls must not be interleaved on the screen. For example, if two `WriteTerminal` calls are made currently 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, etc. 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, and must finish all of those characters before beginning the output of characters from some third `WriteTerminal` call, etc.

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 input character processing described above, however, must be displayed without other interleaved output characters from `WriteTerminal` on that terminal. This restriction 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, one at a time, must be filled by characters typed sequentially at the keyboard (after each input line has been terminated by a newline). That is, input characters (after the input line has been terminated by a newline) should go to a single `ReadTerminal` call until that `ReadTerminal` returns, and then on to the next `ReadTerminal` call, and so on. For example, two 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 to terminate this input line. This restriction does not imply that the whole line must be returned to the `ReadTerminal` call: only as many characters as requested should be returned, and the rest of those characters should remain available for the next `ReadTerminal` call to that terminal, 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 with a newline.

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 may 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 must not block when running in an interrupt handler*.

Furthermore, your device driver should allocate and use only a fixed amount of memory for its internal data structures and for each terminal device initialized. In the event that a sequence of terminal operations should require the driver to use more memory than this fixed amount, 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 calling `ReadTerminal`. However, it is unreasonable to require applications to have a pending `ReadTerminal` call at the time every single character arrives. You thus must use buffers inside your device driver to deal with bursts and temporarily "absent" applications, but such buffers must be of finite size.

When adding one input 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 a receive interrupt) you may choose a course of action. You may drop the character as stated above, and you might optionally in this case try to output a bell (`'\a'`) 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).

  Note that when the procedure `func` completes execution, that thread will exit immediately. And if this is the last of all the threads you have created, your entire program will then exit immediately.

- `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 all programs, before the `main()` thread ends, `ThreadWaitAll()` will automatically be called.

- `void Declare_Monitor_Entry_Procedure(void)`

  `Declare_Monitor_Entry_Procedure()` is used declare that the current procedure is an entry procedure of the monitor. This call *must* be used only at the *beginning* of each 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. Before calling any other operation on some condition variable, you *must* have created that condition variable by a call to `CondCreate`. In particular, `CondCreate` returns the condition variable id of the newly created condition variable, and this condition variable id must be used as the argument to any other condition variable operations for which you want to use that condition variable.

- `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. For this project, there should be no reason to need or want to destroy a condition variable, but the `CondDestroy` procedure is present for "completeness."

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. A monitor automatically provides mutual exclusion and also provides operations for concurrent threads to coordinate through condition variables.

Although the C programming language does not provide specific support for monitors, it is possible to create a monitor and to follow the monitor paradigm in C using the threads library procedures defined above. However, doing so requires a little self-disciplined in how the threads library calls are used.

An example of how to correctly write a monitor in 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 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 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 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 your 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()`, and `TerminalDriverStatistics()` at will, but only after the procedure `InitTerminal()` has been called once (and only once) for each terminal used. In addition, some thread 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 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 `ThreadWaitAll()` routine (Section 8.1) is automatically called.

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 Driver

When you have a version of your terminal device driver compiled and you are ready to test it (for example, to test an initial version in which you have implemented some of the required functionality and want to test that before going on to implement and test further pieces of the project), 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 window on your display whenever `InitHardware` is called for some new terminal. This window 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 window from which you invoked the executable file (in the same way as normal output from normal C programs).

As mentioned above in Section 4, the terminals in this project are simulated using the X11 Window System. The window simulating each terminal will open and display on your local screen (on the computer you are using locally, from which you logged on to CLEAR).

For this to work correctly, you will need to log in to CLEAR using `ssh`, telling 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/Mac command line-based `ssh` clients, you probably want to use the `-Y` option on the `ssh` command line to do this (e.g., `ssh -Y ssh.clear.rice.edu`).

Also, for this to work correctly, you will need to be running an X Windows server program on your local machine. Many systems already have this server available, although Microsoft Windows does not come with such a server.

If you are using Microsoft Windows and are new to the X11 Window System, many X11 servers are available for download, including "Cygwin/X" and "Ming," although the Cygwin/X server

(https://x.cygwin.com/) is recommended. Chapter 2 of the Cygwin/X User's Guide describes the steps for downloading and installing Cygwin/X on Microsoft Windows:

```
https://x.cygwin.com/docs/ug/setup.html
```

Cygwin is a system for running just about any Unix/Linux program on top of Microsoft Windows. In addition to this X11 server, there are many other packages available with Cygwin, including a command-line ssh client program (in the Cygwin "openssh" package), which you can install at the same time as installing the Cygwin/X server.

Remember that in all cases, you must be logged in to a CLEAR machine before you can compile, link, or run your terminal device driver. And remember that for the terminal X windows to work correctly, you *must* be running a local X Windows server program on your own local computer, *and must* log in to CLEAR using ssh in a way that allows X Window operations on CLEAR to be forwarded back to your local X Windows server program through this ssh connection.

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.

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 must not copy any of the provided 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 must not copy any of the provided library files to your own directory*.

11 For Students Enrolled in COMP 521 or ELEC 552

For students taking COMP 521 or ELEC 552, the *graduate student* version of this course, you must also write and submit a report addressing issues including the factors affecting the *performance* of the design in your solution to the project and the issues involved in *scaling* the project to larger sizes in larger systems. Students taking COMP 421 or ELEC 421, the *undergraduate student* version of this course, may ignore this section, as this project requirement does not apply to you.

For students taking the graduate version of this course, your report should address issues such as, *but not necessarily limited to*, the general points below:

- What design choices did *you* make in *your* solution to the project that you intended to improve the performance of your solution? Specifically explain these design choices and *why* they should improve performance.
- What design choices *would* you have made in your solution, if you'd had more time to implement them, that could have improved the performance of your solution? Specifically explain these design choices and *why* they should have improved performance.
- What are the biggest issues affecting the ability of *your* solution, and what your solution *could* have been if you'd had more time, to scale to larger sizes and larger systems? Specifically, think about things such as systems with more terminals, more users using the terminals, more output going to the terminals, more input being read from the terminals, etc. Think about factors including both execution time performance and memory consumption.

Your report *must* be limited to a *maximum* of 6 pages, although there is no minimum required length for your report; the report should be as long as you need to specifically and clearly address the issues above. Your report may be formatted in either single-column or double-column format and should use a font size no smaller than 10 points. Your report will be graded based on what you say in it, not on how many pages you fill. In writing your report, please try to think *critically* about what you want to say and to say it clearly and specifically. It is not correct to just give generalized vague statements; and your report should *not* just be a description of your design. You *must* address the issues above.

In addition to the limit of a *maximum* report length of 6 pages, your report *must* be submitted only in PDF format (e.g., *not* in other strange formats such as Microsoft Word).

To submit your report for this project, please include it among the other files you want to turn in for grading for the project, as described below in Section 12. The file name for your report *must* be

```
report.pdf
```

Only reports named exactly (and only) `report.pdf` and that are turned in with the rest of your project, as described below, will be graded. Do not name your report file anything else (not even a minor variant of this file name), and do not attempt to turn it in any other way.

Once again: *Only* reports named exactly (and only) `report.pdf` and that are turned in with the rest of your project, as described below, will be graded.

12 Turning in Your Project

Before turning in your project, please create a file named "`README`" (or "`README.txt`" if you prefer) 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). This directory *must* include everything necessary to build your project, including your `Makefile`.

- 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.

  You may (if you want to) submit your project multiple times or at any time. Only the *most recent* submission you make for this project will be graded. Please refer to the course syllabus for the policy on late work.

13 Use of Piazza

As described previously in class and in the syllabus, this semester, we will be using Piazza for class discussion. Piazza is a web-based platform that will allow you to post questions about the course material, including Lab 1, and to quickly receive answers from me, from the TAs, and from your fellow classmates. With Piazza, you can also easily look to see if someone else has already posted a question about a problem that you may also be experiencing. Instead of sending questions by email to me and/or the TAs, we encourage you to post your questions on Piazza. You should also check Piazza regularly for new questions, answers, and announcements.

When posting questions or answers about the project on Piazza, please be careful about what you post, to avoid inadvertently violating the course's Honor Code policy described in the course syllabus and below in Section 14. *Specifically, please do not post details about your own project solution, such as portions of your source code or details of how your code works, in public questions or answers on Piazza.* If you need to ask a question that includes such details, please make your question private on Piazza by selecting "Instructor(s)" (rather than "Entire Class") for "Post to" at the top, so that only the course instructor and TAs can see your posting.

Most of you have already registered for the course's forum on Piazza, but if not, you will need to do this before you can access this resource. To register for the course's forum on Piazza, go to

```
https://piazza.com/rice/spring2024/comp421
```

Then select "Join as Student" and click on the "Join Classes" button. Note that Piazza requires a "`rice.edu`" email address to register; after registering, you can enter an alternate email address if you want to. You can also access Piazza by clicking on the "Piazza" link in the navigation links in the left column of any page of the course's Canvas site.

14 Honor Code Policy

The Honor Code is a special privilege and responsibility at Rice University. As stated in a student editorial published in the January 20, 2016 edition of *The Rice Thresher*: "As incoming students enter Rice, many are surprised by the degree to which the university's Honor Code extends trust to the student body. ... The privileges of the Honor Code stem from the idea that Rice's aim is not just to instill knowledge in its students, but [to] also help them develop moral character. This idea is fundamental to Rice's identity: Students can and should be held to a high moral character standard."

Specifically, as previously stated, all assignments in this course are conducted under the Rice Honor Code, a code that you pledged to honor when you matriculated at Rice. You are expected to behave in all aspects of your work in this course according to the Rice Honor Code. When in doubt as to whether a specific behavior is acceptable, ask the instructor for a written clarification. *Suspected Honor Code violations on the projects and/or exams in this course will be researched, documented, and reported in extensive detail to the Rice Honor Council or Rice Graduate Honor Council.* For more information on the Rice Honor System, see `http://honor.rice.edu/` and `http://gradhonor.rice.edu/` . In particular, you should consult the Honor System Handbook at

`http://honor.rice.edu/honor-system-handbook/`

This handbook outlines the University's expectations for the integrity of your academic work, the procedures for resolving alleged violations of those expectations, and the rights and responsibilities of students and faculty members throughout the process.

For the programming assignments in this course, students are encouraged to talk to each other, to the TAs, to the instructor, or to anyone else about the assignment. This assistance, however, must be limited to general discussion of the problem; *each student or project group must produce their own solution to each programming project*. Consulting or copying, in any manner, another student's or project group's solution (even from a previous class or previous year) is prohibited, and submitted solutions may not be copied from any source.

Also, for students taking COMP 521 or ELEC 552, for *all* projects, each student must write their *own* report on the project; for the last two projects, for which the project itself will be done in groups of two students, *each* student in the group must still write their *own* report, *not* working together on your report with your group partner. Submitted reports must not be copied, in whole or in part, from any source, and you must fully cite any references you use in preparing your report.

In addition, for all programming assignments, you may not place source code for your project on any *publicly accessible* repository (such as GitHub), including even after the end of the semester; to do so would be a violation of the Honor Code, as it would give aid to other students on the project. Also, if any such public repositories do exist, you may not refer them (or other such sources) in working on or producing your solution for such project.

I want to treat you all in this class as responsible adults. But please be aware that *cheating* on any of the programming projects in this class constitutes a *Rice Honor Code violation*. Submitting a case to the Rice Honor Council requires a lot of work on my part, but it will have a much larger impact on you, your future status as a student at Rice University, and your prospects for a degree from Rice. Please do not make me have to submit any Honor Council cases in this class. This will help both you and me. As I said, I want to treat you all in this class as responsible adults.

15 A Suggested Plan of Attack

Even though the project is not due until Monday, February 12, 2024, I *strongly* recommend you to start on the project *now* and to work on it consistently (although certainly not continuously) throughout this time. I gave you this advise on the first day of classes, and I repeat it here. Please heed this advise now.

There definitely is no need for you to do things in this project in any particular 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 or difficult to understand when they do. One of many reasonable plans to attack this project is as follows:

- While working on the project, pay attention to announcements on Canvas. Any announcements such as clarifications or corrections will be posted there.
- Also, please use Piazza, as described above in Section 13, whether you have questions as you work on the project or not (yet). Here, you can ask questions and find answers to your own questions and questions asked by other students that might be helpful to you too. Please check Piazza regularly.
- 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. Perhaps draw out your design as a diagram on paper. Show the buffers and other data structures, as well as the procedures that will access them.
- Try to 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.
- Think carefully about how you want to use pointers. Think about where it would be best to use an array, or a `struct`, or an array of `structs`.
- 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 still works even if no application calls `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 12.
- 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>   /* COMP 421 threads package definitions */

void pickup_forks(int);
void putdown_forks(int);
void init_philosophers(void);

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" contains the source code for the monitor itself. Each of the two entry procedures, pickup_forks() and putdown_forks(), is actually made into an entry procedure by the Declare_Monitor_Entry_Procedure() call at the beginning of that procedure.

```
#include <stdio.h>

#include <threads.h>   /* COMP 421 threads package definitions */

/*
 * 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 void 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 void
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 void
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 void
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.
 */
void
init_philosophers()
{
    int i;

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