

Concur: A Testing and Debugging Tool for Concurrent Applications Technical Report

• •

This report documents the development of the Concur project: the problems it aims to solve, the design and implementation of the system, how it was tested, and an analysis of how well the system meets its requirements.

Rochelle Palting

Graduate Implementation Project

Table of Contents

TABLE OF CONTENTS.....	2
1 ABSTRACT	4
2 INTRODUCTION.....	4
3 DESIGN SPECIFICATION	5
3.1 REQUIREMENTS	5
3.2 USE CASES.....	7
3.3 USER INTERFACE	7
4 SOFTWARE IMPLEMENTATION.....	8
4.1 FUNCTIONAL COMPONENTS	8
4.2 DATA AND CONTROL FLOW	9
4.3 SCHEDULER.....	10
4.3.1 <i>Key Features:</i>	10
4.3.2 <i>Data Structures</i>	11
5 USING THE FRAMEWORK.....	12
5.1 TEST & DEBUG PROCEDURE.....	12
5.2 RUNNING CONCUR.....	12
5.3 RUNNING THE WRAPPER	13
6 TESTING & ANALYSIS.....	14
6.1 TEST PLAN	14
6.2 TEST SPECIFICATION.....	14
6.2.1 <i>FE-01: Random Mode for Thread Selection</i>	15
6.2.2 <i>FE-02: Interactive Mode for Thread Selection</i>	15
6.2.3 <i>FE-03: Log Trail</i>	15
6.2.4 <i>FE-04: Thread, Lock, and Semaphore Status</i>	15
6.2.5 <i>FE-05: Wrapper</i>	15
6.2.6 <i>FE-06: Deadlock Detection</i>	16
6.2.7 <i>FE-07: Non-Deadlock Error Detection</i>	16
6.2.8 <i>FE-08: Reproduce Failed Test Run</i>	16
6.2.9 <i>FE-09: Startup</i>	16
6.3 TEST CASES.....	16
6.3.1 <i>FE-01: Random Mode for Thread Selection Test Cases</i>	16
6.3.2 <i>FE-02: Interactive Mode for Thread Selection Test Cases</i>	16
6.3.3 <i>FE-03: Log Trail Test Cases</i>	17
6.3.4 <i>FE-04: Thread, Lock, and Semaphore Status Test Cases</i>	17
6.3.5 <i>FE-05: Wrapper Test Cases</i>	17
6.3.6 <i>FE-06: Deadlock Detection</i>	19
6.3.7 <i>FE-07: Non-Deadlock Error Detection</i>	20
6.3.8 <i>FE-08: Reproduce Failed Test Run Test Cases</i>	20
6.4 ANALYSIS.....	21
7 FUTURE WORK	22
8 CONCLUSION.....	22
APPENDIX A. EXAMPLE FOR TEST AND DEBUG PROCEDURE.....	23

List of Figures

FIGURE 1. CONCUR USE CASE DIAGRAM	7
FIGURE 2. CONCUR FUNCTIONAL DECOMPOSITION DIAGRAM.....	9
FIGURE 3. CONCUR DATA AND CONTROL FLOW DIAGRAM	10

List of Tables

TABLE 1. CONCUR FUNCTIONAL REQUIREMENTS.....	5
TABLE 2. CONCUR NON-FUNCTIONAL REQUIREMENTS.....	5
TABLE 3. TEST CONDITIONS AND RESULTS	21

1 Abstract

This report documents the development of the Concur test framework, which was completed over two academic quarters as a graduate implementation project. The beginning of this report provides a discussion of the project including the design specification and software implementation. Following the project discussion is the presentation, discussion, and analysis of the results in which we discuss how the end system fared with our design. Next an analysis of errors is discussed as well as the approach we took to determine the root cause of any errors. A test section that includes the test plan, specification, and cases used to test our system is provided. The summary and conclusion sections provide a look back at the main points of the report. Finally, the Concur test framework source code is provided in the Appendices.

2 Introduction

The primary goal of the graduate software implementation project was to design and implement software that aims to address problems. The problems that Concur was designed to address are those related to testing and debugging multi-threaded applications.

The key problem is that students do not have sufficient resources to adequately determine if their multithreaded program is free from deadlock and/or race conditions. Testing alone is insufficient because many conditions require specific schedules that are not exercised. This leads to the need of a tool that assists students in finding and removing synchronization bugs.

For this project I built Concur, which is a testing and debugging tool for multithreaded programs involving synchronization. Concur was built as a new system making use of existing pieces. There are two primary goals of Concur that both aim at helping students with little or no experience with multithreading. The first goal is to allow a student to test his or her multithreaded program for correctness (i.e. detect bugs). The second goal is to help a student debug the program when an error is present. Concur achieves these goals by providing an environment in which the user has more control over thread scheduling, displaying the current state of each thread, lock and semaphore, detecting synchronization errors such as deadlocks, and reproducing test results. Teachers will also benefit from Concur when used as an instructional aid or for evaluating multithreaded programs.

3 Design Specification

Several high-level requirements have been identified for the system. The functional requirements include allowing control over thread scheduling (through random and interactive modes), capturing programming errors related to multithreading and synchronization, making certain information available to facilitate debugging efforts (e.g. the current state of each thread, lock, and semaphore and information about the failure). The nonfunctional requirements include usability, maintainability, and extensibility.

3.1 Requirements

Several key functional and non-functional requirements have been identified for Concur.

Functional Requirements:

ID	Title	Description
F01	Random mode for scheduling	Concur shall support a random mode for scheduling in which threads are automatically and randomly scheduled.
F02	Interactive mode for scheduling	Concur shall support an interactive mode for scheduling in which the user will select which thread to run next.
F03	Failure trail	Concur shall provide information about the failure.
F04	Thread, lock and semaphore status	Concur shall provide current state of each thread, lock, and semaphore when in interactive mode.

Table 1. Concur Functional Requirements

Non-Functional Requirements:

ID	Title	Description
NF01	Usability	Concur shall be usable for the intended users.
NF02	Maintainability	Concur shall be maintainable
NF03	Extensibility	Concur shall be extensible

Table 2. Concur Non-Functional Requirements

Requirements F01 and F02 require Concur to provide two scheduling modes that grant the user varying degrees of control over how the threads shall be scheduled for execution. For requirement F01: Random mode for scheduling, the order of thread execution will be determined randomly and automatically. The benefit of this method is that it requires no user intervention. The disadvantage, however, is that even after numerous random executions not all of the interesting cases may have been explored.

For requirement F02: Interactive mode for scheduling, Concur shall allow the user to specify the schedule of threads. A positive aspect of this method is that the student is forced to think about synchronization and testing. A drawback, however, is that continuously choosing which thread to execute next can be tedious and only a few schedules would get run.

Requirement F03: Failure trail requires Concur to output information regarding an encountered failure. In random mode, Concur will provide some information about the failure including the schedule that exercised the bug.

Requirement F04: Thread, lock, and semaphore status requires Concur to provide current status of each thread, lock, and semaphore.

The features developed to satisfy requirements F03 and F04 help the user debug and capture most of the programming mistakes related to multithreading and synchronization made by the student. In random mode, the failure information gained from requirement F03 will inform the user that an error occurred. The error could be dependent on the problem, however, such as the restrictions in the unisex bathroom problem. The use of either assertions or separate checkers could help address these cases. Concur could allow a deadlock to manifest itself by letting the program “hang”, but the user would need to distinguish between a long running execution versus one that hangs. For random modes, it suffices to run an evaluation to see if it is effective or not in finding errors. For interactive modes, trying to determine if the error can be reproduced would satisfy requirement F03.

Although Concur in random mode exposes errors, it does not necessarily uncover what failure occurred and for what reason. The user will need to do additional investigation in interactive mode to determine what type of failure occurred (deadlock or race condition) and why the failure transpired. In interactive mode, the user can step through the program execution following the same thread schedule that exercised the bug. By examining the current state of each thread, lock, and semaphore leading up to the failure, the user should gain a clearer understanding with regards to what caused the error. The user can then make necessary updates to the code and rerun the failed schedule until the error has been resolved. The user would repeat this debugging process until the program is error free.

The Concur must also satisfy the quality requirements of usability, maintainability, and extensibility (requirements NF01, NF02, and NF03, respectively). It must be usable for student programmers that have some familiarity with Linux but are new to synchronization. The tool must also be extensible to a more systematic approach and should be able to support different synchronization problems with their checkers.

3.2 Use Cases

Figure 1. Concur Use Case Diagram is the use case diagram for Concur.

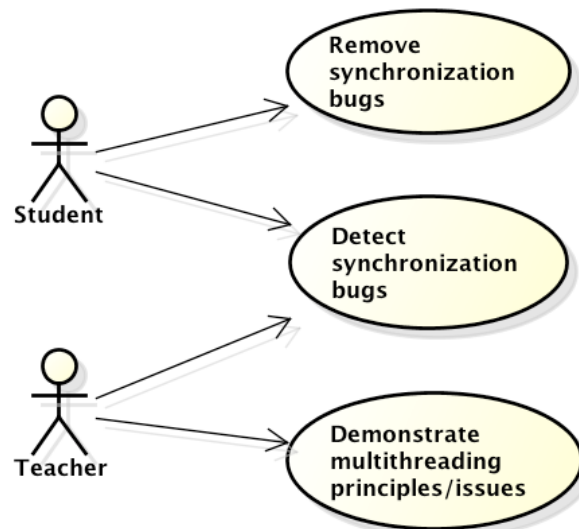


Figure 1. Concur Use Case Diagram

The primary user of Concur is a student who is beginning to learn multithreaded programming and synchronization. The student can use Concur to test the program for errors, using its controlled environment to increase test case coverage and hence increase the chances of detecting bugs. If an error is detected, the student can then use Concur to debug the program. Concur will provide some information about the failure including the schedule that exercised the bug. With this information along with the ability to choose which threads to execute, the student will be equipped to reproduce the execution that caused the error. The ability to reproduce the failed execution is a powerful tool in debugging a multithreaded application that the student typically wouldn't have due to the nondeterministic nature of the operating system's scheduler.

The secondary user of Concur is a teacher who can use the tool for demonstration and for evaluation. A teacher can use Concur in class to demonstrate how deadlock and/or race conditions can occur in a program. The tool can also be used to discover programming errors while evaluating students' programming assignments.

3.3 User Interface

The user will run Concur via a terminal window. The user will provide a program file that adheres to a certain specification that will be tested by the framework. The user can execute Concur that runs once through the multithreaded application, or he can use the wrapper tool that automatically executes Concur multiple, sequential times. Concur allows thread selection (e.g. the next thread to run, the next waiting thread to acquire a lock, or the next waiting thread to use a semaphore) to be done either interactively by the user or automatically and randomly.

4 Software Implementation

This section describes how the system is implemented.

4.1 Functional Components

Figure 2 is a diagram of the functional components comprising the Concur test framework. Components illustrated by a single-outlined box represent existing pieces that were integrated into the system. Components illustrated by a double-outlined box represent new pieces that I have created specifically for this project.

The Multithreaded Application component is the standalone application that addresses a synchronization problem. The sections.c file is the student's implementation of the solution. It contains a representative set of functions commonly found in synchronization solutions: entrySection(...) contains logic a thread performs before entering the critical section; exitSection(...) contains logic a thread performs after exiting the critical section; criticalSection(...) contains logic a thread performs inside a critical section; and remainderSection(...) is performed by a thread in between the exit and entry sections.

SUDS is a C program that allows a user to specify how and where code is to be instrumented within a file. In this project, SUDS injects calls to the scheduler after each statement in the sections.c file. SUDS is an important tool in the framework in that it simulates an erratic scheduler behavior and promotes an increase in conditions being exercised.

The Scheduler is a collection of .cpp files (sched.cpp is the primary file) and is the major component of the framework. It will be discussed more in depth later in this report. The scheduler simulates an operating system scheduler. Instead of allowing the operating system to handle thread scheduling and maintaining the states of threads, locks, and semaphores, the scheduler component takes on these responsibilities. The scheduler also contains the functionality to detect deadlocks in the application.

The Logging component is a C file used by the Scheduler and is responsible for logging messages to an output file.

The Wrapper is a python script that allows Concur to be executed sequentially for a desired number of runs. Multiple runs with the Wrapper creates many random thread schedules to be exercised.

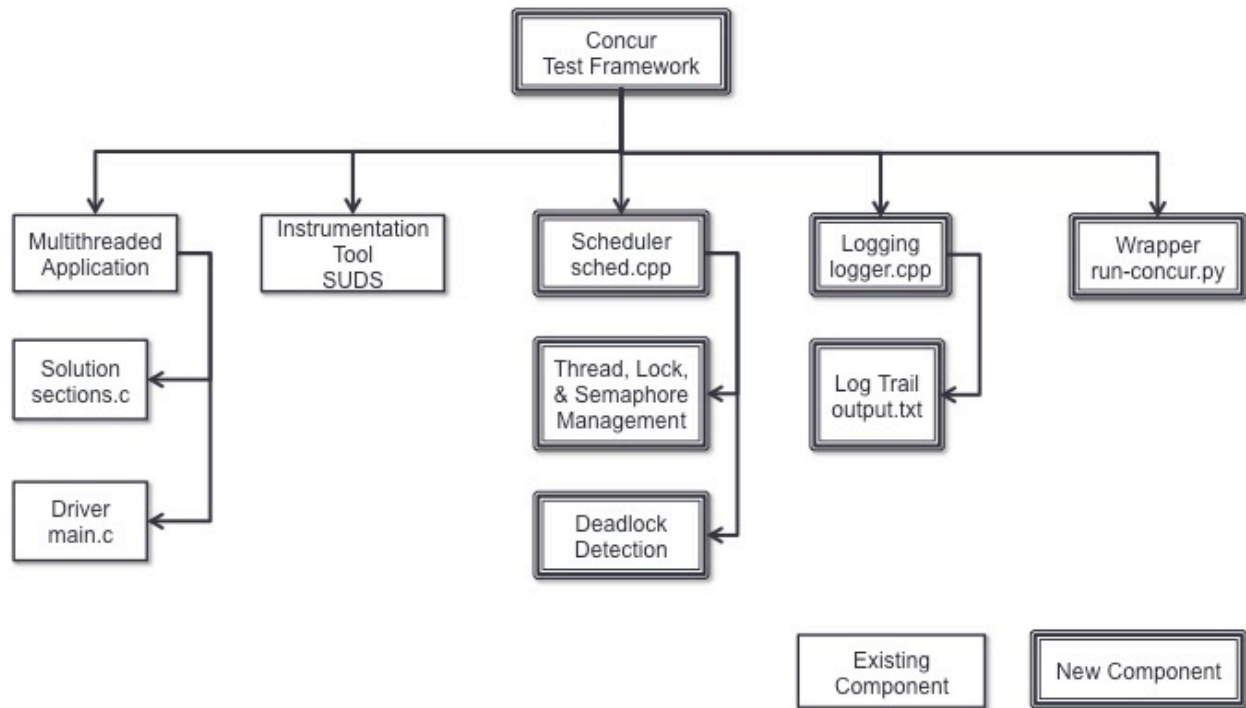


Figure 2. Concur Functional Decomposition Diagram

4.2 Data and Control Flow

Figure 3 illustrates how data and control flows through the Concur framework.

- Beginning at the top left side of the diagram, an Instructor provides the driver of the program, main.c, which will spawn child threads. Each child thread will execute the entry, critical, exit, and remainder functions found in sections.c.
- The student implements his solution in sections.c. Instead of calling the POSIX lock and semaphore functions, it will call the corresponding lock and semaphore functions of the Scheduler.
- SUDS takes sections.c and injections function calls to Scheduler to allow opportunities to switch threads.
- Calls from sections.c to the Scheduler typically involve a state change of at least one thread, lock, or semaphore. The Scheduler is responsible for managing these states.
- When the Scheduler starts the function to examine which thread to run next, it first examines the state of all active threads. An active thread is any thread that has not exited the system. If there are no more active threads available to run (i.e. all threads that have not exited the system are blocked, waiting for either a lock or semaphore), then a deadlock has been detected.
- If logging is turned on for the system, the Logger will output messages to the log trail output file.
- Concur can be run once using the concur executable.
- Concur can be run many times sequentially with random thread scheduling by running the Wrapper. The Wrapper will output the result of the test runs.

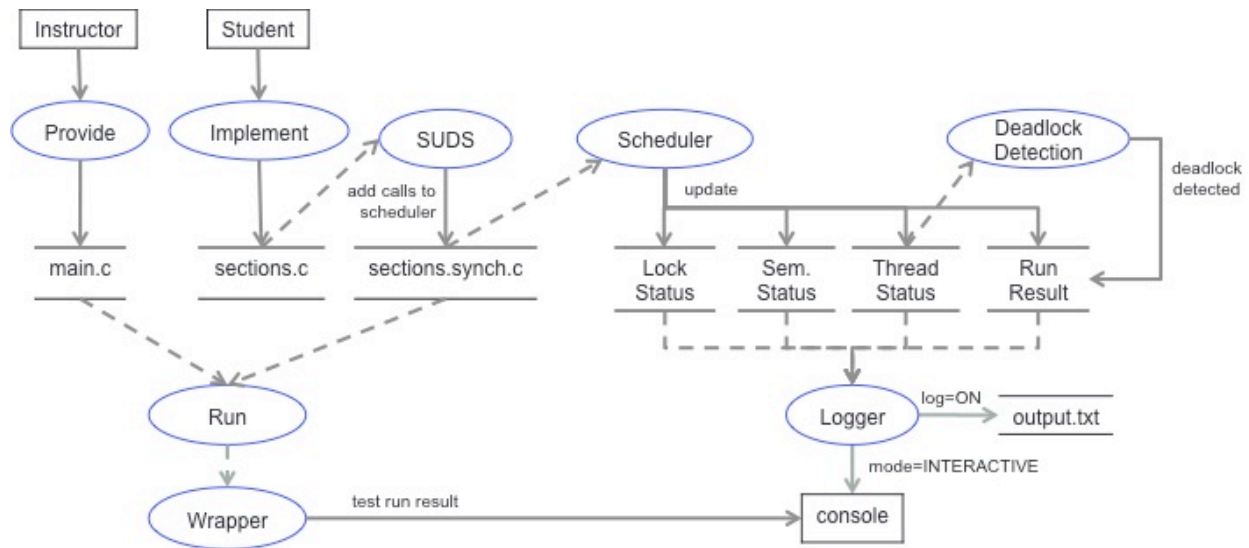


Figure 3. Concur Data and Control Flow Diagram

4.3 Scheduler

The Scheduler is the most important component of the framework. It is here where majority of the framework benefits stem.

4.3.1 Key Features:

The key features of the Scheduler are listed as follows:

One Thread Runs at a Time

The Scheduler permits only one thread to run at a time. This is done by the use of signals. All other threads are forced to wait for a signal before proceeding.

Random & Interactive Scheduler Modes

There are three opportunities for thread selection during the execution of a multithreaded application:

1. When a thread calls the Scheduler's `mutexUnlock(...)` function and there are threads waiting to acquire the lock, a thread from the waiting list will be selected to acquire the lock next.
2. When a thread calls the Scheduler's `semPost(...)` function and there are threads waiting to use the semaphore, a thread from the waiting list will be selected to use the semaphore.
3. When a thread calls the Scheduler's `invokeScheduler(...)` function, the Scheduler chooses the next thread to run amongst the list of active threads. The currently running thread is also considered a candidate for being the next thread to run. Once the Scheduler identifies the next thread, it forces the current thread to wait and signals the next thread to become running.

The manner of how a thread is selected is governed by one of the two Scheduler mode types: Random or Interactive.

Random mode allows candidate threads to be selected randomly. The purpose of the Random mode is to exercise different schedules. For example, given a list of n candidate threads, the Scheduler will choose a random number x between 0 and $(n-1)$ and the x th thread in the list will be selected. The random seed (if not initialized as an input parameter) is generated using the current date and time.

The other Scheduler mode is Interactive mode, which, at each thread selection opportunity, prompts the user to select which thread should run next, acquire a lock, or use a semaphore. The Interactive mode offers the user a more controlled debugging experience and the instructor a demonstration tool.

Application Programming Interface

The Scheduler exposes an Application Programming Interface (API) similar to that of the POSIX thread, lock, and semaphore functions. The API includes the following functions:

- `mutexLock(...)`: If the lock is available, then set the calling thread to holding. If the lock has been acquired by another thread, then set the current thread's state to Waiting, add the current thread to the waiting list for that lock, and call `invokeScheduler(...)` since it is now blocked until the lock becomes available.
- `mutexUnlock(...)`: Release the lock. If there are threads waiting for this lock, select one of these threads to hold the lock and change the selected thread's state from Waiting to Ready.
- `semWait(...)`: If the semaphore is available for additional use (i.e. the value is non-negative), let the calling thread use it. Otherwise, change the calling thread's state to Waiting, add it to the semaphore's waiting list, and call `invokeScheduler(...)` since it is now blocked until the semaphore becomes available.
- `semPost(...)`: Increments the semaphore value. If there are threads waiting to use the semaphore, select one of the waiting threads and change its state from Waiting to Ready.

4.3.2 Data Structures

The Scheduler maintains a collection of threads, locks, and semaphores whose data types are defined herein.

A Thread has the following properties:

- `state`: A thread may be in one of the following states:
 - o Ready: the thread is a candidate to run; it is not waiting for any lock or semaphore
 - o Running: the thread is currently running (only one thread may be running at a time)
 - o Waiting: the thread is waiting for a lock or semaphore to become available
 - o Completed: the thread has exited the system
- `waitingName`: If a thread is waiting for a resource, the resource's name is saved in this variable.

A Lock has the following properties:

- `name`: Provided by the user

- isLocked: flag indicating whether or not it is held by a thread
- holdingThread: id of thread holding
- collection of waiting threads

A Semaphore has the following properties:

- name: Provided by the user
- value: semaphore value (non-negative values indicate not available)
- list of waiting threads

5 Using the Framework

5.1 Test & Debug Procedure

The user would typically go through the following steps in order to test and debug his application using the framework:

1. Make the project (compiles source code and creates executables).
2. Use the Wrapper to execute multiple test runs. This runs multiple random schedules.
3. Observe the status at the end of each test run. (proceed to the next step if a run failed).
4. Run Concur with the same parameters as the failed test run, but set logging to be turned on. Use Random mode for faster results. Use Interactive mode to step through execution and examine status in real time via the console.
5. After the failed run is reproduced, examine the log trail in the output file for the specific error.
6. Update the solution with the fix to the error.

Repeat steps 4-6 as necessary until the bug is removed and the test run passes.

5.2 Running Concur

Input: Concur accepts a multithreaded program to be tested as input (sections.c) along with the following control parameters: number of threads, number of rounds, scheduler mode (Random or Interactive), logging mode (On or Off), and random seed. The random seed is only used during Random mode. If none is specified, one will be automatically generated.

Example command for running Concur with 5 threads, 2 rounds for each thread, in Random mode, with logging turned off, and setting the random seed to 1234567:

```
> ./concur.synch.exe 5 2 RANDOM OFF 1234567
```

Example command for running Concur with 5 threads, 2 rounds for each thread, in Interactive mode, with logging turned on, and no random seed since it is not used/relevant in Interactive mode (the user performs all the thread selections):

```
> ./concur.synch.exe 5 2 INTERACTIVE ON
```

Output: At startup Concur prints to the console the control parameters used for the run. If mode is Interactive, status and use prompts will be displayed to the screen. Otherwise only

a failure or success message will be output to the screen. A log trail in an output text file will be produced only if logging is turned on.

5.3 Running the Wrapper

Input: The wrapper requires the following control parameters: number of runs, number of threads, and number of runs.

Example command for running the Wrapper for 10 runs, 5 threads, 2 rounds for each thread:

```
>./run-concur.py --runs=10 --threads=5 --rounds=2
```

By default, the Wrapper invokes concur with the mode set to RANDOM and logging turned off.

Output: The wrapper will echo the control parameters used to invoke Concur for each test run. If a test run fails, an error message will be displayed to the screen and the script will cease to run additional test cases. If all the test cases pass, a success message will be output to the screen.

6 Testing & Analysis

This section includes the test plan, test specification, and test cases used to test Concur.

6.1 Test Plan

The following describes the test of the Concur test framework. In each execution of Concur, a single user-provided sections.c file will be tested with the following control parameters set by the user or inferred by another control parameter: number of test runs, number of threads, number of rounds, scheduler mode, and logging mode.

To ensure operation of the system according to the initial requirements and design specification, the following items must be tested:

1. Ability to run in random scheduler mode
2. Ability to run in interactive scheduler mode
3. Ability to view thread, lock, and semaphore status
4. Ability to log results
5. Ability to detect deadlocks
6. Ability to run sequential test runs in random mode

6.2 Test Specification

This section provides more detail regarding the function to be tested and the input to be used.

Inputs:

- sections.c file: This file is to be provided by the user and must include the following functions to be called by main.c:
 - o initGlobals(int id): initialize variables global to the application including locks and semaphores
 - o entrySection(int id)
 - o criticalSection(int id)
 - o exitSection(int id)
 - o remainderSection(int id).
- Number of threads: This is a positive integer that indicates how many child threads to execute within the program.
- Number of rounds: This is a positive integer that indicates how many rounds each thread will run. Each round consists of a single execution of each of the entry, critical, exit, and remainder section functions.
- Scheduler mode: This is one of the following strings: "random" or "interactive". This input is not case sensitive. If scheduler mode is "random", thread selection will be performed randomly. If scheduler mode is interactive, thread selection will be performed through user prompts. Scheduler mode is set to "random" when running via the Wrapper.
- Logging mode: This is one of the following strings: "on" or "off". This is not case sensitive. If logging mode is on, an output file will be produced and the test run

status will be written to it. If logging mode is on, no log file will be created. Logging mode is set to “off” when running via the wrapper.

- Number of runs (applicable only with the Wrapper): This is a positive integer indicating how many Concur executions to run.

6.2.1 FE-01: Random Mode for Thread Selection

If scheduler mode is “random”, thread selection shall be done randomly.

If random seed is -1 and scheduler mode is random, Concur shall automatically generate the random seed.

6.2.2 FE-02: Interactive Mode for Thread Selection

If scheduler mode is “interactive”, Concur shall prompt the user for thread selection. The random seed parameter is ignored.

6.2.3 FE-03: Log Trail

If logging mode is on, Concur shall open a new output file and log status to it during program execution. Concur shall safely close the output file upon exiting the program.

If logging mode is off, no output file shall be created.

6.2.4 FE-04: Thread, Lock, and Semaphore Status

If logging mode is “on”, thread, lock, and semaphore status shall be output to log file at each call to `invokeScheduler(...)`.

6.2.5 FE-05: Wrapper

FE-05.1: If any of the Wrapper inputs are invalid, the Wrapper shall return an error message to the console and exit.

FE-05.2: The Wrapper will continue to execute additional Concur test runs provided each test run completes without error.

FE-05.3: If Concur returns an error for the current test run, the Wrapper shall output an error message to the screen and exit. No additional test runs will be performed once a failed test run is encountered.

FE-05.4: If Concur hangs due to a non-deadlock error, the Wrapper will also hang. The user will have to kill the process (CTRL+C).

6.2.6 FE-06: Deadlock Detection

If all active threads are blocked waiting for a lock or semaphore, Concur shall output an error message indicating that a deadlock has occurred and exit the application with an error exit status.

6.2.7 FE-07: Non-Deadlock Error Detection

If Concur encounters a non-deadlock error, the user shall be able to examine the log trail and determine at what point during execution the error was encountered.

If all threads have completed without error, Concur shall return with a success status.

6.2.8 FE-08: Reproduce Failed Test Run

If Concur receives inputs identical to control parameters of a failed test run, Concur shall reproduce the same failure.

6.2.9 FE-09: Startup

Upon startup, Concur shall output to the screen the values of the following control parameters used for the run: number of threads, number of rounds, scheduler mode, logging mode, and random seed value.

6.3 Test Cases

The following test cases identify how each component of the system identified in the test plan and quantified in the test specification is to be tested.

6.3.1 FE-01: Random Mode for Thread Selection Test Cases

ID:	FE01_TC1
TCS Name:	Random mode
Description:	This test will determine if Random mode is supported.
Expected Outcome:	Thread selection via Random mode is supported.
Status:	Success

6.3.2 FE-02: Interactive Mode for Thread Selection Test Cases

ID:	FE02_TC1
TCS Name:	Interactive mode
Description:	This test will determine if Interactive mode is supported.
Expected Outcome:	Thread selection via Interactive mode is supported.
Status:	Success

6.3.3 FE-03: Log Trail Test Cases

ID:	FE03_TC1
TCS Name:	Log Trail per logging mode
Description:	This test will determine if the log trail output file is correctly outputted based on the log mode.
Expected Outcome:	Log trail logged to output file only if log mode is turned on.
Status:	Success

6.3.4 FE-04: Thread, Lock, and Semaphore Status Test Cases

ID:	FE04_TC1
TCS Name:	Thread, Lock, and Semaphore Status is Outputted
Description:	<p>This test will determine whether thread, lock and semaphore status is outputted to the log file with the required fields.</p> <p>Input:</p> <ul style="list-style-type: none">- sections file: sections1.c- ./concur.synch.exe 5 2 RANDOM ON 1338753699- number of threads = 5- number of rounds = 2- schedule mode = RANDOM- log mode = ON- seed = 1338753699
Expected Outcome:	Thread, lock, and semaphore status will be outputted to the log file with the required fields.
Status:	Success: The status was outputted to the log file.

6.3.5 FE-05: Wrapper Test Cases

ID:	FE05_TC1
TCS Name:	Wrapper Receives Invalid Input
Description:	This test will determine the Wrapper's response to invalid input parameters.
Expected Outcome:	The Wrapper will output an error message and exit. No test runs will be executed.
Status:	SUCCESS

ID:	FE05_TC2
TCS Name:	Wrapper with successful test runs.
Description:	<p>This test will determine if the wrapper will continue to run the subsequent test run if the current test run completes without error.</p> <p>Input to Wrapper:</p>

	<ul style="list-style-type: none"> - Sections file: sections3.c - number of test runs = 10 - number of threads = 5 - number of rounds = 2
Expected Outcome:	SUCCESS
Status:	<p>Success. All test runs were completed without error.</p> <p>Console output from Wrapper:</p> <pre>[paltingr@css2 concur]\$./run-concur.py --runs=10 --threads=5 --rounds=2 ./concur.synch.exe 5 2 RANDOM OFF Run 1: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338707229 Run 2: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338707238 Run 3: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338707246 Run 4: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338707255 Run 5: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338707266 Run 6: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338707277 Run 7: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338707287 Run 8: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338707296 Run 9: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338707307 Run 10: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338707316 ----All tests PASSED</pre>

ID:	FE05_TC3
TCS Name:	Wrapper encounters deadlock.
Description:	<p>This test will determine how the Wrapper responds when one of the test runs encounters a deadlock error (i.e. Concur exits with an error status).</p> <p>Input:</p> <ul style="list-style-type: none"> - sections file: sections1.c - ./run-concur.py --runs=10 --threads=5 --rounds=2 - number of runs = 10 - number of threads = 5 - number of rounds = 2
Expected Outcome:	The Wrapper will output an error message and exit. No additional test runs will be executed.
Status:	Success. Deadlock encountered. Concur returned with an error status code. Wrapper output a message that the test run failed

	<p>and the script was exited.</p> <pre>[paltingr@css2 concur]\$./run-concur.py --runs=10 --threads=5 --rounds=2 ./concur.synch.exe 5 2 RANDOM OFF Run 1: -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1338763589 ----Test FAILED at round 1.</pre>
--	--

ID:	FE05_TC4
TCS Name:	Wrapper Encounters Non-Deadlock Error
Description:	<p>This test will determine how the Wrapper responds when one of the test runs fails due to a non-deadlock error (Concur hangs and fails to return).</p> <p>The input was kept simple for this test initially to verify that sections could pass for a single thread.</p> <p>Input:</p> <ul style="list-style-type: none"> - sections file: sections2.c - number of test runs = 1 - number of threads = 1 - number of rounds = 1
Expected Outcome:	If Concur hangs and fails to return, the Wrapper will also hang. The user will have to kill the process to exit.
Status:	Success. During the first and only test run, Concur failed to return after waiting approximately one minute. The user killed the process to exit.

6.3.6 FE-06: Deadlock Detection

ID:	FE03_TC1
TCS Name:	Deadlock Detection
Description:	This test will determine
Expected Outcome:	<p>This test will determine whether Concur can detect deadlock within a program.</p> <p>Inputs:</p> <ul style="list-style-type: none"> - sections file: sections1.c - ./concur.synch.exe 5 2 RANDOM ON 1338753699 - number of threads = 5 - number of rounds = 2 - schedule mode = RANDOM - log mode = ON

	- seed = 1338753699
Status:	SUCCESS

6.3.7 FE-07: Non-Deadlock Error Detection

ID:	FE07_TC1
TCS Name:	Non-Deadlock Error Detection
Description:	<p>This test will determine whether the user will be able to determine at what point during program execution a non-deadlock error occurred.</p> <p>Input:</p> <ul style="list-style-type: none"> - sections file: sections.2.c - number of threads = 1 - number of rounds = 1 - scheduler mode = RANDOM - log mode = ON
Expected Outcome:	<p>If Concur hangs, the last log trail entries should provide enough information for the user to determine at what point the program hung. In particular, the sections.c line number shall be provided.</p>
Status:	<p>Success. After Concur hung, I examined the last entry in the log trail and saw that the line corresponded to a call to function personEnterRestroom(id, isFemale). From examining the first line of this function, it is evident that there is an infinite loop error when isFemale=true (isFemaleUsingBathroom is initialized to false):</p> <pre>while(isFemale != isFemaleUsingBathroom);</pre> <p>Concur output:</p> <pre>[paltingr@css2 concur]\$./concur.synch.exe 1 1 RANDOM ON 1338742293 output/output-20120603_10:38:56.txt -- numThreads=1, numRounds=1, schedMode=RANDOM, logMode=ON, seed=1338742293 [paltingr@css2 concur]\$</pre>

6.3.8 FE-08: Reproduce Failed Test Run Test Cases

ID:	FE08_TC1
TCS Name:	Reproduce failure with Concur
Description:	This test will determine whether Concur can duplicate a failure encountered by the Wrapper.
Expected Outcome:	Concur will duplicate a failure encountered by the Wrapper, using the same control parameters.

Status:	Success
---------	---------

Supplemental Notes:

Testing was conducted by testing the Wrapper with four student submissions from Fall 2011 for the Unisex Bathroom problem.

Table 3. Test Conditions and Results Summarizes the test conditions and results.

Test File	Wrapper Parameters			Result
	# Runs	# Threads	# Runs	
sections1.c	10	5	2	Deadlock detected at 1 st run.
sections2.c	10	5	2	Programming error detected (infinite loop) due to improperly formed spinlock.
sections3.c	100	5	2	No deadlock detected.
sections4.c	10	5	2	Deadlock detected at 1 st run.

Table 3. Test Conditions and Results

6.4 Analysis

Following the completion of testing, it is evident that Concur is successful at detecting deadlocks, supports the execution of many test runs via the Wrapper, and it satisfies all of its functional requirements: supports both Random and Interactive modes; maintains and displays thread, lock, and semaphore status; and produces a log trail of program execution.

The observed benefits that Concur provides are as follows:

- Deadlock detection: If all remaining active threads are blocked waiting for either a lock or semaphore, then a deadlock has occurred.
- Controlled environment:
 - Only a single thread can run at once.
 - Next ready thread to run, next waiting thread to acquire lock or use semaphore is deterministic.
- Increased schedule coverage:
 - Wrapper allows automated execution of randomly produced schedules
 - Interactive mode allows user to choose schedule
- The framework, with minimal work of the instructor, can be used for a variety of synchronization exercises.
- The library (the scheduler and logging components) can be used for other multithreaded programs.

7 Future Work

Areas that could be improved or augmented within Concur include:

- Improve performance: Running the Wrapper isn't as fast as I as a user would like it to be. For example, running the Wrapper with 100 runs, 5 threads, and 2 rounds took approximately 45 minutes to complete (almost 30 seconds per run).
- Additional analysis can be done to determine if more random schedule test runs is equivalent to higher confidence that the program is error free.
- Add functionality to detect mutual exclusion violations.

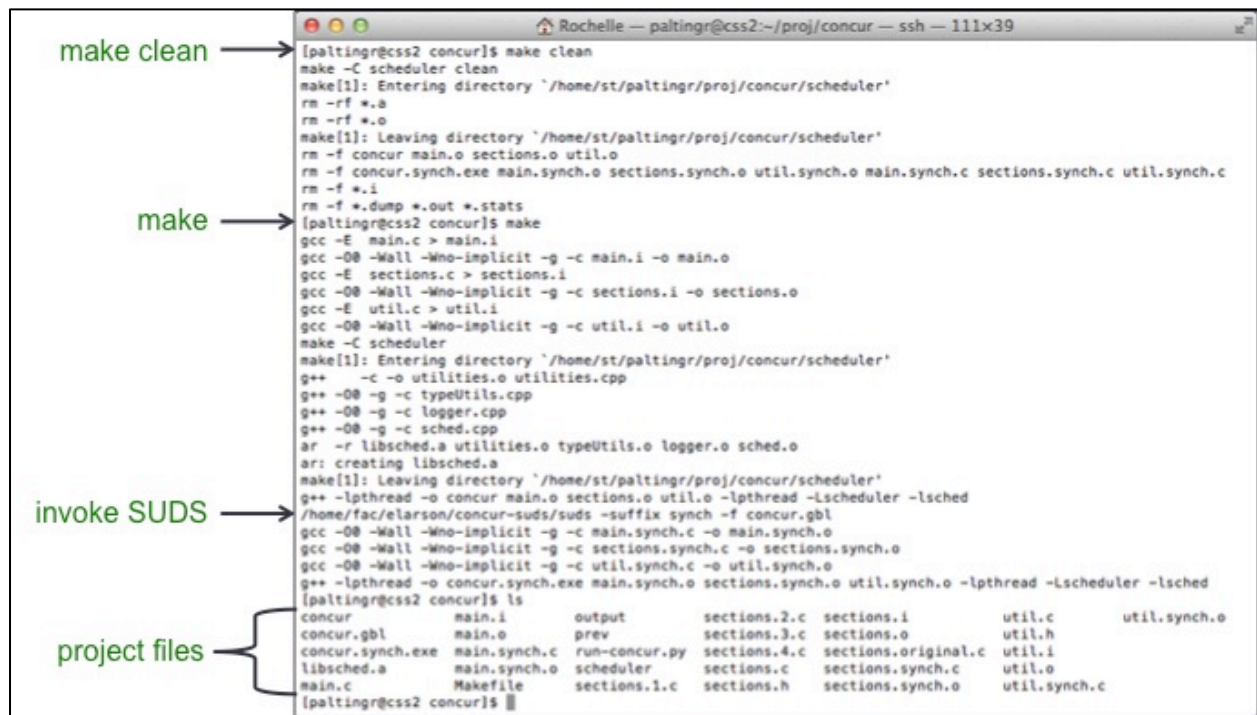
8 Conclusion

Concur has the capabilities to help students test and debug their multithreaded programs by providing an environment with more deterministic thread management, deadlock detection, increased multiple schedule coverage, and a view into thread, lock, and semaphore status.

Appendix A. Example for Test and Debug Procedure

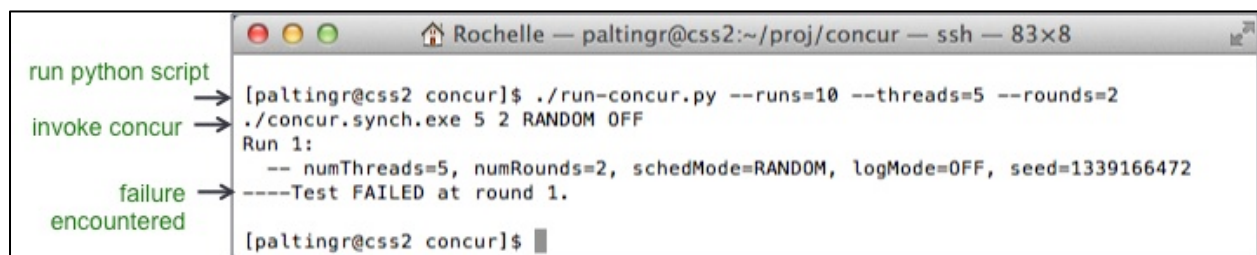
The screenshots below illustrate the process a user may follow to test and debug his multithreaded application. Explanations of each step can be found earlier in the report.

1. Make the Project



```
[paltingr@css2 concur]$ make clean
make -C scheduler clean
make[1]: Entering directory `/home/st/paltingr/proj/concur/scheduler'
rm -rf *.a
rm -rf *.o
make[1]: Leaving directory `/home/st/paltingr/proj/concur/scheduler'
rm -f concur main.o sections.o util.o
rm -f concur.synch.exe main.synch.o sections.synch.o util.synch.o main.synch.c sections.synch.c util.synch.c
rm -f *.i
rm -f *.dump *.out *.stats
[paltingr@css2 concur]$ make
gcc -E main.c > main.i
gcc -O0 -Wall -Wno-implicit -g -c main.i -o main.o
gcc -E sections.c > sections.i
gcc -O0 -Wall -Wno-implicit -g -c sections.i -o sections.o
gcc -E util.c > util.i
gcc -O0 -Wall -Wno-implicit -g -c util.i -o util.o
make -C scheduler
make[1]: Entering directory `/home/st/paltingr/proj/concur/scheduler'
g++ -c -o utilities.o utilities.cpp
g++ -O0 -g -c typeUtils.cpp
g++ -O0 -g -c logger.cpp
g++ -O0 -g -c sched.cpp
ar -r libsched.a utilities.o typeUtils.o logger.o sched.o
ar: creating libsched.a
make[1]: Leaving directory `/home/st/paltingr/proj/concur/scheduler'
g++ -lpthread -o concur main.o sections.o util.o -lpthread -lscheduler -lsched
/home/fac/clarson/concur-suds/suds -suffix synch -f concur.gbl
gcc -O0 -Wall -Wno-implicit -g -c main.synch.c -o main.synch.o
gcc -O0 -Wall -Wno-implicit -g -c sections.synch.c -o sections.synch.o
gcc -O0 -Wall -Wno-implicit -g -c util.synch.c -o util.synch.o
g++ -lpthread -o concur.synch.exe main.synch.o sections.synch.o util.synch.o -lpthread -lscheduler -lsched
[paltingr@css2 concur]$ ls
concur      main.i      output      sections.2.c sections.i    util.c      util.synch.o
concur.gbl  main.o      prev         sections.3.c sections.o    util.h
concur.synch.exe main.synch.c run-concur.py sections.4.c sections.original.c util.i
libsched.a  main.synch.o scheduler    sections.c  sections.synch.c util.o
main.c      Makefile    sections.1.c sections.h  sections.synch.o util.synch.c
[paltingr@css2 concur]$
```

2. Run Wrapper



```
[paltingr@css2 concur]$ ./run-concur.py --runs=10 --threads=5 --rounds=2
./concur.synch.exe 5 2 RANDOM OFF
Run 1:
-- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1339166472
----Test FAILED at round 1.
[paltingr@css2 concur]$
```

3. Run Concur

```
[paltingr@css2 concur]$ ./run-concur.py --runs=10 --threads=5 --rounds=2
./concur.synch.exe 5 2 RANDOM OFF
Run 1:
  -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=OFF, seed=1339166472
----Test FAILED at round 1.

[paltingr@css2 concur]$ ./concur.synch.exe 5 2 RANDOM ON 1339166472 ← invoke concur
output/output-20120608_07:42:16.txt
  -- numThreads=5, numRounds=2, schedMode=RANDOM, logMode=ON, seed=1339166472
[4] [-1] - No ready threads available to schedule; possible DEADLOCK detected. Exiting.
[paltingr@css2 concur]$
```

failure reproduce

4. Examine Log Trail

```
=====
[4] [62] - Selected next thread to run= 4, which is the same as current thread.
=====
THREAD Status: numThreads: 5
|ID|      |STATUS|  |WAITING_NAME|
|0|      |Waiting_Lock| |limit_male|
|1|      |Waiting_Lock| |limit_female|
|2|      |Waiting_Lock| |limit_male|
|3|      |Waiting_Lock| |numMutex|
|4|      |Running|
|

LOCK Status: numLocks: 3
|NAME|      |LOCKED|  |HOLDING_THREAD|  |WAITING_THREADS|
|numMutex|  |true|    |4|                |{0}3|
|limit_female| |true|    |3|                |{0}1|
|limit_male|  |true|    |4|                |{0}0, {1}2|
|

SEMAPHORE Status: numSens: 2
|NAME|      |VAL/INITIAL|  |WAITING_THREADS|
|women|     |0/1|          |
|men|       |0/1|          |
|

=====
[4] [63] - Selected next thread to run= 4, which is the same as current thread.
[4] [-1] - semWait:  sen women is taken and has value=1; thread 4 will be added to thread waiting list.
=====
THREAD Status: numThreads: 5
|ID|      |STATUS|  |WAITING_NAME|
|0|      |Waiting_Lock| |limit_male|
|1|      |Waiting_Lock| |limit_female|
|2|      |Waiting_Lock| |limit_male|
|3|      |Waiting_Lock| |numMutex|
|4|      |Waiting_Sem|  |women|
|

LOCK Status: numLocks: 3
|NAME|      |LOCKED|  |HOLDING_THREAD|  |WAITING_THREADS|
|numMutex|  |true|    |4|                |
|limit_female| |true|    |3|                |{0}1|
|limit_male|  |true|    |4|                |{0}0, {1}2|
|

SEMAPHORE Status: numSens: 2
|NAME|      |VAL/INITIAL|  |WAITING_THREADS|
|women|     |1-1/1|        |{0}4|
|men|       |0/1|          |
|


=====
[4] [-1] - No ready threads available to schedule; possible DEADLOCK detected. Exiting.
=====
End Program Trail
838,1 80t
```

last statement executed is line 63 of sections.c made by thread 4

status shows all threads blocked

failure due to deadlock

5. Examine sections.c file (student's solution)



The screenshot shows a code editor window titled "Rochelle — paltingr@css2:~/proj/concur — ssh — 67x46". The code is in C and defines two functions: `initGlobals()` and `entrySection(int id)`. The `entrySection` function has a `TODO` comment and contains logic for managing a shared resource with mutexes and semaphores. A green annotation "deadlock encountered at line 63" with an arrow points to line 63 in the `entrySection` function, which is the `semWait(id, &women);` call within the `if(menCount == 1)` block. The code is as follows:

```
void
initGlobals()
{
    // LEAVE THIS STATEMENT
    pthread_mutex_init(&numMutex, NULL);
    pthread_mutex_init(&limit_female, NULL);
    pthread_mutex_init(&limit_male, NULL);
    sem_init(&men, 0, 1);
    sem_init(&women, 0, 1);
    menCount = 0;
    womenCount = 0;
    addLock("numMutex", &numMutex);
    addLock("limit_female", &limit_female);
    addLock("limit_male", &limit_male);
    addSemaphore("women", 1, &women);
    addSemaphore("men", 1, &men);
}

void
entrySection(int id)
{
    bool isFemale = id % 2;
    // TODO: Complete this function
    if(isFemale) {
        mutexLock(id, &limit_female);
        semWait(id, &women);
        mutexLock(id, &numMutex);
        womenCount++;
        if(womenCount == 1)
            semWait(id, &men);
        mutexUnlock(id, &numMutex);
        semPost(id, &women);
        mutexUnlock(id, &limit_female);
    }
    else {
        mutexLock(id, &limit_male);
        semWait(id, &men);
        mutexLock(id, &numMutex);
        menCount++;
        if(menCount == 1)
            semWait(id, &women);
        mutexUnlock(id, &numMutex);
        semPost(id, &men);
        mutexUnlock(id, &limit_male);
    }
}
```

63,1 36%