CPSC 3500 Computing Systems

MDAT Guide

Programming with MDAT

To use MDAT, there are two restrictions. First, you must use the MDAT synchronization primitives which are similar to the pthreads functions, except that they have different names:

MDAT Lock and Semaphore Functions

Pthreads function name	***MDAT function name***
`pthread_mutex_init`	`mdat_mutex_init`
`pthread_mutex_lock`	`mdat_mutex_lock`
`pthread_mutex_unlock`	`mdat_mutex_unlock`
`sem_init`	`mdat_sem_init`
`sem_wait`	`mdat_sem_wait`
`sem_post`	`mdat_sem_post`

```
void mdat_mutex_init(const char *name, pthread_mutex_t *lock,
  pthread_mutexattr_t *attr);
void mdat_mutex_lock(pthread_mutex_t *lp);
void mdat_mutex_unlock(pthread_mutex_t *lp);

void mdat_sem_init(const char *name, sem_t *sem, int pshared, int value);
void mdat_sem_wait(sem_t *sp);
void mdat_sem_post(sem_t *sp);
```

The argument lists for the MDAT functions are identical to corresponding pthreads function except for `mdat_mutex_init` and `mdat_sem_init`. These functions have an additional `name` argument that serves as the name of that entity. That name is subsequently used in the debugging trace in MDAT (more on that later).

The second restriction is that you must use the C programming language. Since the programming in this assignment is rather limited, the limitation is not as imposing as it might first appear. Here is a brief, but not exhaustive, list of differences when using C instead of C++:

- All local variables, if you use any, must be declared at the beginning of the function before any other statements.
- For output, you must use the function `printf` instead of using `cout`. Your implementation should not print any additional output when submitted.
- C does not allow for type `bool`. Instead, use an integer: zero is false, nonzero is true.
- C does permit any of the following: classes, templates, and pass by reference parameters. You would not need to use these constructs anyway for this assignment.

Compiling

Use the provided Makefile to compile your program; just type `make` in the current directory. Do not edit this Makefile.

In some cases, the permissions are not properly set when copying the files. If you get a "permission denied" errors with `suds` (which is run automatically in the Makefile), execute the following command in the directory where `suds` is located:

```
chmod +x suds
```

Running the Program

If compiled successfully, two executables `mdatA` and `mdatB` will be created for parts a and b respectively. The execution of both executables are the same. There are five command line flags, two are required and three are optional:

```
Required flags:
-t <num-threads>  (or --threads)     Number of threads
-r <num-rounds>   (or --rounds)      Number of rounds

Optional flags:
-i                (or --interactive) Run in interactive mode
-o <filename>     (or --output)      Create output trace in specified file
-s <number>       (or --seed)        Sets the random number generator seed
```

Notes:

- The required flags specify the number of threads and the number of rounds. Each thread is automatically assigned a type (based on the assignment in class, the different types are split evenly among the threads).
- Use the "`-o`" flag to create a debugging trace.
- By default, the scheduler will randomly select threads and is seeded by the current time. The seed is printed out at the beginning of each run. To rerun a program with a specified seed and obtain the exact same schedule, then use the "`-s`" option. Note: The schedule will only be the same for a particular executable with the same number of threads and rounds. If you recompile your program and/or use a different number of threads or rounds, the schedule will likely be different.
- Use the "`-i`" flag to run MDAT in interactive mode where the user gets to choose which thread runs next.

Example:

```
./mdatA –t 16 –r 10 –o trace.mdat
```

Runs `mdatA` with 16 threads for 10 rounds. The debugging trace is placed in the file `trace.mdat`.

Using MDAT

MDAT takes over the scheduling responsibilities from the OS by making sure that only one thread is active at a time. During the compilation process, MDAT will instrument the files `sectionsA.c` and `sectionsB.c` to add calls to invoke the scheduler (called `mdat_invoke_scheduler`) after each statement in the program. Furthermore, complex statements are split into two or more basic statements adding a call to `mdat_invoke_scheduler` after each one. Unless you are using the interactive mode, the scheduler selects which thread to run next randomly. In addition, all other thread selection choices are made randomly. The act of randomly scheduling plus the myriad of scheduler invocations causing many context switches more aggressively tests your program than a normal pthread program.

MDAT automatically checks for deadlock and mutual exclusion violations while the program runs. The mutual exclusion checker will check for the properties of the problem. The checker for `mdatA` will be used to check the properties for part a and the checker for `mdatB` will check the properties for part b. When an error occurs, the program will stop execution.

To facilitate testing of many schedules (important in multithreaded testing), a script is provided called `run-mdat.py` which will run your implementation a user-specified number of times invoking the checker after each run. If a run fails (either to deadlock or due to checker error), the script will stop. The trace is stored in `trace.mdat`. The trace can be opened in Emacs or using the MDAT trace analyzer (described later in this guide) to help debug the problem.

The command line for `run-mdat.py` is as follows:

```
./run-mdat.py -R <runs> -t <threads> -r <rounds> -m <mode (a or b)> [-c]
```

These four command line switches are required:
- The `–R` switch is used to specify the number of runs – the number of times to run your program.
- The `–t` and `–r` switches are used to specify the number of threads and number of rounds. These parameters are identical to the `–t` and `–r` switches for `mdatA` and `mdatB`.
- The `–m` switch is used to run either part a or b. Specifying ‘`a`’ will direct the script to run `mdatA` and to use ‘`a`’ when running the checker. Similarly, specifying ‘`b`’ will direct the script to run `mdatB` and to use mode ‘`b`’ when running the checker.

Note: If you get a “permission denied” error with `run-mdat.py`, execute the following in the directory where `run-mdat.py` resides: `chmod +x run-mdat.py`

By default, `run-mdat.py` will stop anytime deadlock or a mutual exclusion error is detected. There is an optional `–c` flag that will direct `run-mdat.py` to continue past any errors.

Here is a good command line to start testing:

```
./run-mdat.py –R 10 –t 6 –r 5 –m a
```

When you are confident that it works, increase the number of threads. When testing part b, use at least 15 threads. Before submitting, run the script with 100 runs for both modes.

CAUTION: MDAT only randomly generates schedules. It is computationally infeasible to test all of the number of schedules. Your solution may only fail on a specific schedule that is not generated. It is important to manually inspect that your solution is correct. You can try out your own schedules using the interactive mode, described later in this guide.

One of the requirements for this assignment is that the synchronization cannot be overly restrictive. For instance, a single lock that only permits one thread in the critical section will meet exclusion requirements. To ensure that your implementation allows for some concurrency, MDAT will print a warning if there was never a time where two or more threads in the critical section concurrently. Since schedules are random, it is possible that you may get this warning on an acceptable implementation, especially when running with a small number of threads. If you are getting this warning every time when running with sufficiently large thread counts, it is highly recommend you inspect your program more carefully, with the help of interactive mode, to make sure you are not accidentally being more restrictive than necessary.

An excerpt of a trace produced by MDAT is shown on the next page. For consistency, the trace and interactive outputs are identical except when interactive mode prompts the user for a thread. The trace is structured by alternating one or more informational messages followed by a status table. The informational messages indicate which thread was run, the section (if a new section was entered), and a message indicating what happened when the scheduler was invoked. Most of the time, the scheduler was invoked due to a `mdat_invoke_scheduler` call. No state is changed excepted for the location field of the running thread. The scheduler is also invoked anytime a synchronization call is made. In these cases, the message will indicate what happened.

The status table is divided into three sections: threads, locks, and semaphores. The columns for the threads are:

- Id – Identification number for the thread.
- Property – The property associated with the thread. The example trace uses Readers and Writers. The property is fixed per thread during the run.
- Loc (location) – Location of the thread. The unique identifier of the last `mdat_invoke_scheduler` call executed by that thread. To see the what the unique identifiers refer to, refer to the instrumented source code files: `sectionsA.mdat.c` and `sectionsB.mdat.c`. These files are preprocessed source code files – your code is at the bottom of the file.
- Section – The section the thread is currently in: Entry, Exit, Critical, or Remainder. This field will be empty until the thread enters the first section and after the thread has completed.
- Status – The status of the thread: ready, running, completed, waiting-lock, or waiting-sem.
- Waiting on – If the thread is waiting, this column will show the name of the lock or semaphore that the thread is waiting on.

The lock section displays the name of each lock, the status of the lock (which includes the thread currently holding the lock if it is held), and a list of threads waiting for that lock. The semaphore section displays the name of each semaphore, its current value, and a list of threads waiting for that semaphore.

The first line of the trace will display the random seed and the last line of the trace will be an error message if deadlock or a mutual exclusion error occurred.

```
THREAD: 4
SECTION: Critical
MESSAGE: Invoked scheduler at location 21
******************************************************************************
|ID   |PROPERTY     |LOC  |SECTION      |STATUS       |WAITING ON            |
|0    |reader       |13   |Exit         |ready        |                      |
|1    |writer       |7    |Entry        |waiting-lock |writerLock            |
|2    |reader       |10   |Entry        |ready        |                      |
|3    |writer       |7    |Entry        |waiting-lock |writerLock            |
|4    |reader       |21   |Critical     |running      |                      |
|5    |writer       |19   |             |completed    |                      |
------------------------------------------------------------------------------
|LOCK NAME                 |STATUS       |WAITING THREADS                      |
|readCountLock             |held by 2    |                                     |
|writerLock                |held by 4    |1 3                                  |
------------------------------------------------------------------------------
|SEMAPHORE NAME            |VALUE        |WAITING THREADS                      |
|babySemaphore             |0            |                                     |
******************************************************************************
THREAD: 2
MESSAGE: Releasing lock readCountLock
******************************************************************************
|ID   |PROPERTY     |LOC  |SECTION      |STATUS       |WAITING ON            |
|0    |reader       |13   |Exit         |ready        |                      |
|1    |writer       |7    |Entry        |waiting-lock |writerLock            |
|2    |reader       |10   |Entry        |running      |                      |
|3    |writer       |7    |Entry        |waiting-lock |writerLock            |
|4    |reader       |21   |Critical     |ready        |                      |
|5    |writer       |19   |             |completed    |                      |
------------------------------------------------------------------------------
|LOCK NAME                 |STATUS       |WAITING THREADS                      |
|readCountLock             |unlocked     |                                     |
|writerLock                |held by 4    |1 3                                  |
------------------------------------------------------------------------------
|SEMAPHORE NAME            |VALUE        |WAITING THREADS                      |
|babySemaphore             |0            |                                     |
******************************************************************************
THREAD: 2
MESSAGE: Waiting for sem babySemaphore (-1)
******************************************************************************
|ID   |PROPERTY     |LOC  |SECTION      |STATUS       |WAITING ON            |
|0    |reader       |13   |Exit         |ready        |                      |
|1    |writer       |7    |Entry        |waiting-lock |writerLock            |
|2    |reader       |10   |Entry        |waiting-sem  |babySemaphore         |
|3    |writer       |7    |Entry        |waiting-lock |writerLock            |
|4    |reader       |21   |Critical     |ready        |                      |
|5    |writer       |19   |             |completed    |                      |
------------------------------------------------------------------------------
|LOCK NAME                 |STATUS       |WAITING THREADS                      |
|readCountLock             |unlocked     |                                     |
|writerLock                |held by 4    |1 3                                  |
------------------------------------------------------------------------------
|SEMAPHORE NAME            |VALUE        |WAITING THREADS                      |
|babySemaphore             |-1           |2                                    |
******************************************************************************
```

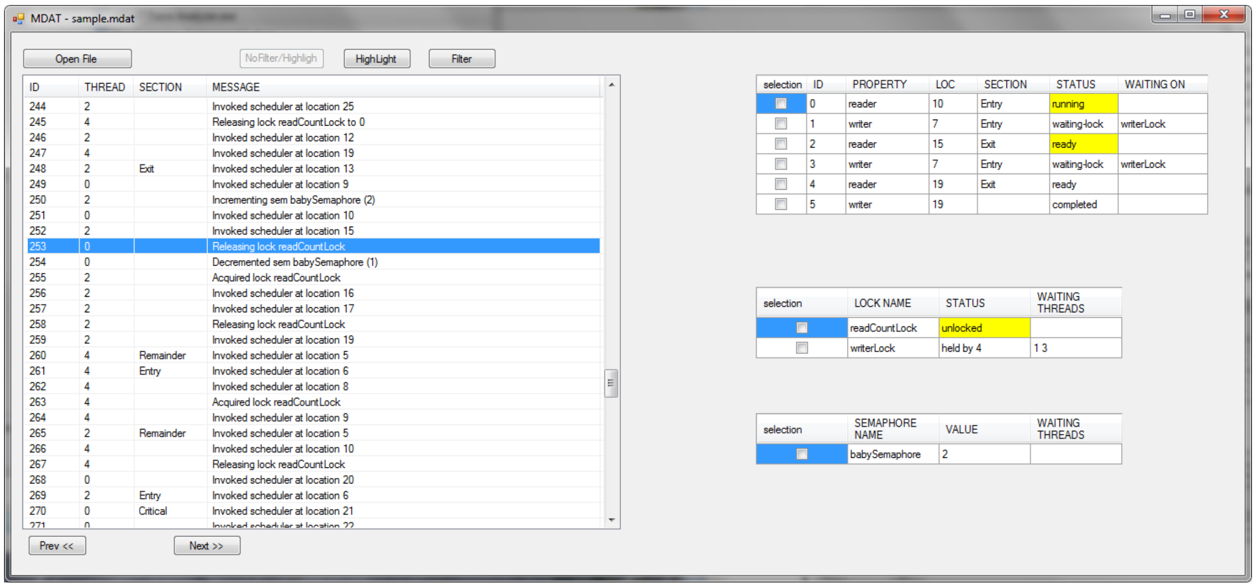
Interactive Mode

When running in interactive mode (`-i`), a prompt appears anytime there is a choice regarding threads. At the start, the user is prompted which thread should start running first. Each time the scheduler is invoked, the user is prompted on which thread to run next. Only the threads that are ready are acceptable; the user is asked to select a different thread if their selected thread is waiting or has already completed. If threads are waiting, the user is also prompted which thread to wake up when a lock is released or a semaphore is incremented. To quit, simply type ‘Q’ (must be capitalized) instead of entering a thread.

MDAT Trace Analyzer

The MDAT trace analyzer runs on Windows only (sorry – no Macs) and is available from the course website.

First, it is necessary to transfer the trace file from Linux to Windows using a program such as WinSCP (it is possible to automate this step – more on this later). Then start MDAT Trace Analyzer and press the Open File button. Use the dialog box to select the trace file to analyze. You will get a screen like this:



ID	THREAD	SECTION	MESSAGE
244	2		Invoked scheduler at location 25
245	4		Releasing lock readCountLock to 0
246	2		Invoked scheduler at location 12
247	4		Invoked scheduler at location 19
248	2	Exit	Invoked scheduler at location 13
249	0		Invoked scheduler at location 9
250	2		Incrementing sem babySemaphore (2)
251	0		Invoked scheduler at location 10
252	2		Invoked scheduler at location 15
253	0		Releasing lock readCountLock
254	0		Decremented sem babySemaphore (1)
255	2		Acquired lock readCountLock
256	2		Invoked scheduler at location 16
257	2		Invoked scheduler at location 17
258	2		Releasing lock readCountLock
259	2		Invoked scheduler at location 19
260	4	Remainder	Invoked scheduler at location 5
261	4	Entry	Invoked scheduler at location 6
262	4		Invoked scheduler at location 8
263	4		Acquired lock readCountLock
264	4		Invoked scheduler at location 9
265	2	Remainder	Invoked scheduler at location 5
266	4		Invoked scheduler at location 10
267	4		Releasing lock readCountLock
268	0		Invoked scheduler at location 20
269	2	Entry	Invoked scheduler at location 6
270	0	Critical	Invoked scheduler at location 21
271	0		Invoked scheduler at location 22

selection	ID	PROPERTY	LOC	SECTION	STATUS	WAITING ON
☐	0	reader	10	Entry	running	
☐	1	writer	7	Entry	waiting-lock	writerLock
☐	2	reader	15	Exit	ready	
☐	3	writer	7	Entry	waiting-lock	writerLock
☐	4	reader	19	Exit	ready	
☐	5	writer	19		completed	

selection	LOCK NAME	STATUS	WAITING THREADS
☐	readCountLock	unlocked	
☐	writerLock	held by 4	1 3

selection	SEMAPHORE NAME	VALUE	WAITING THREADS
☐	babySemaphore	2	

The left hand side of the screen shows the schedule of threads and the right hand side of the screen shows the thread, lock, and semaphore tables. The “Next >>” button will step to the next state in the list. The table entries that change during that step will be highlighted in yellow. The “Prev <<” button can be used to step back to the previous state. You can also click on an entry in the left table to jump immediately to that entry.

The tables on the right are identical to the tables in the text trace file except there is a checkbox to select the threads, locks, and semaphores. These checkboxes are used in conjunction with the highlighting and filtering mechanisms. Select the thread, locks, and semaphores that you are interested in. Select Highlight to highlight the entries in the left-hand table associated with the selected entities. Select Filter to only show the entries in the left-hand table that are associated with the selected entities (in other words, if the entry is not associated with one of the selected entries, it does not appear on the list). Select No Filter/Highlight to turn of the filter or highlighting.

To simplify the process of using the MDAT trace analyzer, you can direct Windows to automatically execute the MDAT trace analyzer when double-clicking a file with the extension `.mdat` (this assumes that all of your traces have a name with the extension `.mdat`). The simplest way to do this is to double-click on an `.mdat` file and select to browse for the executable. Use the dialog box to select the MDAT trace analyzer. Check the box to always use this program when opening this file. Once `.mdat` files are associated with the MDAT trace analyzer, you can double-click files in WinSCP, eliminating the extra step of downloading the trace file first. You may need to configure the options in WinSCP. In particular, go to Options→Preferences→Environment→Panels. The operation to perform on double-click should be Open.