



Programming with POSIX* Threads 4

Based on

Multi-Core Programming –

increasing performance through software multi-threading

by Shameem Akhter and Jason Roberts



Another Producer/Consumer Example

- Threaded word counter for two files
- Version 1 - comments
 - `progs/pthread/Molay/twordcount1.c`
- Version 2 - mutex
 - `progs/pthread/Molay/twordcount2.c`
- Version 3 - local counters
 - `progs/pthread/Molay/twordcount3.c`

Early Reporting by completed threads

- Version 4
 - `progs/pthread/Molay/twordcount4.c`
- Purpose
 - Suppose you want to make results available to the main thread as soon as a worker thread completes its work
 - Analogous to election counting where main count center will announce the district result
 - this is a combination of all the individual results from precincts
 - main count center wants to get individual precinct results as soon as they are available
 - Standard `thread_join` may not accomplish
 - may cause main thread to wait until last thread has completed

Programming with POSIX* Threads

3

Early Reporting by completed threads

- This is example of inter-thread notification problem
- Worker threads are producers in this case and need to notify the main thread that the work is done
 - They do this by signaling - raising a flag
- They also need to deliver the result
 - They do this by placing the result in a "mailbox"
 - This has space for only one result and so must be protected to avoid overwriting before the result has been read/consumer by the main thread
- This example achieves this using condition variables and mutex

Programming with POSIX* Threads

4

Early Reporting Logic

- Main thread launches counting threads and waits for results
 - Does this by calling `pthread_cond_wait` to wait for the flag to be signaled
- When counting thread finishes
 - Acquires mutex on mailbox
 - Checks the mailbox
 - If it is not empty, unlocks the mutex and waits for the flag to be signaled
 - If the mailbox is empty, thread delivers result, and signals the condition variable by calling `pthread_cond_signal`

Programming with POSIX* Threads

5

Early Reporting Logic (ctd)

- Signal wakes up main thread
 - It tries to acquire mutex on the mailbox
 - When the counting thread releases it, the main thread gets it
 - gets report from mailbox, and processes it (by adding to total and reporting on screen)
 - It then signals flag so any waiting counting thread can enter
 - Returns to call `pthread_cond_wait`

Programming with POSIX* Threads

6

Other things to note

- Need to pass file name to worker threads and get count back
 - Use structure to get around limitation on only one arguments

```
struct arg_set {           /* two values in one arg */
    char *fname; /* file to examine */
    int count; /* number of words */
};
```

- When mailbox is empty the pointer is NULL
 - When full it is set to pointer to the argument structure of the worker thread
 - The main thread can thus identify the worker that has reported and use pthread_join to join with it
- ```
if (mailbox == &args1) pthread_join(t1,NULL);
```

Programming with POSIX\* Threads

7

## Matrix Multiplication example Parallelism for Performance

- \pthreads\Nichols\examples\matrix
- Serial version
  - has a routine to multiply a row by a column and place element in resulting matrix
  - Calls this for each element of the product matrix
- Parallel version
  - Creates threads which run mult\_worker
  - Each mult\_worker runs the serial routine
  - How parallel is this?
  - What is the granularity?
  - Do we expect a speedup?

Programming with POSIX\* Threads

8

## Matrix Multiplication- Points to note

- Each worker thread is passed a structure of `package_t` type
  - Specifies row and column to be multiplied, size of matrix, and pointers to the matrices MA, MB and MC
- The code illustrates how to call `pthread_create` with non-default attributes

```
pthread_create(&threads[num_threads], &pthread_custom_attr,
 mult_worker, (void *) p);
```
- The attribute arguments are initialized with

```
pthread_attr_init(&pthread_custom_attr);
```

  - initialises a thread attributes object with the default value for all of the individual attributes used by a given implementation
  - Individual attributes can be changed later

Programming with POSIX\* Threads

9

## Matrix Multiplication- Points to note

- Example - changing the stack size

```
pthread_attr_getstacksize(&pthread_custom_attr, &thread_stack_size);
if (thread_stack_size < MIN_REQ_SSIZE) {
 pthread_attr_setstacksize(&pthread_custom_attr, (long)MIN_REQ_SSIZE);
}
```

Programming with POSIX\* Threads

10

## Threading for user interfaces

- To permit interaction while background processing is proceeding
- Also useful to permit animation of multiple images or advertisements in web browser
- Simple example
  - Animate a message using curses
  - separate the animation code from keyboard-input code
  - Share variables that define position and velocity
  - On multicore systems each thread may run on different core
  - pthreads\Molay\tbounce1d.c

Programming with POSIX\* Threads

11

## Message Animation using threads

- Two threads
- Main thread creates worker thread to move the message
- Continues to accept keyboard input to change speed, direction or to quit the application
- To change speed or direction updates the shared global variables `dir` and `delay`
- Also uses global variables `row` and `col` to keep state (position) of the message

Programming with POSIX\* Threads

12

## Animating Several Messages

- `threads\Molay\tanimate.c`
- Accepts up to ten strings as command line arguments and animates each independently on a separate line with its own direction and speed
- One thread to handle keyboard input
  - Q for quit, ' ' to reverse direction of each string, 0-9 to reverse direction of 0 to 9<sup>th</sup> string respectively
  - This thread also initializes the strings, rows and velocities and sets up curses
- One thread to handle each string
  - String, row, delay (speed) and direction are passed to each thread in a structure

Programming with POSIX\* Threads

13

## Animating Several Messages Issues with curses

- The screen and the curses functions that modify it are also shared by the animation threads
- The program uses a mutex to prevent simultaneous access to curses functions to avoid inconsistencies
  - Otherwise get inconsistencies if 2 or more threads execute the `move`, `addch`, `addstr`, `addstr`, etc concurrently or in parallel
- Also curses library does not know about threads
  - Functions are not reentrant and must not be interrupted by another thread
  - Use of the mutex here also prevents corruption of the internal data structures in curses
  - Need to do this for all non-reentrant system libraries

Programming with POSIX\* Threads

14

## Animating Several Messages

### Improvements to Note

- `dir` is shared between main thread and the thread handling the string animation and so should be changed in a critical section
  - This is missing in this version
  - Better to have different mutex for each string and include this in the structure
  -