ECE 459 W18 Midterm Solutions

P.Lam, J. Zarnett

February 28, 2019

(1)

(a) We are looking for one of the tasks to migrate between threads. In terms of program output, that means that there is a column with `rN` and then `sM` where $N \neq M$, or else `sN` and then `cM`. The most obvious output is:

```
00  01  02  03  04
r1
    r2
        r0
s1
    s2
        s0
c2
    c2
        c0
                r0
            r2
                s0
            s2
                c0
            c2
```

(b) Mandatory:

1. What percentage of threads only read when in this critical section
2. The time it takes to lock/unlock rw locks compared to regular ones.

And one of:

- Whether starvation is an issue / risk in your implementation / code
- The difficulty of making the change in the source code (ie is it worth it)
- Any other reasonable consideration

(c) This is basically a side channel attack: if you know the cache size is n pages one test looks like: you load $n+1$ into memory and see which page was replaced by testing access times to each of the first n pages. The page where it takes longer than the others was replaced. You'll need a few test scenarios to figure it out, but it is easily possible.

(d) Yes—cancellation handlers do improve performance. Cancelling a thread means that unnecessary work is no longer performed, so it is desirable. However, cancellation might leave resources locked or allocated; cancellation handlers ensure that they are unlocked/freed, meaning it is safe to cancel more often.

(2) Here's the code with line numbers.

```
1  static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2  static int acquires_count = 0;
3
4  int trylock() {
5      int res = pthread_mutex_trylock( &mutex );
6      if (res == 0) {
7          ++acquires_count;
8      }
9      return res;
10 }
```

Consider the following execution trace (the easiest thing is to assume that the instructions are interleaved on one processor); both T_1 and T_2 are running `trylock()`. Let's assume `acquires_count` starts at 0.

1. T_1, line 5: acquire the lock, return 0.
2. T_2, line 5: fail to acquire the lock, return 1
3. T_1, line 6: *compare* res, set T_1 carry to true
4. T_2, line 6: *compare* res, set T_2 carry to false
5. T_1, line 7: *add* 1 to `acquires_count` and *unconditionally store* 1.
6. T_2, line 7: *add* 0 to `acquires_count` and *unconditionally store* 0, which is the wrong answer.

(3)

```
double total = 0.0;

int main( int argc, char** argv ) {
  #pragma omp parallel
  {
    #pragma omp single /* master also OK */
    {
      while( 1 ) {
        course_term* next = get_next( );
        if ( next == NULL ) {
          break;
        }
        #pragma omp task shared(total)
        {
          double aus = query_db( next );
          free( next ); /* Can also go after the addition */

          #pragma omp atomic /* Atomic is better than critical, but critical is acceptable */
          total += aus;
        }
      }
      #pragma omp taskwait /* Wait for all tasks to be done */
    }
  }
  printf( "Total AUs: %g\n.", total );
  return 0;

}
```

(4)

One solution involve checking `i` a lot:

```
void process( char* buffer ); /* Implementation not shown */

int main( int argc, char** argv ) {
  char* buffer1 = malloc( MAX_SIZE * sizeof( char ));
  char* buffer2 = malloc( MAX_SIZE * sizeof( char ));

  int fd = open( argv[1], O_RDONLY );
  memset( buffer1, 0, MAX_SIZE * sizeof( char ));
  read( fd, buffer1, MAX_SIZE );
  close( fd );

  for ( int i = 2; i < argc; i++ ) {
    int nextFD = open( argv[i], O_RDONLY  );

    aiocb cb;
    memset( &cb, 0, sizeof( aiocb ));

    cb.aio_nbytes = MAX_SIZE;
    cb.aio_fildes = nextFD;
    cb.aio_offset = 0;
    if ( i % 2 == 0 ) {
      memset( buffer2, 0, MAX_SIZE * sizeof( char ));
      cb.aio_buf = buffer2;
    } else {
      memset( buffer1, 0, MAX_SIZE * sizeof( char ));
      cb.aio_buf = buffer1;
    }
    aio_read( &cb );

    if ( i % 2 == 0 ) {
      process( buffer1 );
    } else {
      process( buffer2 );
    }

    while( aio_error( &cb ) == EINPROGRESS ) {
      sleep( 1 );
    }
    close( nextFD );
  }
  process( argc % 2 == 0 ? buffer1 : buffer2 );

  free( buffer1 );
  free( buffer2 );

  return 0;
}
```

Alternatively, you can swap buffers:

```
void process( char* buffer ); /* Implementation not shown */

int main( int argc, char** argv ) {
  char* buffer1 = malloc( MAX_SIZE * sizeof( char ));
  char* buffer2 = malloc( MAX_SIZE * sizeof( char ));

  int fd = open( argv[1], O_RDONLY );
  memset( buffer1, 0, MAX_SIZE * sizeof( char ));
  read( fd, buffer1, MAX_SIZE );
  close( fd );

  for ( int i = 2; i < argc; i++ ) {
    int nextFD = open( argv[i], O_RDONLY  );

    aiocb cb;
    memset( &cb, 0, sizeof( aiocb ));

    cb.aio_nbytes = MAX_SIZE;
    cb.aio_fildes = nextFD;
    cb.aio_offset = 0;
    memset( buffer2, 0, MAX_SIZE * sizeof( char ));
    cb.aio_buf = buffer2;

    aio_read( &cb );

    process( buffer1 );

    while( aio_error( &cb ) == EINPROGRESS ) {
      sleep( 1 );
    }
    close( nextFD );

    char* tmp = buffer1;
    buffer1 = buffer2;
    buffer2 = tmp;
  }
  process( buffer1 );

  free( buffer1 );
  free( buffer2 );

  return 0;
}
```