1 Shared Memory I; Processes, Threads

1. Processes; the following program determines the sum the elements of an array A[1000], we also need lockinit, lock, unlock.

   - The sequential structure is

     ```
     int sum, A[1000];
     sum = 0;
     for (i = 0, i < 1000; i++) sum += A[i];
     ```

   - Division Into two processes; in this method, we will divide the program into two parts, one doing the even i and one doing the odd i.
     - Running the two processes, *sum* will need to be **shared**.
   - For demonstration purpose; creating a library is also presented.
     - After creating the C source files, compile the files into object files.

       ```
       gcc -c -fPIC lock.c
       gcc -c -fPIC lockinit.c
       gcc -c -fPIC unlock.c
       ```

     - *Static library*
       * Create a static library called *libspin.a*. Then, create an index inside the library.

         ```
         ar rc libspin.a lock.o lockinit.o unlock.o
         ranlib libspin.a
         ```

       * Remember to prototype your library function calls so that you do not get implicit declaration errors.
       * When linking your program to the libraries, make sure you specify where the library can be found:

         ```
         gcc -o process  process.c -L. -lspin
         ```

         The $-L.$ piece tells gcc to look in the current directory in addition to the other library directories for finding *libspina.a*.
     - *Shared library*
       * To create a shared library

         ```
         gcc -shared -fPIC -o libspin.so lock.o lockinit.o unlock.o
         ```

       * The $-fPIC$ option is to tell the compiler to create Position Independent Code (create libraries using relative

addresses rather than absolute addresses because these libraries can be loaded multiple times)

* To compile the actual program using the libraries

  `gcc -o process1 process.c -L. -lspin`

* The key to making your program work with dynamic libraries is through the $LD\_LIBRARY\_PATH$ environment variable.

* Append to the variable

  `export LD_LIBRARY_PATH=/path/to/library:${LD_LIBRARY_PATH}`

– Notice they are exactly the same as creating. Although, it is compiled in the same way, none of the actual library code is inserted into the executable, hence the dynamic/shared library.

– Shared libraries dynamically access libraries at run-time thus the program needs to know where the shared library is stored.

– The executable is much smaller than with static libraries. If it is a standard library that can be installed, there is no need to compile it into the executable at compile time!

2. Threads;

- The following program determines the sum by using threads;
  – Compile the code as

    `gcc -o sumthread sumthread.c -lpthread`

  – Increase the number of the threads and change the size of array. Observe if the all the threads have the partial sum all the time. Why not?

- The following code segment is for finding the minimum of a list of integers.
  – The list is partitioned equally among the threads.
  – The size of each thread's partition is stored in the variable *partial_list_size*.
  – The pointer to the start of each thread's partial list is passed to it as the pointer *list_ptr*.

  Complete the program, compile and execute.

```
1   #include <pthread.h>
2   void *find_min(void *list_ptr);
3   pthread_mutex_t minimum_value_lock;
```

```
4   int minimum_value, partial_list_size;
5
6   main() {
7       /* declare and initialize data structures and list */
8       minimum_value = MIN_INT;
9       pthread_init();
10      pthread_mutex_init(&minimum_value_lock, NULL);
11
12      /* initialize lists, list_ptr, and partial_list_size */
13      /* create and join threads here */
14  }
15
16  void *find_min(void *list_ptr) {
17      int *partial_list_pointer, my_min, i;
18      my_min = MIN_INT;
19      partial_list_pointer = (int *) list_ptr;
20      for (i = 0; i < partial_list_size; i++)
21          if (partial_list_pointer[i] < my_min)
22              my_min = partial_list_pointer[i];
23      /* lock the mutex associated with minimum_value and
24      update the variable as required */
25      pthread_mutex_lock(&minimum_value_lock);
26      if (my_min < minimum_value)
27          minimum_value = my_min;
28      /* and unlock the mutex */
29      pthread_mutex_unlock(&minimum_value_lock);
30      pthread_exit(0);
31  }
```

- For computing the value of π; the following program computes the value of the π number by a given number of threads;
  - Analyze the code for the possible synchronization issues.
  - Compile and execute the code by increasing the number of the threads and sample points.
  - Make the following procedures:
    - Draw a figure as Execution Time vs Number of Threads. If you increase the number of threads as the multiple of 2, then take the ln. If you increase the number of threads as the multiple of 10, then take the log.
    - Label the curve as "local".
    - Illustrate an important performance overhead called *false sharing*. Consider the following change to the program: instead of incrementing a local variable, *local_hits*, and assigning it to

the array entry outside the loop, we now directly increment the corresponding entry in the *hits* array. This can be done by changing line 64 to **(hit_pointer) ++;*, and deleting line 67.

* Repeat the procedure above and label the curve as "spaced_1". This represents a significant slowdown instead of a speedup! The drastic impact of this seemingly harmless change is explained by a phenomenon called false sharing. In this example, two adjoining data items (which likely reside on the same cache line) are being continually written to by threads that might be scheduled on different processors. We know that a write to a shared cache line results in an *invalidate* and a subsequent read must fetch the cache line from the most recent write location. With this in mind, we can see that the cache lines corresponding to the hits array generate a large number of invalidates and reads because of repeated increment operations. This situation, in which two threads 'falsely' share data because it happens to be on the same cache line, is called false sharing.
* It is in fact possible to use this simple example to estimate the cache line size of the system. We change *hits* to a two-dimensional array and use only the first column of the array to store counts. By changing the size of the second dimension, we can force entries in the first column of the *hits* array to lie on different cache lines (since arrays in C are stored row-major).
* Make this change and repeat the procedure above for two different size values. Label the curves as "spaced_16" and "spaced_32", in which the second dimension of the *hits* array is 16 and 32 integers, respectively. It is evident that as the entries are spaced apart, the performance improves. This is consistent with our understanding that spacing the entries out pushes them into different cache lines, thereby reducing the false sharing overhead.
* Draw this 4 lines in one figure. Make the Speed-Up and Efficiency analysis.