

## Objectives:

Learn to:

- Solve inter-process communication problems during concurrent execution of processes.
- Use Posix Pthread library for concurrency.

## Problem Statement:

**1. Multi-processor Synchronization:** Larry, Moe, and Curly are planting seeds. Larry digs the holes. Moe then places a seed in each hole. Curly then fills the hole up.

There are several synchronization constraints:

- Moe cannot plant a seed unless at least one empty hole exists, but Moe does not care how far Larry gets ahead of Moe.
- Curly cannot fill a hole unless at least one hole exists in which Moe has planted a seed, but the hole has not yet been filled. Curly does not care how far Moe gets ahead of Curly.
- Curly does care that Larry does not get more than MAX holes ahead of Curly. Thus, if there are MAX unfilled holes, Larry has to wait.
- There is only one shovel with which both Larry and Curly need to dig and fill the holes, respectively.

Design, implement and test a solution for this IPC problem, which represent Larry, Curly, and Moe. Use semaphores as the synchronization mechanism.

(LarryMoeCurly.c →LCM)

**2.**<sup>1</sup> Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and once finished will return them. As an example, many commercial software packages provide a given number of licenses, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. If all the licenses are in use, requests to start the application are denied. Such requests will only be granted when an existing license holder terminates the application and the license is returned.

The following program segment represents the pseudo code that can be used to manage a finite number of resources. The maximum number of resources and the number of available resources are initialized as follows:

```
#define MAX_RES 5  
int avail_res = MAX_RES;
```

---

<sup>1</sup> This problem is based on exercises 6.27 and 6.28 from SilberChatz et al. Operating Systems: 7<sup>th</sup> Edition.

When a process wishes to obtain a number of resources, it invokes **decrease\_count()** function.

```
int decrease_count(int count) {  
    {  
        if (avail_res < count) return -1;  
        else avail_res = avail_res - count;  
        return 0;  
    }  
}
```

When the process wants to return a number of resources, it calls the **increase\_count()** function.

```
int increase_count(int count)  
{  
    avail_res = avail_res + count;  
    return 0;  
}
```

The preceding code results in race condition. Identify the critical section and protect then using semaphores. These functions will be called by a resources manager; Make sure the resource manager does not busy-wait but blocks the process (thread) calling the **decrease\_count()**. The two choices are shown below: preferred choice is the second.

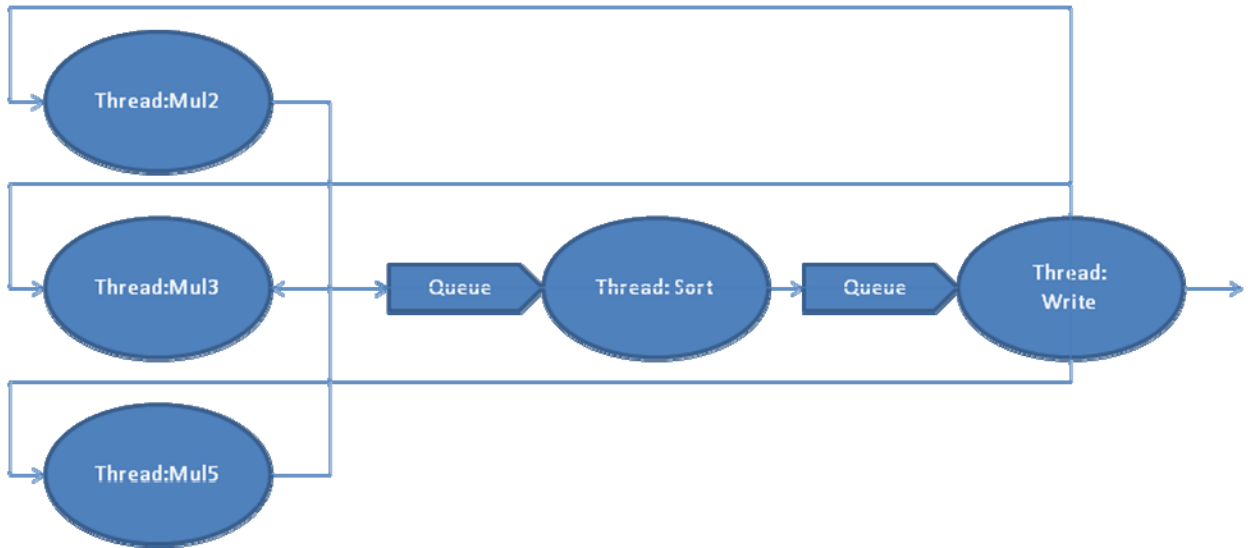
**while (decrease\_count(count) = -1) ; // busy waiting: not a good choice**

**decrease\_count(count);  
// block; process will return from fun. call when sufficient resources are available**

Design and implement the solution in ResMgt.c → ResMgt

3. This problem applies concurrency to a problem in an application domain. We have chosen an unusual domain, that of “regular numbers”. Hamming numbers are used in error correction and you are required to design a multi-threaded Hamming-code generator. Hamming numbers can be generated by positive numbers greater than or equal to 1 containing no prime factors other than 2, 3 and 5, i.e. numbers of the form  $2^i \times 3^j \times 5^k$  ( $i, j, k \geq 0$ ). The ideas to compute them are the following:

- Given a hamming number  $h$ , then  $2h$ ,  $3h$ ,  $5h$  are hamming numbers.
- $1(One)$  is a hamming number at it is used to start the computation.
- To maintain them sorted, it is only needed to compare the numbers coming from the multiplication for 2, 3, 5 not used yet.



**Figure 1 Design of a Hamming Generator**

**Material to be submitted:**

- Submit the source code for the programs. Use meaningful names for the file so that the contents of the file is obvious from the name. You may zip all the source files into a single file. Also provide a Pr2README file that explains the contents of the zip file. Pr2README file should have an observation section for each of the three problems. Use tables where ever suitable (5 points for documentation)
- Use internal documentation to explain your design.
- Test runs: It is very important that you show that your program works for all possible inputs. Submit a single script that shows for each program the working for correct input as well as graceful exit on error input.
- Include your makefile within your zip/tar file.

**Due date**

3/27 submit on-line before mid-night.