CS 361–Operating Systems – Fall 2017
Homework Assignment 5 - pThreads and Synchronization

Due: Monday 4 December at 4:00 P.M. via Blackboard. Optional hard copy may be submitted to the TA.

Overall Assignment

For this assignment, you are to write a simulation in which a number of threads access a group of shared buffers for both reading and writing purposes. Initially this will be done without the benefit of synchronization tools (semaphores) to illustrate the problems of race conditions, and then with the synchronization tools to solve the problems. (A separate mutex will be used to coordinate writing to the screen.) Optional enhancements allow for the use of pipes, message queues, and shared memory.

Man Pages

The following man pages may be useful for this assignment, (some for optional enhancements.)

- ipc(5)
- ipcs(8)
- ipcrm(8)
- pthreads(7)
- pthread_create
- pthread_join
- pthread_exit
- pthread_cancel
- msgget(2)
- msgctl(2)
- msgsnd(2)
- msgrcv(2)
- shmget(2)
- shmctl(2)
- shmat(2)
- shmdt(2)
- semget(2)
- semctl(2)
- semop(2)

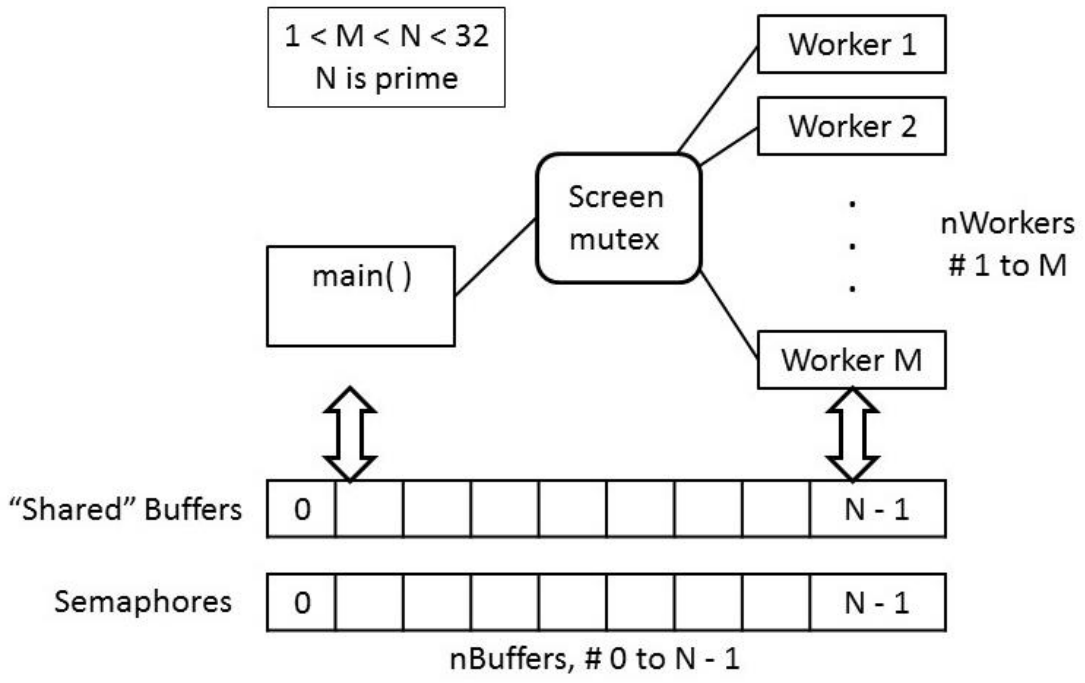
- rand(3)
- srand(3)
- RAND_MAX
- qsort(3)

- fork(2)
- exec(3)
- pipe(2)
- dup2(2)
- wait(2)

- sprintf
- time(2)
- gettimeofday(2)
- clock_gettime(2)
- sleep(3)
- usleep(3)
- nanosleep(2)
- perror(3)

Program Structure

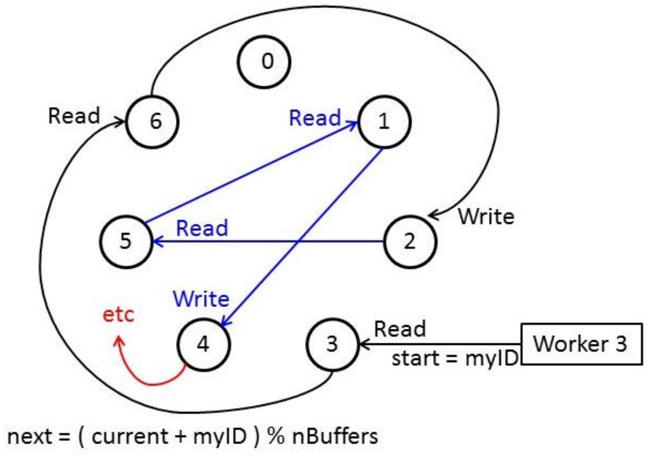
The main thread will create nWorker worker threads, which will access nBuffer buffers controlled by nBuffer semaphores as follows. (Note: nBuffers buffers numbered 0 to N – 1, nWorkers workers numbered 1 to M.)



The Workers' Task

- Each worker will perform a sequence of read and write operations on the shared buffers, simulating the memory accesses performed by the simple statement "A = B + C;", i.e. (**fetch** B, **fetch** C, **store** A). However it will not actually add up the buffer contents. Instead it will access the buffers in the following manner. (where nBuffers and nWorkers correspond to M and N in the diagram above.)
    - The first buffer accessed will be the worker's ID number, e.g. buffer[3] for worker 3.
    - Successive accesses will jump by steps of the worker's ID number, in a circular fashion. Worker number 3 will access every 3rd buffer, worker number 5 will access every 5th buffer, etc., wrapping back to zero on a mod basis when the numbers exceed nBuffers - 1.
    - The first two accesses of every three will be for reading, and the third for writing.
    - The cycle continues until each worker has written into nBuffers of the buffers, which should involve writing into each buffer exactly once, (3 * nBuffers total accesses), since nBuffers is prime and nWorkers must be strictly less than nBuffers.
    - So for example, if nBuffers = 7, then the order of access for worker number 3 would be as follows, where the underlined numbers are write access and the others are reads. (Note that every buffer gets written into exactly once, and read from exactly twice, by each worker.)

3, 6, **2**, 5, 1, **4**, 0, 3, **6**, 2, 5, **1**, 4, 0, **3**, 6, 2, **5**, 1, 4, **0**



- A **read** operation will consist of the following steps:
    - Read the initial value in the buffer.
    - Sleep for sleepTime seconds, where sleepTime is the random sleep time assigned to this worker. (Note: use usleep( ) instead of sleep( ) for sub-second resolution. See man 3 sleep, man 3 usleep and man 2 nanosleep for details.)
    - Read the value of the buffer again, and print a message to the screen if the value changed while the worker was sleeping. The message should include (at a minimum) the ID of the worker that detected the error, the buffer which changed, the initial value, and the final value.

- A **write** operation will consist of the following steps:
    - Read the initial value of the buffer.
    - Sleep for sleepTime seconds. (See above.)
    - Add 1 << (ID – 1) to the value read initially, and store the result back into the buffer. (E.g. worker number 3 adds binary 000001 shifted left 2 places = 000100 = 4. Worker 4 adds 8, worker 5 adds 16, etc.) Each worker will add a single binary bit in a different position, which will make it possible to later identify which worker(s) were involved in any error conditions that erased write operations.

Command Line Arguments passed to main( )

For this assignment the program name will be "raceTest" having the following command line arguments:

```
raceTest nBuffers nWorkers [ sleepMin sleepMax ] [ randSeed ] [ -lock | -nolock ]
```

- nBuffers **must be a PRIME integer between 2 and 32** for this program to run correctly. (Easy test: 2 < nBuffers < 32 and nBuffers not evenly divisible by 2, 3, or 5)

- nWorkers is the number of threads that will execute the worker function, and must be a positive number less than nBuffers.

- sleepMin and sleepMax are positive **real** numbers, max greater than min, that define a range of sleeping times for the worker threads. The initial thread will begin by generating a series of nWorkers random numbers within this range. Default = 1.0 and 5.0 if not provided.

- If provided, randSeed is the integer value to use to seed the random number generator. If it is missing or zero, then seed the random number generator with time(NULL).

- –lock or –nolock indicates whether or not semaphores are to be used to restrict exclusive access to the shared memory area. Default if this argument is not present is –nolock, i.e. without using semaphores.

Thread Function - void * worker(void *);

The threads created for this assignment will launch the function worker( ), which takes a void pointer as its only input parameter, and returns a void pointer. In practice the input pointer will point to a struct, containing the following pieces of information:

- int nBuffers is the same parameter passed to main( ) and explained above.
- int workerID is a unique identifier (ID) for this particular worker, **starting with 1** for the first worker and continuing to nWorkers for the last worker.
- double sleepTime is the amount of time this particular worker will spend sleeping in each work cycle.
- int semID is the integer ID number for the semaphore block used to control access to the buffers, or -1 if that feature is not currently enabled
- int mutexID is the ID for the separate semaphore used to control access to the screen, or -1.
- int *buffers holds the address of an array of ints, created and initialized to zeros by main( ).
- int nReadErrors reports the number of read errors that this particular worker discovered.
- Additional parameters may be necessary. You should avoid the use of global variables if at all possible.

Error Detection

Two types of errors will be detected and reported (when not using semaphores.)

- Read errors occur when buffers change values while a worker is sleeping during a read cycle.

- Write errors occur when one worker overwrites another worker's contribution during a write cycle.

- Worker threads can detect and report read errors. Main( ) will have to check for write errors.

Evolutionary Development

It is recommended to develop this program in the following steps:

- After parsing the command line arguments, main( ) should create a local array of nBuffer integers, and initialize them all to zero.

- Next main( ) will need to create an array of nWorker structs as defined above, and populate each struct with the values needed for one of the threads. (Note that you cannot just use a single struct repeatedly.)
    - The sleepTime is a random value in the range from sleepMin to sleepMax. (Seed the random number generator first, with a single call to srand( ). use randSeed if provided, or time(NULL) otherwise.) (Random double precision floating point numbers in the range from A to B can be generated as `A + ( B - A ) * rand( ) / RANDMAX.`)
    - The sempahore ID and mutex ID should both be -1 for now.
    - nReadErrors is a result that will be returned by the worker thread. Can be 0 for now.

- Next main should create nWorkers new threads, each running the worker( ) function, passing each a pointer to one of the structs in the array.

- At this point the worker threads should just print the values they received, store their ID number in buffers[ID], and then either return or call pthread_exit( ). The printed messages will probably be all mixed together on the screen, which is okay for now. (You could try having each of them sleep for their randomly assigned sleep time first and then print. The result will probably still be somewhat garbled, but not as badly.)

- After having created all the workers, main( ) should then call pthread_join( ) for each of the threads to wait for them to finish. After all the threads have completed, main( ) should report the values stored in each of the buffers (to confirm that each of the workers were able to write to a buffer.)

- Next modify the worker processes to execute the cycle of reading from and writing described above, ***instead of*** the storing their ID number. Without semaphores in place, there should be errors caused by race conditions, which are reported to the screen as each thread detects them, and reported by main( ) after the threads have completed their work.

    - Worker threads should check for, count, and report read errors when they occur.

    - After a worker has completed all write operations, it should set the nReadErrors field of its struct equal to the number of read errors it discovered, and then either return NULL or call pthread_exit(NULL), (unless you feel a need to return something, possibly for debugging purposes.)

    - After all threads have finished, main( ) should examine the contents of the shared buffers, and report any write errors detected. Without errors each buffer should contain (2^nWorkers) - 1, which can be calculated as (1 << nWorkers) - 1. (I.e. if there were 3 workers, then each buffer should contain 7 if there are no errors.)

        - A bitwise exclusive OR (^) between the expected and actual answers will result in bits turned on only where the two values do not match. A bitwise AND (&) in a loop can then test each bit one by one to report which bits were "lost" due to overwriting.

- Main should report the total number of read and write errors discovered.

- Up to this point of development, all the printed messages should have been garbled up together. The next step is to implement the screen lock mutex to address this problem.
    - Main( ) should create a single semaphore using semget( ), and use semctl( ) to initialize it to 1. The ID should be stored in the mutexID field passed to the worker( ) functions.
    - Before any thread writes to the screen, it should lock the mutex using semop( ) with an argument of -1. When they are done, they call semop( ) again with an argument of +1.
    - As long as main( ) does not print to the screen or access the workers' structs while the worker threads are running, then it does not need to use the screen lock mutex.
    - See below for full details of semaphore operations.
- Finally implement semaphores using the –lock / -nolock flags. The errors should go away when semaphores are in use. An optional enhancement is to detect the time penalty imposed by the use of exclusive access through semaphores.
    - System V semaphores are actually allocated in sets, which happens to work out perfectly for this part of the assignment. Main( ) needs to allocate a set (array) of nBuffers semaphores, with one assigned to protect each of the shared memory buffers.
    - Main( ) should use semctl( ) to initialize all the semaphores to 1 before forking off any children. The worker processes will then use semop( ) to implement wait( ) and signal( ) as appropriate.

Pthreads

Put some instructions in here explaining how to use pthreads.

Interprocess Communications, IPC

(See previous assignments for general information on IPC, as well as specifics for message queues, pipes, and shared memory as needed for optional enhancements.)

Semaphore Operations

- The relevant commands for semaphores are semget(2), semctl(2), and semop(2), analogous to msgget, msgctl, and msgsnd/msgrcv as described previously.
- Note that System V semaphores come as an <u>**array of**</u> semaphores, which is actually convenient for our purposes. The semget command returns an ID for the entire set of semaphores generated. The three arguments to semget are a key number (IPC_PRIVATE), the number of semaphores desired in the set, and a flags word.
- Semctl uses a ***union*** of an integer value, an array of shorts, and an array of structs. A union is similar to a struct, except that only enough memory for the largest data item is allocated, and that memory is shared by all elements of the union. (A struct containing a double, an int, and a char would be given enough space to store all three independently; A union with the same variables would only allocate enough room for the double (the largest element), and all three variables would be assigned the same storage space. Obviously in practice one (normally) only uses one of the elements of a union.)

- For compatibility purposes, you may need the following code at the top of your program (right after the #includes) when using semphores:

  ```
  #if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
  /* union semun is defined by including <sys/sem.h> */
  #else
  /* according to X/OPEN we have to define it ourselves */
  union semun {
        int val;                    /* value for SETVAL */
        struct semid_ds *buf;       /* buffer for IPC_STAT, IPC_SET */
        unsigned short *array;      /* array for GETALL, SETALL */
                                    /* Linux specific part: */
        struct seminfo *__buf;      /* buffer for IPC_INFO */
  };
  #endif
  ```

- Semctl is used to get / set the ***initial*** values of semaphores, query how many processes are waiting for semaphore operations, and to get specific process IDs of such processes. Semctl is used for initialization and maintenance of semaphores, but should NOT be used for the wait and signal operations discussed in class. (See semop, next bullet point.)
- The semop system call is used to perform normal semaphore operations. The command takes three arguments: a semaphore set ID, an array of sembuf structs, and an integer indicating the size of the array of structs, i.e. the number of semaphores being acted on at one time.
- The sembuf struct contains three fields: the semaphore number to operate on, a short int indicating the semaphore operation desired, and another short int of flags (e.g. IPC_NOWAIT). Negative semaphore operations decrement the semaphore, blocking if appropriate (e.g. wait), a zero value blocks until the semaphore is exactly zero, and positive numbers increment (e.g. signal). If you are not familiar with bit twiddling in C/C++, you may want to find a good book and review the bitwise operators ~, &, |, ^, <<, and >>.

Required Output

- All programs should print your name and CS account ID as a minimum when they first start.
- Each read error should be reported to the screen as it is detected.
- The main thread should report how many read errors occurred (counted by workers), as well as how many write errors occur (buffers which do not hold the correct value at the end, which should be 2^nWorkers - 1 for all buffers.) Note the following:
  - In the case of read errors, the process(es) that wrote to the buffer while the reading worker was sleeping can be determined by examining the bits of the difference between the initial and final values of the buffer. (Bits can also be lost, if two race conditions occur at once.)
  - In the case of write errors, the bits turned on in the difference between the correct value and the actual value indicates which workers did not contribute to the total because their results were lost in a race condition situation. (The correct answer should be all 1s in the rightmost nWorkers bit positions. Any 0 in any of these positions indicates a particular worker whose write operation was over-written by another process.)
  - Report bad bits and buffers starting at 0, and count each wrong bit separately.

Sample Output

```
Running simulation for 6 children accessing 13 buffers, without locking

Child number 3 reported change from 0 to 8 in buffer 12.  Bad bits = 3
Child number 3 reported change from 0 to 16 in buffer 2.  Bad bits = 4
Child number 3 reported change from 0 to 8 in buffer 11.  Bad bits = 3
Child number 3 reported change from 0 to 16 in buffer 4.  Bad bits = 4
Child number 1 reported change from 32 to 8 in buffer 10.  Bad bits = 3 -5
Child number 5 reported change from 1 to 3 in buffer 3.  Bad bits = 1
Child number 2 reported change from 0 to 1 in buffer 7.  Bad bits = 0
Child number 2 reported change from 0 to 1 in buffer 0.  Bad bits = 0
Child number 5 reported change from 3 to 7 in buffer 0.  Bad bits = 2

Error in buffer 5.  Bad bits = 2
Error in buffer 8.  Bad bits = 4
Error in buffer 10.  Bad bits = 5
Error in buffer 11.  Bad bits = 0
Error in buffer 12.  Bad bits = 3 5

10 Read errors and 6 write errors encountered.
```

Other Details:

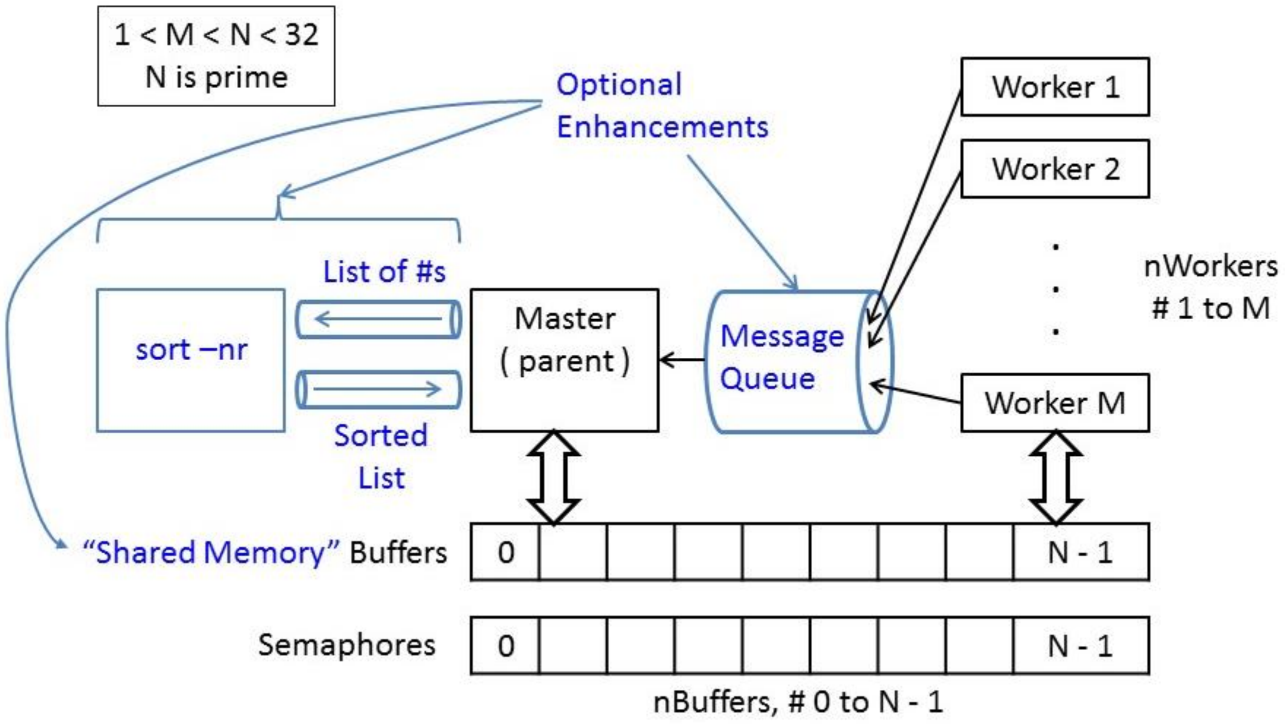
- The TA must be able to build your executable by typing "make raceTest". As always, you are free to develop wherever you wish, but the TA will be grading the programs based on their performance on the standard system machines.

What to Hand In:

1. Your code, **including a readme file, and a makefile if necessary,** should be handed in electronically using Blackboard.
2. The purpose of the readme file is to make it as easy as possible for the grader to understand your program. If the readme file is too terse, then (s)he can't understand your code; If it is overly verbose, then it is extra work to read the readme file. It is up to you to provide the most effective level of documentation.
3. The readme file must specifically provide direction on how to run the program, i.e. what command line arguments are required and whether or not input needs to be redirected. It must also specifically document any optional information or command line arguments your program takes.
4. If there are problems that you know your program cannot handle, it is best to document them in the readme file, rather than have the TA wonder what is wrong with your program.
5. A printed copy of your program, along with any supporting documents you wish to provide, may optionally be provided to the TA.
6. Make sure that your **name and your CS account name** appear at the beginning of each of your files. Your program should also print this information when it runs.

Optional Enhancements:

It is course policy that students may go above and beyond what is called for in the base assignment if they wish. These optional enhancements will not raise any student's score above 100 for any given assignment, but they may make up for points lost due to other reasons. Note that all optional enhancements need to be clearly documented in your readme files. The following are some ideas, or you may come up with some of your own:



- Instead of putting random sleep times directly into the worker structures, main can sort them first (in descending order) by storing the random numbers into an array, sorting the array, and then copying the times from the array into the structures. Sorting can be done using one of the following methods:

A. The base assignment does not require sorting. This would not count as an optional enhancement.

B. The easiest way to sort an array in C is to call the qsort(3) function built in to the language. Note that this requires a comparator function, that takes two pointers to doubles, and returns a -1, 0, or 1 if the first double pointed to is less than, equal to, or greater than the second.

C. As an interesting exercise in using pipes to communicate with a child process, sort the array by forking off a child which execs the system sort command (with -nr flags), connecting the child to the parent with two pipes in a loopback configuration.

    - For option C, the parent first creates two pipes, pipe1 and pipe2, and then forks off a child.
    - The child will close the pipe ends it doesn't need, dup2 the others over stdin and stdout, (close them after they are duped), and then exec "sort –nr".
    - The parent will generate nWorkers random sleep times and write them to the writing end of pipe1, one per line as character strings, e.g. using sprintf. Close all unused pipe ends as soon as they are no longer needed, e.g. as soon as the writing is completed, to start the sort process.
    - The parent will then read in as many random numbers from the reading end of pipe2 as it wrote to the writing end of pipe1, again as strings of chars which must be converted to doubles using fscanf. For verification purposes, the random numbers should be printed before they are written to pipe1, and again as they are read from pipe2.

- As an alternative to using a mutex for locking access to the output screen, a message queue could be used instead. Under this approach each thread would send a message back to the main thread when errors were detected, and it would be the main thread's job to print the corresponding error messages. Under this scenario the worker threads would also need to send a final message letting the main thread know when they were done with their work, so the main thread will know when to join them. (Alternatively a separate thread could be created for reading and printing the messages from the message queue. In this case the main thread would cancel and join the printing thread after having joined all of the worker threads.)
- Shared memory, e.g. allocated with shmget could be used instead of just memory that all threads have access to anyway. In this case the shared memory ID should be passed to the thread functions, which should then call shmat to get values for local pointer variables.
- Measure and report the times required for each process to complete their task, as well as the amount of time spent waiting for locks to clear. (These numbers should be reported for each child, in the form of a table. The time spent waiting for locks should probably be determined by each child, and reported to the parent in their final message.) See time(2), gettimeofday(2), clock_gettime(2). (Time spent sleeping should be easily calculated. Can you check this?) The overall goal of this enhancement is to show the time penalty incurred by the use of locks, as compared to running the program without locks.
- Instead of using simple locks (semaphores) on the data buffers, try implementing reader-writer locks, and see if you can detect a difference in the performance.
- As written above, each child will only need access to one buffer at a time, (either for reading or writing), so there cannot be any deadlocks. A modification is to require that each child gain access to two buffers for reading and one for writing **<u>at the same time</u>**, and then to implement code to ensure that deadlocks cannot occur. Note that it will be necessary to show that deadlock can and does occur if deadlock avoidance is not implemented, just as the base assignment shows that race conditions occur when synchronization is not implemented. (It may be necessary to increase the number of child processes or the number of buffers simultaneously held in order to ensure deadlock occurs.)
- Provide a command-line option for implementing the assignment using forking instead of pthreads, and compare the performance of the two implementation methods. Since threads already share data space, the forked implementation would require the use of shared memory using shmget( ) and message queues as described above.. Note that this enhancement calls for implementing forking **<u>IN ADDITION TO</u>** pthreads, not instead of pthreads. (Two separate programs instead of a command-line option is acceptable.)
- Can you identify any appreciable difference running the program on a multicore computer as opposed to one with a single CPU? What if anything changes if you change the contention scope between threads (assuming you did the above optional enhancement also?)
- There are several adjustable parameters given in this assignment that may have an impact on the overall performance. (E.g. the ratio of children to buffers and the sleep times.) You could play with these parameters to try and find a correlation between the number of simultaneous contenders and the number of errors that occur without locking or the amount of delay introduced with locking, etc. Note that nWorkers must be at least 2 and strictly less than nBuffers.