CS 385 –Operating Systems – Fall 2011
Homework Assignment 5
Process Synchronization and Communications

Due: **Friday December 2 at 8:00 P.M. via Blackboard**

Overall Assignment

For this assignment, you are to write a simulation in which a number of processes 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. Interprocess communications (messages) will be used by child processes to notify the parent of the results of their work. Optional enhancements allow for the use of threads and the investigation of deadlocks as well as race conditions.

Man Pages

The following man pages will likely be useful for this assignment:

- ipc(5)
- ftok(3)
- ipcs(8)
- ipcrm(8)
- msgget(2)
- msgctl(2)
- msgsnd(2)
- msgrcv(2)
- semget(2)
- semctl(2)
- semop(2)
- shmget(2)
- shmctl(2)
- shmat(2)
- shmdt(2)
- sleep(3)
- usleep(3)
- nanosleep(2)
- time(2)
- gettimeofday(2)
- clock_gettime(2)

Program Overview

The general operation of the program will be as follows:

1. First a number of buffers will be allocated in shared memory, i.e. an array of nBuffer integers, where nBuffer is a PRIME number given on the command line. All of the buffers will be initialized to zero.

2. Next the program will create a shared message queue, which child processes can use to send messages back to the parent. (It may also be possible to send messages from parent to child, but that complicates issues a bit.)

3. Then the parent will fork off a number nChild processes, where nChild = nBuffer / 2. (Note that nBuffer must be at least 5 for the simulation to be meaningful.) The ID numbers for each child will range from 1 to nChild. (Note that ID numbers start at 1, not 0).

4. Each child will then perform a sequence of read and write operations on the shared buffers, accessing them in the following order:

   a. The first buffer accessed will be the child's ID number % nBuffer. (Which should just be the child's ID so long as nChild < nBuffer.)

   b. Successive accesses will jump by steps of the child's ID number, in a circular fashion. Child number 3 will access every 3rd buffer, child number 5 will access every 5th buffer, etc., wrapping back to zero on a mod basis when the numbers exceed nBuffer.

   c. The first two accesses of every three will be for reading, and the third for writing.

d. The cycle continues until each process has written into nBuffer of the buffers, which should involve writing into each buffer exactly once, (3 * nBuffer total accesses), since nBuffer is prime and nChild must be strictly less than nBuffer.

e. So for example, if nBuffer = 7, then the order of access for child number 3 would be as follows, where the underlined numbers are write access and the others are reads:

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

5. A read operation will consist of the following steps:

   a. Read the initial value in the buffer.

   b. Sleep for K * ID seconds, where ID is the child's ID number and K is a constant adjusted to make the simulation last a reasonable amount of time. An initial suggestion is to try K = 1 / nBuffer. (Note: use usleep( ) instead of sleep( ) for sub-second resolution. See man 3 sleep, man 3 usleep and man 2 nanosleep for details.)

   c. Read the value of the buffer again, and send a message to the parent if the value changed while the child was sleeping. The message should include (at a minimum) the ID of the child sending the message, the buffer which changed, the initial value, and the final value.

6. A write operation will consist of the following steps:

   a. Read the initial value of the buffer.

   b. Sleep for K * ID seconds. (See above.)

   c. Add 1 << (ID – 1) to the value read in step (a), and store the result back into the buffer. (E.g. child number 3 adds binary 000001 shifted left 2 places = 000100 = 4). Each child will add a single binary bit in a different position, which will make it possible to later identify which child(ren) were involved in any error conditions.

7. After a child has completed all write operations, it sends a final message indicating completion back to the parent, and then exits.

8. The parent reads messages from the message queue until it has read and processed the exit messages from each child process. Each message received is printed to the screen. Then the parent does a wait( ) on its children. (Or the parent could wait on the children one by one as each ending message is received.)

Message Queue Operations

- Messages, semaphores, and shared memory all use a similar set of commands and operations, following roughly the same sequence of operations shown below. (See the man page of ipc(5) for more details on the common features of these commands. Note carefully that the use of System V IPC is recommended for this assignment, which is not exactly the same as the POSIX versions. (Make sure to avoid the (P) sections of the man pages to avoid confusion.)

1. Acquire a "key" value, which is just a numeric "name" identifying this particular resource. There are three means of acquiring a key value: (A) Just pick a random number and hope no one else uses the same one, for example the last 4 or 5 digits of your student number. (B) Use IPC_PRIVATE, (0), for your key number, which means this resource will not be accessed by any other processes. This MIGHT work for common access by a parent and children of a fork for resources allocated before the fork takes place, but you'll have to try it to find that out. (C) The system call ftok(3) will generate a unique key number based on a unique filename

and a user chosen number (8 bit character). This is most appropriate if the resource needed relates to a file somehow (e.g. semaphores for file locking). Because key access is identical for all resources, it will not be discussed further under semaphores or shared memory.

2. Either create a new instance of the desired resource, or get access to one that has already been created, (presumably by some other process), using the unique key number acquired in step 1. The command to do this for message queues is msgget(2), which takes two arguments – the key described above and an integer of ORed bit flags. The low order 9 bits of this flag are read/write permission bits, so include 0600 as part of the flag for read/write permission by this user, or 0400 or 0200 for read-only or write-only permissions respectively. (Execute bits are ignored.) You should also OR in IPC_CREAT when creating a new instance of this resource (as opposed to getting your hands on a resource that has already been created.) msgget returns a queue ID, which can then be used for later operations.

3. Control the resource, which basically means getting and setting parameters which control its operation. The msgctl(2) command takes three arguments – the message queue ID, a command constant, and a pointer to a struct of type msqid_ds. The man page for msgctl(2) describes the legal commands, and ipc(5) describes the msqid_ds structure. Among other information, msqid_ds holds the number of bytes currently in the queue and the process IDs of the last sending and receiving process, as well as the last sending, receiving, or changing times.

4. Use the resource, which for messages means using msgsnd(2) and msgrcv(2).

   - msgsnd takes four arguments: The queue ID, a pointer to a msgbuf struct (see below), the size of the message, and a flags word.

   - msgrcv takes five arguments: The queue ID, a pointer to a msgbuf struct, the maximum size of message accepted, the type of message desired (see below), and a flags word.

   - The msgbuf struct contains two fields: a long int for the type of the message, and a char * (e.g. an array of characters). The type information is used with msgrcv to request specific types of messages out of the queue, or else to request whatever message is available. (Note: char * was commonly used as a generic pointer type in C before the void * type was added to the language. You can use typecasts to include any type of data in msgbuf, not only characters.)

   - The IPC_NOWAIT flag is used to specify whether sends or receives should be blocking or non-blocking.

- **Message Types:** When receiving messages, a process may grab the next available message of any kind, or search the queue for messages of a given “type”. For this assignment as written so far, messages are one-way traffic from children to parent, and since only the parent is reading messages, there is no need to specify types.

  Alternatively, to provide for two-way message traffic, each message can be given a “type”, where the type = the ID number for messages from children to the parent, and 100 + ID for messages from parent to child. Children search the message queue looking only for messages with type = 100 plus their ID, and the parent reads all messages less than or equal to nChild. See the man pages for msgsnd(2) and msgrcv(2) for more information on this idea.

- **Orphaned Message Queues:** There is a problem that you should be aware of regarding message queues, (and other IPC resources?), and fortunately there is also a solution:
    - Programs that create message queues and do not remove them afterwards can leave "orphaned" message queues, which will quickly consume the system limit of all available queues. Then no more can be created and your programs won't run. (And neither will anyone else's.)
    - The command "ipcs -q -t" will show you what queues are currently allocated and who has them, along with an ID number for each one.
    - Then the command "ipcrm -q ID" can be used to delete any that are no longer needed, where "ID" is the number you got from ipcs.
    - You should check this periodically, and certainly before you log off. You should also be sure to free up your allocated queues in your program before exiting, (using msgctl with the cmd type of IPC_RMID. You may also want to employ exception handling so that in the event of a program crash the queues get cleaned up on exit.)
    - Read the man pages for ipcs and ipcrm for more information.
    - The lab computers can also be safely rebooted if you discover the problem there and the queues belong to someone else.

Shared Memory Operations

- The relevant commands for shared memory are shmget(2), shmctl(2), shmat(2), and shmdt(2), analogous to msgget, msgctl, and msgsnd/msgrcv as described above.

- Shmget operates similarly to new or malloc, by allocating or getting shared memory. The three arguments are the key ID (as above), the number of bytes of memory desired, and flags as discussed above. However note that shmget does not return a memory address the way new and malloc do – Rather it returns an integer ID for this block of shared memory, similar to msgget and semget.

- Shmctl is used to examine and modify information regarding the shared memory. It will probably not be needed for this assignment.

- Shmat returns a memory address given a shared memory ID number, much like new or malloc. At this point the memory address can be used for accessing the shared memory just as any other address is used for accessing “normal” memory. Essentially shmat binds the shared memory to the user’s local address space.

- Shmdt detaches shared memory from the local address space, and is a companion to shmat in the same way that delete or free are companions to new or malloc.

Semaphore Operations

- The relevant commands for semaphores are semget(2), semctl(2), and semop(2), analogous to msgget, msgctl, and msgsnd/msgrcv as described above.

- Note that System V semaphores come as an array of 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 (as described for messages above), 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.)

- 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.

- 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 >>.

Additional Details

- Command Line Arguments – The first argument on the command line should be a prime integer, indicating how many buffers to use. The second argument is optional, and should be either "-lock" if semaphore locks are to be used or "-nolock" otherwise. Default operation should be "-nolock" if the second argument is not provided. You may add additional command-line arguments if you wish (e.g. for optional enhancements such as the number of children), but you must document them thoroughly.

Required Output

- All programs should print your name and CS account ID as a minimum when they first start.
- Each message received by the parent process from a child should be printed to the screen.
- The parent should keep track of how many read errors occur (reported by children), as well as how many write errors occur (buffers which do not hold the correct value at the end, which should be 2^nChild - 1 for all buffers.) Note the following:
    - In the case of read errors, the process(es) that wrote to the buffer while the reading child was sleeping can be determined by examining the bits of the difference between the initial and final values of the buffer.
    - In the case of write errors, the bits turned on in the difference between the correct value and the actual value indicates which children 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 nChild bit positions. Any 0 in any of these positions indicates a particular child whose write operation was over-written by another process.)

Other Details:

- A makefile is required, that will allow the TA to easily build your program, unless your program compiles easily with a standard g++ command. As always, you are free to develop wherever you wish, but the TA will be grading the programs based on their performance on the CS department Linux machines.

What to Hand In:

1. Your code, **<u>including a readme file, and a makefile if needed,</u>** 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, should be handed in **<u>at the beginning of class</u>** on the date specified above.
6. Make sure that your **<u>name and your CS account name</u>** 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:

- 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.
- 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.)
- 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.
- Provide a command-line option for implementing the assignment using pthreads instead of forking, and compare the performance of the two implementation methods. Since threads already

share data space, the threaded implementation could avoid the use of shared memory, and could possibly also avoid the use of message passing, though semaphores would still be required. Note that this enhancement calls for implementing pThreads IN ADDITION TO forking, not instead of forking. (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?)
- Try experimenting with random sleep times instead of times fixed by the ID number, to see if that has any effect on the results. Note that in order to analyze this properly will require running the program several times each way and applying statistical analysis to determine if the differences are significant or not. (Suggestion: Apply the student t-test, described in any book on statistics and probably available in Excel, (along with Excel help.)) Note: the timing will obviously change, so what is more interesting is the error rate as a function of the sleep times.
- There are several adjustable parameters given in this assignment that may have an impact on the overall performance. (Such as the ratio of children to buffers and the relative 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 nChild must be at least 2 and strictly less than nBuffer.
- The base assignment only requires messages to be sent from the children to the parent, which means the “type” feature of messages can be ignored. If you can think of a reason for the parent to send messages to the children also, then the type can be used to specify the intended recipient of any particular message. (E.g. when the child sends a message to the parent, the parent could then send a reply back acknowledging the message and perhaps giving further instructions.)