CS 385 –Operating Systems – Spring 2013
Homework Assignment 2 – Second Draft
Inter-Process Communications and Synchronization

Due: **Thursday March 12th at 3:00 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 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)
- ipcs(8)
- ipcrm(8)
- msgget(2)
- msgctl(2)
- msgsnd(2)
- msgrcv(2)
- shmget(2)
- shmctl(2)
- shmat(2)
- shmdt(2)
- semget(2)
- semctl(2)
- semop(2)
- fork( )
- exec( )
- pipe( )
- dup2( )
- wait( )
- rand( )
- srand( )
- RANDMAX
- time(2)
- gettimeofday(2)
- clock_gettime(2)
- sleep(3)
- usleep(3)
- nanosleep(2)

Command Line Arguments

For this assignment you will have to write TWO programs, named “`master`” and “`worker`”, having the following command line arguments:

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

    - nBuffers **<u>must be a positive PRIME integer</u>** for this program to run right.

    - nWorkers is the number of child processes that will exec the worker program, and must be strictly less than nBuffers.

    - sleepMin and sleepMax are positive **<u>real</u>** numbers, max greater than min, that define a range of sleeping times for the worker processes. The master process will begin by generating a series of nWorkers random numbers within this range. (Try 1.0 to 5.0 as a starting point.)

    - 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.

- `worker workerID sleepTime msgID shmID [ semID ]`
  - workerID is a unique identifier passed by the parent when forking and execing the children, starting with 1 for the first child and continuing to nWorkers for the last child.
  - sleepTime is the amount of time the workers will spend sleeping in each work cycle, generated by master as a series of random numbers in decreasing order. (See below.) Note that this is a floating point number, not an integer.
  - msgID, shmID, and semID are the integer ID numbers for the message queue, shared memory block, and semaphore block, as returned by msgget( ), shmget( ), and semget( ), described below.

Evolutionary Development

It is recommended to develop this program in the following steps, each of which will be described below in further detail.:

1. For the first step, the parent creates two pipes named 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 numbers in the range (sleepMin to sleepMax) and write them to the writing end of pipe1, one per line. (Seed the random number generator first, with a single call to srand( ). use randSeed if provided, or time(NULL) otherwise.) Close all unused pipe ends as soon as they are no longer needed, e.g. as soon as the writing is completed.
   - 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. For verification purposes, the random numbers should be printed before they are written to pipe1, and again as they are read from pipe2.
2. Next the parent should create a message queue, and then fork off nWorkers children.
   - Each child will exec the worker program, passing command line arguments for the workerID, sleepTime, and messageID for now. The workerID should be 1 for the first child, 2 for the second, etc. The sleepTime is the next random number in the sequence read from pipe2, and messageID is the value that the parent received from msgget( ). (I.e. each of the random numbers generated in step 1 will now be passed to the children as command-line arguments, from largest for child 1 to smallest for child nWorker.)
   - At this point the worker process should just write a startup message to the parent's message queue, and then exit. The startup message should include the workerID and sleepTime for this child.
   - After forking off all the children, the parent should read in nWorker messages from the message queue, report the results, and then wait for all of the children before exiting.
3. Create and test shared memory.
   - The parent will now allocate sufficient shared memory for an array of nBuffer integers, and zero it out before forking off the children. **Note: nBuffers must be a PRIME number larger than nWorkers for this program to eventually work right.**

    - Each worker process will now send a startup message, then write their ID to the buffer corresponding to their ID, (i.e. child 1 stores a 1 in buffers[1] and so on), and then send a cleanup message to the parent.

    - The parent will read messages and report results, waiting for each child when their cleanup message arrives. After the parent has waited for all children, it reports the contents of the shared memory.

4. Next the worker processes execute a round-robin cycle of reading from and writing to various places in shared memory. Without semaphores in place, there should be errors caused by race conditions, which are reported to the parent when the children detect them, and reported by the parent at the end.

- Each worker will perform a sequence of read and write operations on the shared buffers, accessing them in the following order:
    - The first buffer accessed will be the woker's ID number % nBuffers. (Which should just be the worker's ID so long as nWorkers < nBuffers.)
    - 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.
    - 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:

      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 was passed in as a command-line argument to the worker process. (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 send a message to the master process if the value changed while the worker was sleeping. The message should include (at a minimum) the ID of the worker sending the message, 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 in step (a), and store the result back into the buffer. (E.g. worker number 3 adds binary 000001 shifted left 2 places = 000100 = 4). 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.

- After a worker has completed all write operations, it sends a final message indicating completion back to the master, and then exits.

- The master should monitor the message queue for incoming messages, and report read errors as each worker process reports them, in a well-formatted fashion. (Note that due to race conditions, bits could either be added **or removed** while a worker is sleeping during the read cycle. As each worker sends in their final completion messages, the master should wait( ) for that particular child to exit.

- After all children have exited, the master 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.) If you do a bitwise exclusive OR (^) between the expected and actual answers, whatever bits are turned on in the result will indicate which worker process had their write operation erased due to a race condition. A bitwise AND (&) in a loop can test each bit one by one.

5. 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 assignment. The master needs to allocate a set (array) of nBuffers semaphores, with one assigned to protect each of the shared memory buffers.

    - The master process should use semctl( ) to initialize all the semaphores to 1 before forking off any children. The worker processes will then use semop( ) to implement wati( ) and signal( ) as appropriate.

    - See below for full details of semaphore operations.

Interprocess Communications, IPC

Shared memory, messages, and semaphores 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.) The basic steps for using IPC resources are as follows:

1. Acquire a “key” value, which is just a numeric “name” identifying this particular resource. There are several means of acquiring a key value, but for this particular application IPC_PRIVATE, (0), is probably the best choice. This is a bit of a misnomer, and really means that a number will be chosen by the system that is not already in use by any other process.

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, (e.g. IPC_PRIVATE.) The specific commands for this step are shmget(2), msgget(2), and semget(2) for shared memory, messages, and semaphores respectively.

3. Control the resource, which basically means getting and setting parameters which control its operation. The specific commands are shmctl(2), msgctl(2), and semctl(2).

4. Use the resource, which varies with the specific IPC resource being used. Commands used include shmat(2), msgsnd(2), msgrcv(2), and semop(2).

5. Clean up the resource when finished using it. See shmdt(2) and the separate section on orphaned resources below.

Orphaned IPC Resources

There is a problem that you should be aware of regarding IPC resources explained here in terms of message queues:

- 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.

Message Queue Operations

- The command to acquire message queues is msgget(2), which takes two arguments – the key described above, (IPC_PRIVATE), and an integer of ORed bit flags as discussed above.

- msgget returns a queue ID, which can then be used for later operations.

- 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.

- Messages are written to and read from queues using msgsnd(2) and msgrcv(2) respectively.

    - 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 * (Note that the pointer can really point to any kind of data, though it may need to be typecast if used with non-char data. Standard C does not have the void * type, so char *s are used instead.). 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.

  - 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, unless perhaps you want to distinguish exit messages from results or other types of messages. (You could also include that information as fields within the messages themselves.)

  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.

Shared Memory Operations

- Shmget operates similarly to new or malloc, by allocating or getting shared memory. The three arguments are the key ID (IPC_PRIVATE), the number of bytes of memory desired, 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.)

- 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, which is then used in further operations.

- 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.

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

- 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.)

- For compatibility purposes, you will 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.
- 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 message received by the master process from a worker should be printed to the screen.
- The master should keep track of how many read errors occur (reported 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 owrkers 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.)

Other Details:

- A makefile is required, that will allow the TA to easily build your programs. 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 machines. A sample makefile will be provided.

What to Hand In:

1. Your code, **<u>including a readme file, and a makefile,</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, may optionally be provided to the TA.
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?)
- 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 nWorkers must be at least 2 and strictly less than nBuffers.
- 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.)

OLD STUFF

- First get the shared memory working and verify children can all access it.
    - The parent will need to shmget and shmat before forking the children, and also zero out the memory. The children should then inherit the attached shared memory.
    - Fork off children in a for loop, so that each child knows its ID according to the value of the loop counter at the time they were forked off. (E.g. for(ID = 1; ID <= nChildren; ID++))
    - Have each child write it's ID number into a separate buffer and then exit.
    - After the parent forks off all children, it should then wait for all children, and then print the contents of all of the buffers.
- Then get message queues working, by having each child send a "goodbye" message containing its child ID and PID before exiting. The parent can either wait for all the children and then print all the messages, or read the messages and wait for each child as their exit message comes in. (Keep a counter of remaining children in the latter case.)
- Implement the reading / writing loops in the children's code. They will now send different messages if they discover errors during read cycles.
    - The parent now checks for and reports write errors after all children have completed.
    - If necessary, play with parameters until you observe errors due to race conditions. (In particular the sleep times for reads and writes. You might try K / ID instead of K * ID and/or separate K_R and K_W for read and write cycles respectively.
- Implement the locking with semaphores.
- Implement optional enhancements - See below.

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. 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.)

5. An optional command line argument will specify whether or not semaphores are to be used to protect the critical sections, i.e. the read and write operations described in points 5 and 6 above. (By default semaphores are not used, and you should observe errors caused by race conditions. If the "-lock" flag is present, then semaphores should be used and there should be no errors.) Optional enhancements listed below provide for timing analysis and reader-writer locks.

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 omitted otherwise. 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.
-