CS 361 – Computer Systems – Fall 2017
Homework Assignment 4 - Inter-Process Communications & I/O

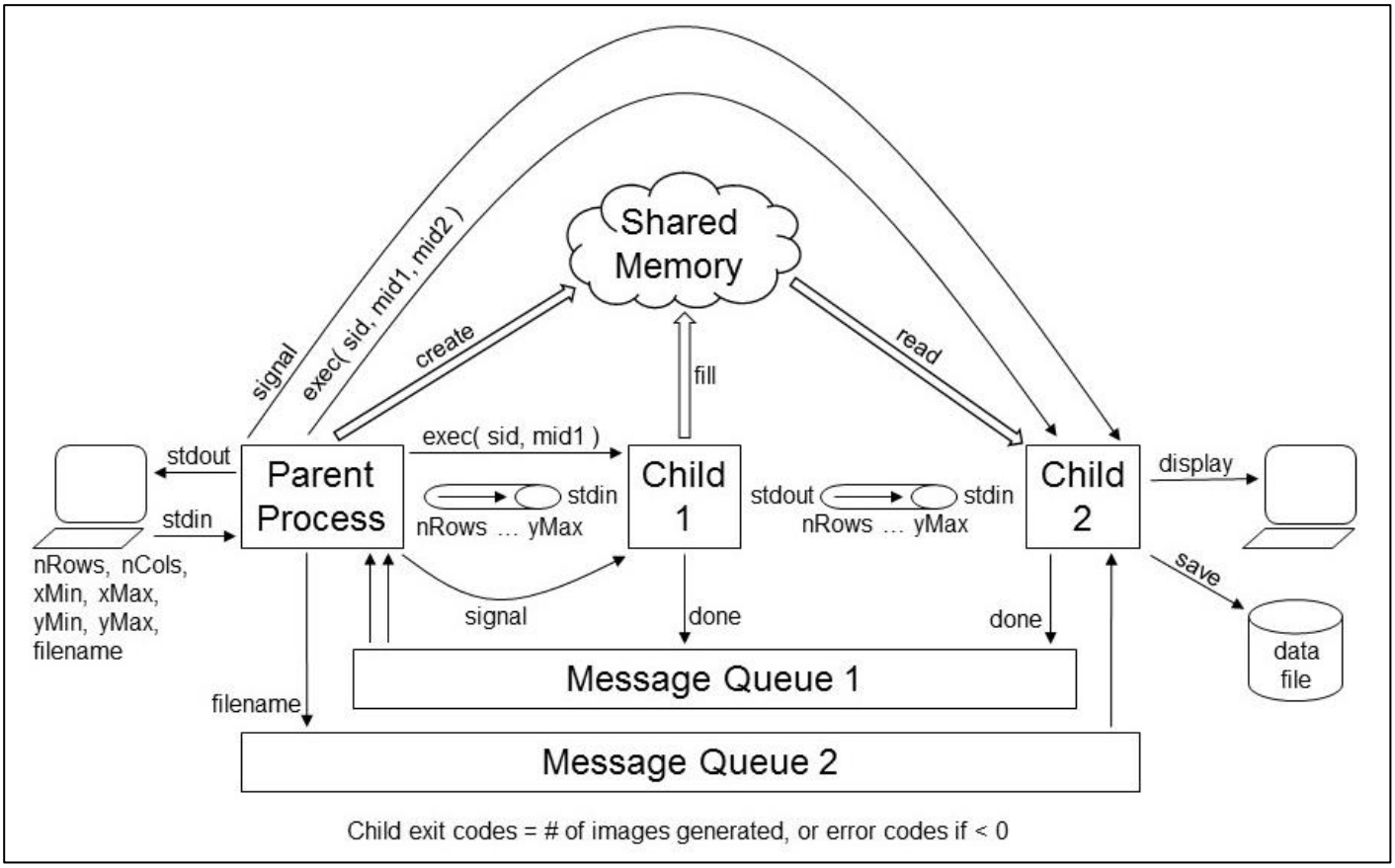
Overall Assignment

For this assignment, you are to write three programs that will work together using inter-process communications to create, display, and store Mandelbrot fractal images. More specifically, the processes will use fork / exec, pipes, message queues, shared memory, signals, and process exit codes to coordinate their efforts and communicate information. As shown in the structure diagram below:

- The parent process is responsible for interacting with the human, and for creating and later terminating the inter-process communications structures and child processes.
- The first child performs the Mandelbrot calculations and stores the results in shared memory.
- The second child displays the results to the screen and saves the data to a file for further processing, e.g. for graphical display in Matlab or some other program.

Sample Mandelbrot Images

Overall Process Diagram



Parent's To Do List (Final Version)

1. Create pipe(s)
2. Create message queue(s)
3. Create shared memory
4. Set signal handler for SIGCHLD (In case children exit due to error conditions)
5. Fork children
6. Close unused pipe ends
7. While true:
    a. Read problem info from keyboard
    b. If user is not done yet:
        i. Write filename to message queue 2
        ii. Write xMin, xMax, yMin, yMax, nRows, nCols, and maxIters to pipe
        iii. Listen for done messages from both children
    c. else:
        i. Send SIGUSR1 signals to both children
        ii. Wait for both children, and report exit codes.
8. Report final results and exit

First Child's To Do List (Final Version)

1. Close unused pipe ends
2. Redirect stdin and stdout with dup2. (Close these original pipe ends after they've been copied.)
3. exec mandelCalc with shmid and msgid as arguments. (Last step in mandelbrot program.)
4. Parse shmid and msgid from command line arguments. (First step when mandelCalc launches.)
5. Set a signal handler to catch SIGUSR1
6. While true:
    a. read xMin, xMax, yMin, yMax, nRows, nCols, and maxIters from stdin
    b. Implement Mandelbrot algorithm to fill shared memory. (See below.)
    c. Write xMin, xMax, yMin, yMax, nRows, and nCols to stdout
    d. Write done message to message queue 1
7. When SIGUSR1 arrives, exit with non-negative exit code equal to number of images calculated.
8. (If any errors occur, exit with a different negative exit code for each potential problem.)

Second Child's To Do List (Final Version)

1. Close unused pipe ends
2. Redirect stdin with dup2. (Close the original pipe end after it has been copied.)
3. exec mandelDisplay with shmid and 2 msgids as arguments. (Last step in mandelbrot program.)
4. Parse shmid and 2 msgids from command line arguments. (First step in mandelDisplay.)
5. Set a signal handler to catch SIGUSR1
6. While true:
    a. read xMin, xMax, yMin, yMax, nRows, and nCols from stdin
    b. Read filename from message queue 2 and open file. (If fail, don't save to file.)
    c. Read data from shared memory and display image on screen
    d. If file opened properly, save data to a data file. (May be combined with step c.)
    e. Close file when all data have been written.
    f. Write done message to message queue 1.
7. When SIGUSR1 arrives, exit with non-negative exit code equal to number of images calculated.
8. (If any errors occur, exit with a different negative exit code for each potential problem.)

Man Pages

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

- pipe(2)
- fork(2)
- exec(3)
- dup2(2)
- ipc(5)
- ipcs(8)
- ipcrm(8)
- msgget(2)
- msgctl(2)
- msgsnd(2)
- msgrcv(2)
- shmget(2)
- shmctl(2)
- shmat(2)
- shmdt(2)
- signal(2)
- kill(2)
- open(2) or
- fopen(3)
- fdopen(3)
- fflush(3)

Pipes

- The pipe( ) system call fills in an array of two integers, (which must have been previously declared and allocated), with two file descriptors pointing to the same open file - one for reading and one for writing.
- Pipe is typically called by a parent process prior to forking. Because child processes inherit all open files from their parent, the two processes can now communicate via this shared file.
- Alternatively, if two children are forked off after the pipe call, then the two children can use the shared pipe file to communicate.
- In any case, all pipe ends that are not being used must be closed by their respective processes. For example, if the intention is that the parent write information to the pipe for the child to read, then the parent should close the reading end of their pipe and the child should close the writing end of their pipe. For this assignment there are 4 pipe ends and 3 processes - Each process will need to close any of the pipe ends that they will not be using.

Fork, Exec, and Duplicating File Descriptors

- When the parent forks off children, those processes inherit the open files of the parent, and in particular, the file descriptors returned by the pipe( ) system call.
- After the child execs, however, the new process will expect to read from standard input and write to standard output, and will be unaware of the file descriptors returned by pipe( ).
- It will therefore be necessary for the child process to duplicate the file descriptors returned by pipe into the process file table in the place of stdin and/or stdout, using the dup2( ) system call.
- Note that after a pipe file descriptor has been copied, then the original needs to be closed.
- The executable names for the two children should be mandelCalc and mandelDisplay, with command line arguments as follows:

```
mandelCalc shmID msgqID
mandelDisplay shmID msgqID1 msgqID2
```

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. For this assignment the best approach will be to simply use IPC_PRIVATE as a key value.

    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.

   For this assignment the only use of msgctl that you are likely to need is the comnmand constant IPC_RMID to remove the resource when the program is done, for which the third argument can be NULL. See discussion below on orphaned resources for more information. (Hint: You will probably want to set up signal handlers to release all of your IPC resources in the case of program exit due to interrupts such as control-C from the keyboard.)

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 is not specifically defined, but can be any user-defined struct in which the first field is a long int, where that long int is used to specify the type of the message.

     Note: The man pages for msgsnd/msgrcv indicate that the msgbuf struct has a char * as its second argument, because char * was commonly used as a generic pointer type in C before the void * type was added to the language. You can actually include any type of data you want in msgbuf, with the exception of pointer types. (And the struct doesn't have to be named msgbuf, and the first field doesn't need any particular field name.)
   - 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 that message queues can theoretically hold messages of different types, which are not of the same size.)
   - The IPC_NOWAIT flag is used to specify whether sends or receives should be 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 (or vice-versa), and since only one process reads any given message queue, there is no need to specify types when receiving messages, or to even define more than one type. (A single defined struct can serve both to deliver the filename to the second child and to deliver "done" messages back from children to the parent.)

  Alternatively the type field could be used to store heterogeneous message types in a single queue, or as a form of "addressing" to indicate which process should receive which message from a shared message queue. We will not need either of these features for this assignment.

- **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 **no one** can run programs using messages.
    - The command "ipcs -q -t" or "ipcs -a" 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. Note that shmget does not return a memory address the way new and malloc do – 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.

Accessing Shared Memory

The data in shared memory will be interpreted as a two-dimensional array, except the system does not know how many elements are on each row, so it is up to you to do the calculations. Given that the data is stored in row-major order (each row stored together as a unit, with row N+1 following row N), and given that there are nCols elements in each row, then to access the data in row R and column C of the array the correct formula is:

$$* (data + R * nCols + C)$$

where data is the address of the beginning of the data storage. (This is how C/C++ accesses two-dimensional arrays behind the scenes, and why row and column indices start at 0. Think about it a little if you haven't seen this before.)

Mandelbrot Algorithm

The Mandelbrot set is based on a very simple series involving complex numbers, defined by:

$$Z_{n+1} = Z_n^2 + C$$

where Z and C are both complex, $Z_0 = 0 + 0i$, and C is an arbitrary point in the complex plane. If Z_n remains within a circle of radius 2.0 from the origin in the complex plane for all values of n from 0 to infinity, then C is a member of the Mandelbrot set, and if Z_n ever leaves that circle, then it is not.

Now in practice we are not willing to count to infinity, so we set some upper limit, and if Z_n is still inside the circle after that many iterations, then we count it as being in the set. If, on the other hand, Z_n leaves the circle, then we color the pixel with a color based on n, the number of iterations required before Z_n left the circle. As a first step we will just store the values of n into an array, or -1 for points in the Mandelbrot set.

So here then is the algorithm for calculating an array of Mandelbrot values, to be displayed later:

Given:

- xMin, xMax, yMin, yMax defining the range of the complex plane to be displayed. (Choose X from -2.0 to 2.0 and Y from -1.5 to 1.5 to capture the entire set, or smaller regions to zoom in on a particular area of interest.)
- maxIter, the maximum number of iterations we are willing to count before we give up and declare C to be in the Mandelbrot set. 100 is probably a reasonable value for this HW.
- nRows and nCols indicating how many pixels (chars in our case) tall and wide the image is to be. 20 x 50 is probably a good starting point for this assignment.
- An integer array "data", containing at least nRows rows and nCols cols. (Note: For this particular assignment we will be accessing data[i][j] as *(data + i * nCols + j). This will be explained in further detail in class.)

Calculate:

- deltaX = (xMax - xMin) / (nCols - 1)
- deltaY = (yMax - yMin) / (nRows - 1)

Then loop through the array, determining for each position whether the corresponding point is in the Mandelbrot set, or if not, how many iterations of the series are required before it leaves the circle:

```
for( r from 0 to nRows - 1 ) {
     Cy = yMin + r * deltaY
     for( c from 0 to nCols - 1 ) {
          Cx = xMin + c * deltaX
          Zx = Zy = 0.0
          for( n = 0; n < maxIter; n++ ) {
               if( Zx * Zx + Zy * Zy >= 4.0 )
                    break
               Zx_next = Zx * Zx - Zy * Zy + Cx
               Zy_next = 2.0 * Zx * Zy + Cy
               Zx = Zx_next
               Zy = Zy_next
          }
          if( n >= maxiter ) store -1 in data[ r ][ c ]
          else store n in data[ r ][ c ]
     }
}
```

Algorithm for Displaying Mandelbrot Values

The simplest approach for displaying the mandelbrot data is to create an array of colors[nColors], and then color each pixel with the color stored at colors[n % nColors]. In the case of ASCII art, as in this HW assignment, colors[nColors] will actually be a char array of different symbols, and we will print the char from array position n % nColors. More specifically for this particular assignment:

```
Let char colors[ nColors ] = ".-~:+*%O8&?$@#X", where nColors = 15
For all r from nRows - 1 to 0, all c from 0 to nCols - 1:
      n = *( data + r * nCols + c ) /* = data[r][c] */
      if( n < 0 ) print a space // Typically black with real pixels.
      else print colors[ n % nColors ]
```

Note that the row numbers decrease as printing proceeds, because the top row (first row) of the printout corresponds to row nRows - 1, and the bottom row (last row) corresponds to row 0.

Output File Format

The second child should write the data that it reads from shared memory into an output file, with a filename specified by the user and which the second child reads from the second message queue. The file should be a plain ascii text file, with one row of output per row of the image, with values on the row separated by spaces.

Signals, Exit Codes, and Wrapping Up

The assignment is designed so that the child processes will run in infinite loops, doing a hanging read on standard input (the pipes) until problem data arrives, then processing it and going back to the hanging read. The parent process waits for the children on a hanging read of the message queue, and then asks the user to enter another set of parameters.

When the user indicates that they are done and want to quit, the parent sends a signal SIGUSR1 to each of the children. The children should catch the signal in a signal handler, and exit with an exit code indicating the number of parameter sets that they processed. The parent should wait for the children after sending them the signal, and report their exit codes.

In the event of early termination, the children should exit with negative exit codes. The parent should have a signal handler for SIGCHLD that will wait for the children and report the results.

Evolutionary Development

The best way to do this assignment is in stages, getting one stage working before moving on to the next. The following steps are recommended, or you can select your own approach:

1. Verify that the parent can fork off two children, and that the children can exec mandelCalc and mandelDisplay. For now pass some simple integers to them as arguments, have the children parse and report their arguments, and then exit. The parent should wait for the children and report their exit values.
2. Next have the parent create two pipes before forking off the children and that they can read and write using the pipes before execing. The first thing each of the processes need to do after the forking is to close the pipe ends that they do not need. Then have the parent write something into the first pipe, and have the first child read and report it. Then have the first child write to the second pipe and the second child read and report it. (Parent will probably use fdopen( ) and fprintf( )).

3. For the next step, have the children use dup2 to copy the pipe ends before execing, and then have the new child processes read from standard in and write to standard out. (Could combine this with step 2.)

   (Steps 4 and 5 could be done in either order.)

4. Once pipe operations are confirmed, get message queues working. The parent should create the message queues and the message queue ID(s) should be passed to the children as command line arguments. The parent should write a filename to the second message queue before writing to the pipe. The second child should read and report the filename from the second queue either before or after reading from the pipe and shared memory. Each child should write a "done" message to the first message queue before exiting. The parent should read the messages and report their results before waiting for the children to exit.
5. Next work on shared memory. Have the parent create the shared memory, and pass the shared memory ID as an argument to the children when execing. The first child should write something to the shared memory, and the second child should read it and report the result. (At this stage a single int will suffice. For the final program a large number of ints will be necessary, which will set an upper limit on the product of nRows times nCols.)
6. Once all the components have been implemented and tested, start modifying the processes to perform the tasks of generating and displaying the Mandelbrot set.

Required Output

- All programs should print your name and ACCC user name as a minimum when they first start.
- Beyond that, the second child should print out the Mandelbrot image to the screen as described above, and save the raw data into a data file for further analysis by other programs, e.g. Matlab.

Other Details:

- Unless the TA specifies otherwise, please name your main program mandelbrot-<<acccName>>.c or .cpp and your child programs mandelCalc-<<acccName>>.c or .cpp and mandelDisplay-<<acccName>>.c or .cpp, where <<acccName>> is your ACCC username. Set up your makefile so that the resulting executables are just mandelbrot, mandelCalc, and mandelDisplay respectively.
- The TA must be able to build your programs by typing "make". If that does not work using the built-in capabilities of make, then you need to submit a properly configured makefile along with your source code. As always, you are free to develop wherever you wish, but the TA will be grading the programs based on their performance on the Official standard grading machines.

What to Hand In:

1. Your code, **<u>including a makefile, and a readme file,</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, as well as any differences between the printed and electronic version of your program.
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, (such as hand-drawn sketches or diagrams) may be handed in **to the TA** on or shortly after the date specified above. Any hard copy must match the electronic submission exactly.
6. Make sure that your **name and your ACCC 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:

- Display scale information on the edge of displayed images (but not in stored data files.)
    - Label the display with Ymax and Ymin along the left edge and Xmin and Xmax along the bottom row.
- Visualize your data files using Matlab or some other program.
- Learn about Joulia sets, and offer them to the user as an alternative to Mandelbrot sets.

References

- http://beej.us/blog/data/mandelbrot-set/ - Accessed 17 Feb 2014