Inter-Thread Communication

Average System Execution Time

X = 500

(Average timing measurement data and its standard deviation for (N,B,P,C) = (398, 8, 1, 3) are highlighted in red.)

N	B	P	C	Time
100	4	1	1	0.00056
100	4	1	2	0.000607
100	4	1	3	0.000778
100	4	2	1	0.000716
100	4	3	1	0.00078
100	8	1	1	0.000561
100	8	1	2	0.00057
100	8	1	3	0.000697
100	8	2	1	0.000692
100	8	3	1	0.000798
398	8	1	1	0.000998
398	8	1	2	0.000936
398	**8**	**1**	**3**	**0.001029**
398	8	2	1	0.00113
398	8	3	1	0.001421

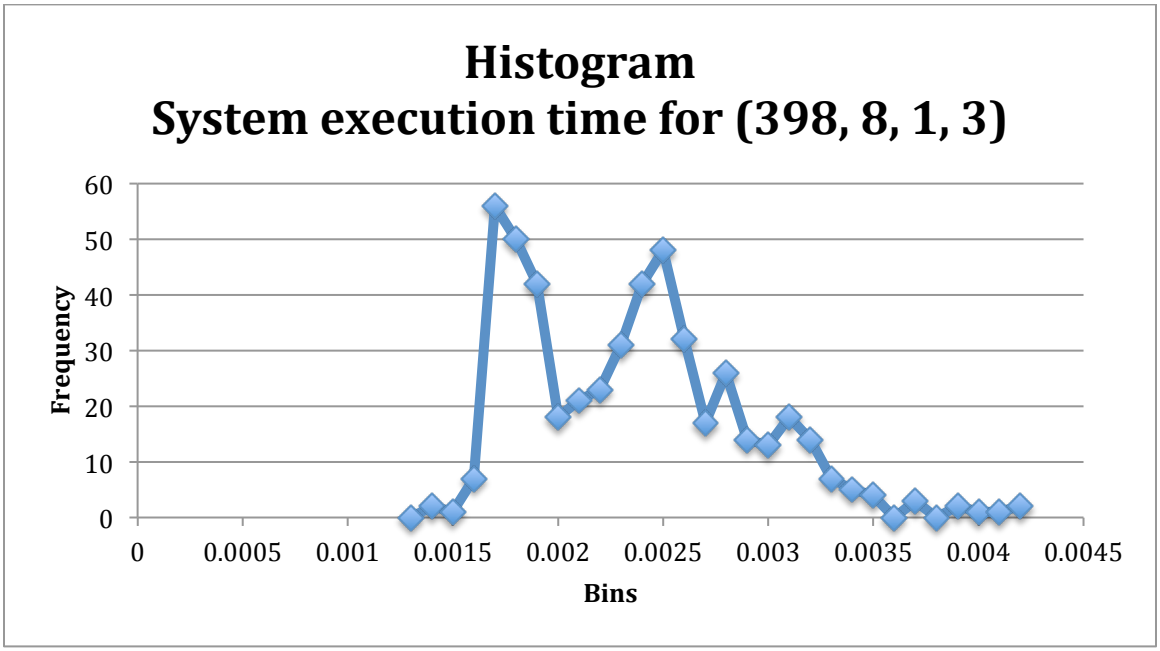
Standard Deviation of System Execution Time

N	B	P	C	Time
100	4	1	1	0.000456
100	4	1	2	0.000552
100	4	1	3	0.000725
100	4	2	1	0.000657
100	4	3	1	0.000655
100	8	1	1	0.00055
100	8	1	2	0.000553
100	8	1	3	0.000644
100	8	2	1	0.000665

100	8	3	1	0.000712
398	8	1	1	0.000647
398	8	1	2	0.000613
398	**8**	**1**	**3**	**0.000642**
398	8	2	1	0.000691
398	8	3	1	0.000693

System execution time for (N, B, P, C) = (398, 8, 1, 3), X=500

Data Bins	Frequency
0.0013	0
0.0014	2
0.0015	1
0.0016	7
0.0017	56
0.0018	50
0.0019	42
0.002	18
0.0021	21
0.0022	23
0.0023	31
0.0024	42
0.0025	48
0.0026	32
0.0027	17
0.0028	26
0.0029	14
0.003	13
0.0031	18
0.0032	14
0.0033	7
0.0034	5
0.0035	4
0.0036	0
0.0037	3
0.0038	0
0.0039	2
0.004	1
0.0041	1
0.0042	2



Inter-Process Communication

Average System Execution Time

X = 500

N	B	P	C	Time
100	4	1	1	0.000474
100	4	1	2	0.000711
100	4	1	3	0.000949
100	4	2	1	0.00069
100	4	3	1	0.000911
100	8	1	1	0.000495
100	8	1	2	0.000693
100	8	1	3	0.000904
100	8	2	1	0.000687
100	8	3	1	0.000919
398	8	1	1	0.000498
398	8	1	2	0.000735
398	**8**	**1**	**3**	**0.000917**
398	8	2	1	0.000707
398	8	3	1	0.00089

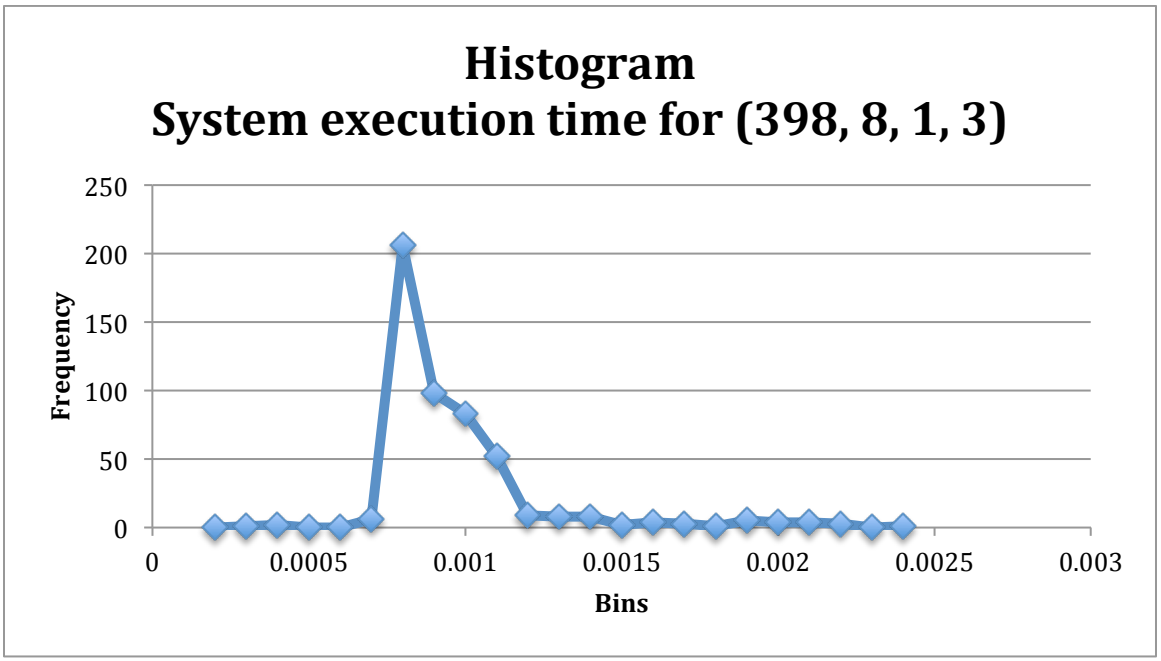
Standard Deviation of System Execution Time

N	B	P	C	Time
100	4	1	1	0.000149
100	4	1	2	0.000239
100	4	1	3	0.000289
100	4	2	1	0.000199
100	4	3	1	0.000264
100	8	1	1	0.000185
100	8	1	2	0.000197
100	8	1	3	0.00024
100	8	2	1	0.000203
100	8	3	1	0.000253
398	8	1	1	0.000196
398	8	1	2	0.000274
398	**8**	**1**	**3**	**0.000268**
398	8	2	1	0.000225
398	8	3	1	0.000233

System execution time for (N, B, P, C) = (398, 8, 1, 3), X=500

Data Bins	Frequency
0.0002	0
0.0003	1
0.0004	2
0.0005	0
0.0006	0
0.0007	6
0.0008	206
0.0009	98
0.001	83
0.0011	52
0.0012	9
0.0013	8
0.0014	8
0.0015	2
0.0016	4
0.0017	3
0.0018	1
0.0019	5
0.002	4
0.0021	4

0.0022	3
0.0023	0
0.0024	1



Discussion

Comparing the times of the inter-process communication method versus the inter-thread communication method, our multi-process implementation has the marginal edge over the multi-threading method. Moreover, we observed a significantly longer system execution time for the multi-threading method when multiple producers were involved, for a lot of items (ie. N = 398, B = 8, P = 2, 3, C = 1).

The advantages of the multi-threading approach are many, the more important ones being that:-

- Data is shared amongst the other threads in the process, which is quite beneficial for tasks that need large amounts of data to be shared
- It is faster to switch between threads than it is to switch between processes, allowing for faster task-switching
- Threads are faster to start than processes
- Use less resources than processes

Some disadvantages of multi-threading include:-

- All threads in the same program have to run the same executable
- If there is an error in one thread that causes data corruption, then that error affects all the other threads as well
- Debugging programs with this approach are a little more difficult due to the possibilities of synchronization errors, data corruptions, etc.
- Requires special handling in the form of mutexes, semaphores, etc.

The multi-processing approach also has advantages such as:-

- Multiple processes are less likely to suffer from concurrency bugs
- Child processes can run different executables using *execvp*
- A faulty process does not affect all the other processes

The multi-processing approach does have some significant disadvantages, such as:-

- Copying memory for a new process adds performance overhead
- Relatively more expensive method, and requires a larger main memory

Appendix

Inter-Thread Communication

Threads.c

```
#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/time.h>

#include <mqueue.h>

#include <sys/stat.h>

#include <time.h>

#include <signal.h>

#include <sys/wait.h>

#include <stdbool.h>
```

```
#include <math.h>
#include <pthread.h>
#include <string.h>
#include<semaphore.h>
#define QUEUE_SIZE  6
#define _XOPEN_SOURCE 600

//typedef struct producePara;
//typedef struct consumePara;
void* produce(void *ptr);
void* consume(void *ptr);
//use unnamed semaphore
sem_t ConsumeSem;
sem_t ProduceSem;
sem_t lock;
int *Buffer = NULL;
int count = 0;
pthread_mutex_t lock_mutex = PTHREAD_MUTEX_INITIALIZER;
int P;
int C;
int N;
int B;
int countC =0;

double getTime(){

      struct timeval tv;
```

```
	double t1;

	gettimeofday(&tv, NULL);
	t1 = tv.tv_sec + tv.tv_usec/1000000.0;
	return t1;

}

struct producePara{
	int PID;
	int MaxNum;
	int NumOfP;
	int Buffersize;
	int NumOfC;

}producePara;

struct consumePara{
	int CID;
	int size;
	int Buffersize;

}consumePara;
```

```
// we need to implement both thread and process and compare the
two

//when a Process run due to system scheduling, it determines
whether it's C or P and perform send/receive.

//also need a function to see if the squareroot of itself is a
int

//need to manually set up buffer for the thread method



//for process method we fork as many as we want, and then assign
each child process to either C or P

//according to the assignment when the process starts we know
it's either C or  P and then we produce/consume









int is_square(int Num){


      double a;

      if(Num == 0){

      return 0;}

      double c = (double) Num * 1.00;

      a = sqrt(c);

      while(a>1){

      a--;
```

```
      }
      if(a == 1){


      return 0;
      }
      else{return 1;}


}
void* produce(void* ptr){//MaxNum = N-1;
//int PID, int *buffer, int MaxNum, int NumOfP
      ////printf("in produce\n");
      struct producePara *data;
    data = (struct producePara*) ptr;
      //////printf("Check buffer\n");
      int i;
      int n = data->PID;
      int semValue;
      while(n < data->MaxNum){
      //    printf("producer%d in while\n",data->PID);
            //printf("n = %d, data->MaxNum = %d\n", n, data-
>MaxNum );


            sem_getvalue(&ProduceSem, &semValue);
      //    printf("producer%d has the produceSem value of %d
\n",data->PID, semValue);
            sem_wait(&ProduceSem);
      //    printf("producer%d passed semaphore\n",data->PID);
```

```
            pthread_mutex_lock (&lock_mutex);
      //    printf("producer locked\n");

            for(i = 0; i <data->Buffersize; i++ ){
      //    printf("in for, buffer[%d] = %d\n", i, Buffer[i]);
                  if(Buffer[i] == -1){

                        //printf("producer%d locked mutex\n",data-
>PID);
                        Buffer[i] = n;
      //                printf("Buffer[%d] = %d Produced\n", i,
Buffer[i] );

                        break;
                  }
            }
            pthread_mutex_unlock (&lock_mutex);
//          printf("producer unlocked\n");


            sem_post(&ConsumeSem);
            //printf("producer%d unlocked sema\n",data->PID);
            n = n + data->NumOfP;
      //    printf("n = %d\n",n);
      }

      //printf("done produce\n");
```

```
	return NULL;
	/**/
}
void* consume(void* ptr){
	//int CID, int *buffer, int size

	struct consumePara *data;
    data = (struct consumePara *) ptr;
	//printf("consumer %d Entered\n",data->CID);
	countC++;
	int semValue;
	////printf("countC =  %d whiled\n",countC);
	while(count < data->size){
		//printf("consumer %d whiled\n",data->CID);



			//printf("In consumer loop\n");
			int i;
			int flag = 0;
			sem_getvalue(&ConsumeSem, &semValue);
	//		printf("consumer%d has the ConsumeSem value of %d
\n",data->CID, semValue);
			sem_wait(&ConsumeSem);
			//printf("consumer%d passed Semaphore\n",data-
>CID);
		/**/for(i = 0; i <data->Buffersize; i++ ){
			pthread_mutex_lock (&lock_mutex);
//			printf("consumer %d locked\n",data->CID);
```

```
                        if(Buffer[i] != -1){




                                //printf("buffer[%d] = %d\n",i,
Buffer[i]);


                                if(is_square(Buffer[i]) == 0){


                                        printf("%d, %d, %d\n", data-
>CID, Buffer[i], (int)sqrt(Buffer[i]));
                                }


     //                         printf(" Buffer[%d] = %d is consumed
,\n", i, Buffer[i]);
                                Buffer[i] = -1;
                                count++;
                                sem_post(&ProduceSem);




                        }
                        pthread_mutex_unlock (&lock_mutex);
//                      printf("consumer%d unlocked\n",data->CID);



                }


     //         printf("count = %d\n",count);
```

```
	}
		if(count >= data->size-1)
			{
				//printf("semaphore added\n");
				sem_post(&ConsumeSem);
			}


return NULL;
}
int main (int argc, char *argv[])
{
	//////printf("1\n");

	double TimeStamp1;
	double TimeStamp2;
	TimeStamp1 = getTime();
	char *Bn;
	B = strtol(argv[2], &Bn, 10);
	char *Nn;
	N = strtol(argv[1], &Nn, 10);
	char *Pn;
	P = strtol(argv[3], &Pn, 10);
	char *Cn;
```

```
C = strtol(argv[4], &Cn, 10);
sem_init(&ProduceSem, 0, B);
sem_init(&ConsumeSem, 0, 0);
//////printf(" N = %d\n", N);
//////printf("B = %d\n", B);
//////printf("P = %d\n", P);
//////printf("C = %d\n", C);
pthread_t threadid[P+C];
Buffer = malloc(sizeof(int)*B);
int i;
for(i = 0; i < B; i ++){
      Buffer[i] = -1;
}

int a;
struct producePara data[P];

for(a = 0; a < P; a++){
      //////printf("a\n");
      data[a].PID = a;
      data[a].Buffersize = B;
      data[a].MaxNum = N;
      data[a].NumOfP = P;
      data[a].NumOfC = C;
      //pthread_t thread;
      int threadNum;
```

```
            threadNum = pthread_create(&threadid[a], NULL,
&produce,(void *) &data[a]);

            //////printf("Thread = %d\n",threadNum );



      }

      struct consumePara data1[C];

      int b;

      for(b = 0; b < C; b++){



            data1[b].CID =b;

            data1[b].size = N;

            data1[b].Buffersize = B;

            //pthread_t thread;

      //    ////printf("data->CID = %d\n",data1.CID );

            int threadNum;

            threadNum = pthread_create(&threadid[b+P], NULL,
&consume,(void *) &data1[b]);

            //threadid[b+P] = thread;



      }



      /*while(1){}*/

      //printf("quitting" );

      int c;

      for(c =0 ; c <C+P; c++ ){

            pthread_join (threadid[c], NULL);

      }

/* Make sure the second thread has finished. */
```

```
      TimeStamp2 = getTime();

      printf ("System Excution Time:   %.6lf  seconds\n",
TimeStamp2 -TimeStamp1);



      return 0;

}
```

Inter-Process Communication

main.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <math.h>

/* Spawn a child process running a new program.  PROGRAM is the
name
   of the program to run; the path will be searched for this
program.
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list.  Returns the process id
of
   the spawned process.  */
int spawn (char* program, char** arg_list)
{
        pid_t child_pid;

        /* Duplicate this process.  */
        child_pid = fork ();
        if (child_pid != 0)
                /* This is the parent process.  */
                return child_pid;
        else {
                /* Now execute PROGRAM, searching for it in the
path.  */
                execvp (program, arg_list);
```

```
                /* The execvp function returns only if an error
occurs.  */
                fprintf (stderr, "an error occurred in
execvp\n");
                abort ();
        }
}

int main(int argc, char *argv[])
{
      struct timeval tv;
      double t1;
      double t2;


        int N = atoi(argv[1]);
        int B = atoi(argv[2]);
        int P = atoi(argv[3]);
        int C = atoi(argv[4]);
        int M = N;

        //Array and counter to store all the process IDs
         int pids[P+C];
         int pidcount = 0;

       //Main message queue
        mqd_t qdes;

        char  qname[] = "/mailbox1_srajguru";

        mode_t mode = S_IRUSR | S_IWUSR;
        struct mq_attr attr;

        attr.mq_maxmsg  = B;
        attr.mq_msgsize = sizeof(int);
        attr.mq_flags   = 0;            /* a blocking queue  */

        qdes  = mq_open(qname, O_RDWR | O_CREAT, mode, &attr);
        if (qdes == -1 ) {
                perror("mq_open() failed");
                exit(1);
        }

//Producer counter, single message queue
        mqd_t prodmq;

        char  prodqname[] = "/prodmq";

        mode_t modepmq = S_IRUSR | S_IWUSR;
        struct mq_attr pmqattr;

        pmqattr.mq_maxmsg  = 1;
        pmqattr.mq_msgsize = sizeof(int);
```

```
        pmqattr.mq_flags   = 0;            /* a blocking queue
*/

        prodmq  = mq_open(prodqname, O_RDWR | O_CREAT, modepmq,
&pmqattr);
        if (prodmq == -1 ) {
                perror("mq_open() failed here");
                exit(1);
        }


        if (mq_send(prodmq, (char *)&N, sizeof(int), 0) == -1)
           {
                        perror("mq_send() failed");
            }

//Consumer counter, single message queue
        mqd_t consmq;

        char  consqname[] = "/consmq";

        mode_t modecmq = S_IRUSR | S_IWUSR;
        struct mq_attr cmqattr;

        cmqattr.mq_maxmsg  = 1;
        cmqattr.mq_msgsize = sizeof(int);
        cmqattr.mq_flags   = 0;            /* a blocking queue
*/

        consmq  = mq_open(consqname, O_RDWR | O_CREAT, modecmq,
&cmqattr);
        if (consmq == -1 ) {
                perror("mq_open() failed");
                exit(1);
        }


        if (mq_send(consmq, (char *)&M, sizeof(int), 0) == -1)
           {
                        perror("mq_send() failed");
            }

//Fork producer processes

        int i;

        gettimeofday(&tv, NULL);
          t1 = tv.tv_sec + tv.tv_usec/1000000.0;

        for(i=0;i<P;i++)
        {
           //Store process ID
        pids[pidcount] = i;
```

```
           pidcount++;

         //Pass process ID as an argument to producer.c, cast it
to a char pointer
           int a = i+48;
           char* t = (char*)&a;

           char* arg_list[] = {"./producer.out", argv[1],
argv[2], argv[3], argv[4], t, NULL};
                spawn("./producer.out", arg_list);
         }

//Fork consumer processes
         int j;
         for(j=0;j<C;j++)
         {
           pids[pidcount] = j;
           pidcount++;
           int b = j+48;
           char* u = (char*)&b;
           char* arg_list2[] = {"./consumer.out", argv[1],
argv[2], argv[3], argv[4], u, NULL};
                spawn("./consumer.out", arg_list2);
         }

       gettimeofday(&tv, NULL);
         t2 = tv.tv_sec + tv.tv_usec/1000000.0;

         if (mq_close(qdes) == -1) {
                perror("mq_close() failed");
                exit(2);
         }

         if (mq_close(prodmq) == -1) {
                perror("mq_close() failed");
                exit(2);
         }

         if (mq_close(consmq) == -1) {
                perror("mq_close() failed");
                exit(2);
         }

     //Wait for all processes
       int k;
       for (k=0;k<pidcount;k++)
       {
            wait (pids);
       }

     //Unlink message queues
       if(WIFEXITED (*pids))
       {
```

```
            if (mq_unlink(qname) != 0) {
            perror("mq_unlink() failed");
            exit(3);

            }
}


if (mq_unlink(prodqname) != 0) {
            perror("mq_unlink() failed");
            exit(3);
}


if (mq_unlink(consqname) != 0) {
                perror("mq_unlink() failed");
                exit(3);
}

//}

printf("System execution time: %f seconds\n", (t2-t1));

return 0;
}
```

producer.c

```
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <math.h>

int main(int argc, char *argv[])
{

//      int N = atoi(argv[1]);
        int B = atoi(argv[2]);
        int P = atoi(argv[3]);
//      int C = atoi(argv[4]);

        //Process ID, passed from main
        int id = atoi(argv[5]);

        //Main mq
        mqd_t qdes;
```

```
        char  qname[] = "/mailbox1_srajguru";

        mode_t mode = S_IRUSR | S_IWUSR;
        struct mq_attr attr;

        attr.mq_maxmsg  = B;
        attr.mq_msgsize = sizeof(int);
        attr.mq_flags   = 0;            /* a blocking queue  */

        qdes  = mq_open(qname, O_RDWR | O_CREAT, mode, &attr);
        if (qdes == -1 ) {
                perror("mq_open() failed");
                exit(1);
        }

        //Prodcount mq - communicates a counter to make sure only
N numbers are produced
        mqd_t prodmq;

        char  prodqname[] = "/prodmq";

        mode_t modepmq = S_IRUSR | S_IWUSR;
        struct mq_attr pmqattr;

        pmqattr.mq_maxmsg  = 1;
        pmqattr.mq_msgsize = sizeof(int);
        pmqattr.mq_flags   = 0;            /* a blocking queue
*/

        prodmq  = mq_open(prodqname, O_RDWR | O_CREAT, modepmq,
&pmqattr);
        if (prodmq == -1 ) {
                perror("mq_open() failed");
                exit(1);
        }


int prodcount;
int count=0;

     while(1)
{
        mq_receive(prodmq, (char *) &prodcount, sizeof(int), 0);

        //Ensure that only N items are produced
        if(prodcount!=0)
        {
                //Decrement the producer counter and send it back
           prodcount--;
           mq_send(prodmq, (char *)&prodcount, sizeof(int), 0);
```

```
                int number;
                number = id + (count*P);

        //Send number produced to the message queue
        if (mq_send(qdes, (char *)&number, sizeof(int), 0) == -1)
{
                        perror("mq_send() failed");
                }

      count++;
         }

//Last item to be produced
else
{
 mq_send(prodmq, (char *)&prodcount, sizeof(int), 0);
break;
}
}

if (mq_close(qdes) == -1) {
           perror("mq_close() failed");
           exit(2);
      }

if (mq_close(prodmq) == -1) {
           perror("mq_close() failed");
           exit(2);
      }
      return 0;
}
```

consumer.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <math.h>

int is_square(int Num){

        double a = (double) Num;
        double b = sqrt(a);

        if(floor(b) == b)
```

```
        {
                return 1;
        }
        else{
                return 0;
        }

}

int main(int argc, char *argv[])
{
//    int N = atoi(argv[1]);
        int B = atoi(argv[2]);
//      int P = atoi(argv[3]);
//      int C = atoi(argv[4]);
        int id = atoi(argv[5]);

        //Main mq
        mqd_t qdes;

        char  qname[] = "/mailbox1_srajguru";

        mode_t mode = S_IRUSR | S_IWUSR;
        struct mq_attr attr;

        attr.mq_maxmsg  = B;
        attr.mq_msgsize = sizeof(int);
        attr.mq_flags   = 0;

        qdes  = mq_open(qname, O_RDWR | O_CREAT, mode, &attr);
        if (qdes == -1 ) {
                perror("mq_open() failed");
                exit(1);
        }

        //Cons mq - communicates a counter to limit items
consumed
        mqd_t consmq;

        char  consqname[] = "/consmq";

        mode_t modecmq = S_IRUSR | S_IWUSR;
        struct mq_attr cmqattr;

        cmqattr.mq_maxmsg  = 1;
        cmqattr.mq_msgsize = sizeof(int);
        cmqattr.mq_flags   = 0;            /* a blocking queue
*/

        consmq  = mq_open(consqname, O_RDWR | O_CREAT, modecmq,
&cmqattr);
        if (consmq == -1 ) {
                perror("mq_open() failed");
```

```
                exit(1);
        }

        int conscount;

        while(1)
        {
                mq_receive(consmq, (char *) &conscount,
sizeof(int), 0);
                if(conscount!=0)
                {
                //Decrement counter if all of the numbers
produced still haven't been consumed
                 conscount--;
                 mq_send(consmq, (char *)&conscount, sizeof(int),
0);

                 int number;

                 if ((mq_receive(qdes, (char *) &number,
sizeof(int), 0)) == -1)
                 {
                   perror("mq_receive() failed");

           } else {

                //Check if the number received meets the print
conditions as mentioned in the manual
                if(is_square(number) == 1)
                {
                    int res = sqrt(number);
                    printf("%i %i %i\n", id, number, res);
                }

                }
                }

//Last number to be consumed
else
{
 mq_send(consmq, (char *)&conscount, sizeof(int), 0);
 break;
}
}

if (mq_close(qdes) == -1) {
  perror("mq_close() failed");
  exit(2);
}

if (mq_close(consmq) == -1) {
  perror("mq_close() failed");
  exit(2);
```

```
}

return 0;
}
```