ECE 254: Lab 5

Ramie Raufdeen, Sadman Khan

Inter-Thread communication vs Inter-process communication

2015

# Overview

This lab is an extension of lab four in addition to implementing another solution to the producer consumer problem. The first solution is to have threads communicate with each other using shared memory (global queue), in this case a queue. Just like the previous lab, this lab uses the POSIX API’s functionalities; the second solution involved multiple processes utilizing the message queue from POSIX.

# Timing Data

## Inter-Thread Communication using Shared Memory

**System Execution Time** (X=500)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Time |
| 100 | 4 | 1 | 1 | 0.000647 |
| 100 | 4 | 1 | 2 | 0.000746 |
| 100 | 4 | 1 | 3 | 0.000879 |
| 100 | 4 | 2 | 1 | 0.000681 |
| 100 | 4 | 3 | 1 | 0.00079 |
| 100 | 8 | 1 | 1 | 0.000624 |
| 100 | 8 | 1 | 2 | 0.00076 |
| 100 | 8 | 1 | 3 | 0.000834 |
| 100 | 8 | 2 | 1 | 0.000641 |
| 100 | 8 | 3 | 1 | 0.000869 |
| 398 | 8 | 1 | 1 | 0.001493 |
| 398 | 8 | 1 | 2 | 0.001809 |
| 398 | 8 | 1 | 3 | 0.002193 |
| 398 | 8 | 2 | 1 | 0.001672 |
| 398 | 8 | 3 | 1 | 0.002011 |

**Standard Deviation of System Execution Time**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Time |
| 100 | 4 | 1 | 1 | 0.000266 |
| 100 | 4 | 1 | 2 | 0.00025 |
| 100 | 4 | 1 | 3 | 0.000212 |
| 100 | 4 | 2 | 1 | 0.0003 |
| 100 | 4 | 3 | 1 | 0.000297 |
| 100 | 8 | 1 | 1 | 0.000286 |
| 100 | 8 | 1 | 2 | 0.000266 |
| 100 | 8 | 1 | 3 | 0.000252 |
| 100 | 8 | 2 | 1 | 0.000291 |
| 100 | 8 | 3 | 1 | 0.000412 |
| 398 | 8 | 1 | 1 | 0.00041 |
| 398 | 8 | 1 | 2 | 0.00025 |
| 398 | 8 | 1 | 3 | 0.000365 |
| 398 | 8 | 2 | 1 | 0.000411 |
| 398 | 8 | 3 | 1 | 0.000508 |

## Inter-Process Communication using POSIX Message Queue

**System Execution Time** (X=500)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Time |
| 100 | 4 | 1 | 1 | 0.001663 |
| 100 | 4 | 1 | 2 | 0.001886 |
| 100 | 4 | 1 | 3 | 0.002291 |
| 100 | 4 | 2 | 1 | 0.001893 |
| 100 | 4 | 3 | 1 | 0.002173 |
| 100 | 8 | 1 | 1 | 0.001714 |
| 100 | 8 | 1 | 2 | 0.001953 |
| 100 | 8 | 1 | 3 | 0.00227 |
| 100 | 8 | 2 | 1 | 0.001863 |
| 100 | 8 | 3 | 1 | 0.002219 |
| 398 | 8 | 1 | 1 | 0.003084 |
| 398 | 8 | 1 | 2 | 0.003582 |
| 398 | 8 | 1 | 3 | 0.004163 |
| 398 | 8 | 2 | 1 | 0.003198 |
| 398 | 8 | 3 | 1 | 0.003477 |

**Standard Deviation of System Execution Time**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Time |
| 100 | 4 | 1 | 1 | 0.00106 |
| 100 | 4 | 1 | 2 | 0.001185 |
| 100 | 4 | 1 | 3 | 0.001465 |
| 100 | 4 | 2 | 1 | 0.001269 |
| 100 | 4 | 3 | 1 | 0.001436 |
| 100 | 8 | 1 | 1 | 0.001144 |
| 100 | 8 | 1 | 2 | 0.001249 |
| 100 | 8 | 1 | 3 | 0.001492 |
| 100 | 8 | 2 | 1 | 0.001279 |
| 100 | 8 | 3 | 1 | 0.001419 |
| 398 | 8 | 1 | 1 | 0.001724 |
| 398 | 8 | 1 | 2 | 0.00183 |
| 398 | 8 | 1 | 3 | 0.002037 |
| 398 | 8 | 2 | 1 | 0.001605 |
| 398 | 8 | 3 | 1 | 0.001572 |

## Case: (N, B, P, C) = (398, 8, 1, 3)

The chart below consists of the timing data for specific parameters.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Inter-Thread | Inter-Process | Ratio |
| Average Execution Time (s) | 0.00219318 | 0.00613259 | 2.796209 |
| Standard Deviation (s) | 0.000365435 | 0.000632096 | N/A |

From the ratio column above, one can conclude that inter-thread communication using shared memory is approximately 2.786209 times faster than inter-process communication with the POSIX message queue.

## Outperformance Ratio

For all parameter values, inter-thread communication outperforms inter-process communication. The performance ratio column was calculated by dividing the execution time of inter-process communication by the execution time of inter-thread communication with the same parameters.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | B | P | C | Performance Ratio |
| 100 | 4 | 1 | 1 | 2.570324575 |
| 100 | 4 | 1 | 2 | 2.528150134 |
| 100 | 4 | 1 | 3 | 2.606370876 |
| 100 | 4 | 2 | 1 | 2.779735683 |
| 100 | 4 | 3 | 1 | 2.750632911 |
| 100 | 8 | 1 | 1 | 2.746794872 |
| 100 | 8 | 1 | 2 | 2.569736842 |
| 100 | 8 | 1 | 3 | 2.721822542 |
| 100 | 8 | 2 | 1 | 2.906396256 |
| 100 | 8 | 3 | 1 | 2.553509781 |
| 398 | 8 | 1 | 1 | 2.065639652 |
| 398 | 8 | 1 | 2 | 1.980099502 |
| 398 | 8 | 1 | 3 | 1.898312813 |
| 398 | 8 | 2 | 1 | 1.912679426 |
| 398 | 8 | 3 | 1 | 1.728990552 |

Averaging the performance ratio column yields approximately 2.42.

*Please refer to the Excel workbook (lab5\_rpt.xlsx) related to this lab for further information on calculations.*

# Discussion

From analyzing the timing data presented above, the average execution time in seconds for inter-thread communication using shared memory is approximately 2.8 times faster than inter-process communication using the POSIX message queue [for the requested case of (N, B, P, C) = (398, 8, 1, 3)] . Graphs were not presented for various parameter values because the inter-thread communication consistently outperformed the inter-process communication regardless of parameter change. From a theoretical perspective of the problem, it is important to consider that it is much easier for the OS to context-switch between threads than processes; primarily because threads are much more lightweight.

## Further Considerations

An analysis beyond the scope of this report might include changing the value of number of integers produced; by looking at the outperformance ratio table, it seems the outperformance of inter-thread communication decreases as the number of integers produced increases. In addition, the buffer size was unable to exceed 10; the maximum size for the POSIX message queue on the ECE-Linux machines is 10. This could have impacted the results of inter-process communication as well.

## Advantages and Disadvantages

### Why use Threads?

The advantage of utilizing a multi-threaded communication approach as opposed to multi-process communication approach is primarily because the OS can quickly switch between threads compared to processes. Threads being lighter weight compared to processes can help a lot with that!

Threads also take less time to start than processes, processes require a call to *fork()* and *exec()*.

Threads use a lot less resources than processes because a new process has to copy memory which is a performance hit. With the use of a shared memory, the data inside it would be shared between all threads in the executable. Finally, in this case we see that the size of a shared memory queue is much larger than the POSIX message queue on ECE-Linux, which has a size limit of 10.

If performance/speed is an important factor, it would be ideal to use inter-thread communication!

### Why use Processes?

Inter-process communication should be used over inter-thread communication primarily because of the POSIX message queue feature. The queue ensures that there would be minimal concurrency bugs between the processes. Due to the fact that threads run in the same virtual memory and share a common resource (in our case a queue), this could lead to concurrency issues between the shared memory which must be handled. Problems such as race conditions and deadlocks are handled using synchronization techniques which may not be optimal and lead to future bugs. With utilizing the POSIX message queue, the synchronization is already handled and there would be no concern about deadlocks and such.

Another advantage is that processes are run in different executables, whereas threads in this case run in the same executable. As a result, a data fault in one thread may lead to affecting other threads. However, with processes running different executables, a faulty process will not affect other processes.

From a developer perspective, debugging programs that use inter-process communication would be a lot easier than debugging inter-thread communication based programs. There wouldn’t be a need to deal with problems such as synchronization errors and data corruption.

If reliability is key, it would be ideal to use inter-process communication!

## Conclusion

Inter-thread communication should be taken advantage of due to its fast performance; due to its context-switching, and resource usage. Inter-process communication could be leveraged if one is seeking reliability at the cost of performance.

# Source Code Listing

## Consumer.c

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <mqueue.h>
4. #include <sys/stat.h>
5. #include <sys/types.h>
6. #include <sys/wait.h>
7. #include <unistd.h>
8. #include <errno.h>
9. #include <sys/time.h>
10. #include <time.h>
11. #include <string.h>
12. #include <math.h>


16. **int** main(**int** argc, **char** \*argv[]) {
18. /\* processing inputs \*/
19. **int** number\_of\_messages = atoi(argv[1]);
20. **int** cid = atoi(argv[2]);
22. /\* opening the message queue and the consumption queue \*/
23. mqd\_t qdes;
24. **char** \*qname = "/mailbox\_lab4\_extended";
26. mqd\_t consumption\_queue;
27. **char** \*cmqname = "/consumption\_queue\_mailbox";
29. qdes  = mq\_open(qname, O\_RDONLY);
30. **if** (qdes == -1 ) {
31. perror("mq\_open()");
32. exit(1);
33. }
35. consumption\_queue = mq\_open(cmqname, O\_RDWR);
36. **if** (consumption\_queue == -1){
37. perror("mq\_open() failed from consumer");
38. exit(1);
39. }

42. /\* consume elements \*/
44. **int** i;
45. **int** count = 0;
46. **int** sqrt\_val;
48. **while**(1){
49. **int** recieved\_message;
50. **int** consumption\_count;
52. /\* checking the number of messages that have been consumed in total.
53. stopping the process if the consumption count == 0
54. \*/
56. mq\_receive(consumption\_queue, (**char** \*)&consumption\_count, **sizeof**(**int**), 0);
57. **if** (consumption\_count ==  0){
58. mq\_send(consumption\_queue, (**char** \*)&consumption\_count, **sizeof**(**int**), 0);
59. **break**;
60. } **else** {
61. consumption\_count --;
62. mq\_send(consumption\_queue, (**char** \*)&consumption\_count, **sizeof**(**int**), 0);
63. }
65. /\* dequeue the message from the message queue \*/
67. **if** (mq\_receive(qdes, (**char** \*)&recieved\_message, **sizeof**(**int**), 0) == -1){
68. printf("mq\_receive() failed\n");
69. **return** 1;
70. }
72. /\* prints the values if perfect square \*/
73. sqrt\_val = sqrt(recieved\_message);
74. **if**((sqrt\_val \* sqrt\_val) == recieved\_message){
75. printf("%i %i %i\n", cid, recieved\_message, sqrt\_val);
76. }


80. }
82. /\* closing the message queue on the consumer side \*/
84. **if** (mq\_close(qdes) == -1) {
85. perror("mq\_close() failed");
86. exit(2);
87. }
89. /\* closing the consumption queue on the consumer side \*/
91. **if**(mq\_close(consumption\_queue) == -1){
92. perror("mq\_close() failed");
93. exit(2);
94. }
96. **return** 0;
98. }

## Producer.c

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <mqueue.h>
4. #include <sys/stat.h>
5. #include <sys/types.h>
6. #include <sys/wait.h>
7. #include <unistd.h>
8. #include <errno.h>
9. #include <sys/time.h>
10. #include <time.h>
11. #include <string.h>

14. **int** main(**int** argc, **char**\* argv[]){
15. **int** pid;
16. **int** total\_message\_number;
17. **int** buffer\_size;
18. **int** producer\_count;
19. **int** consumer\_count;

22. total\_message\_number = atoi(argv[1]);
23. pid = atoi(argv[2]);
24. producer\_count = atoi(argv[3]);
25. consumer\_count = atoi(argv[4]);
27. /\* Open the message queue from the producer side \*/
29. mqd\_t qdes;
30. **char** \*qname = "/mailbox\_lab4\_extended";
32. qdes = mq\_open(qname,  O\_RDWR);
33. **if**(qdes == -1){
34. perror("mq\_opne()");
35. exit(1);
36. }
38. /\* produce elements \*/
40. **int** i;
41. **for**(i = pid; i < total\_message\_number; i += producer\_count){
42. **if**(mq\_send(qdes, (**char**\*)&i, **sizeof**(**int**), 0) == -1){
43. perror("mq\_send() failed");
44. }
45. }
47. /\* close the message queue from the producer side \*/
49. **if** (mq\_close(qdes) == -1){
50. perror("mq\_close() failed");
51. exit(2);
52. }
54. **return** 0;
56. }

## Producer\_Consumer\_Process.c (Inter-Process method)

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <mqueue.h>
4. #include <sys/stat.h>
5. #include <sys/types.h>
6. #include <sys/wait.h>
7. #include <unistd.h>
8. #include <errno.h>
9. #include <sys/time.h>
10. #include <time.h>
11. #include <string.h>

14. **double** time\_in\_seconds(){
15. **struct** timeval tv;
16. **double** t1;
18. gettimeofday(&tv, NULL);
19. t1 = tv.tv\_sec + tv.tv\_usec/1000000.0;
21. **return** t1;
22. }
24. /\* spawn a child process
25. Both the producers and consumers are each their own process
26. \*/
28. **int** spawn (**char**\* program, **char**\*\* arg\_list, **int** pid){
29. arg\_list[0] = program;

32. /\* need to cast the pid to a string to be passed into the process
33. writing to arg\_list[2] because the buffer count is not needed for the consumers or producers
34. \*/
35. **char** pid\_string[10];
36. sprintf(pid\_string, "%d", pid);
37. arg\_list[2] = pid\_string;

40. pid\_t child\_pid;
41. child\_pid = fork();
43. /\* error check for child process \*/
45. **if** (child\_pid != 0){
46. **return** child\_pid;
47. } **else** {
48. execvp(program, arg\_list);
49. printf("%s\n", "an error has occurred in exec");
50. abort();
51. }
52. }
54. **int** main(**int** argc, **char** \*argv []){
56. /\* process and ensure valid inputs \*/
57. **if** (argc < 5){
58. printf("%s\n", "Invalid number of inputs");
59. exit(0);
60. }
62. **int** consumer\_count;
63. **int** producer\_count;
64. **int** buffer\_size;
65. **int** total\_message\_number;
67. total\_message\_number = atoi(argv[1]);
68. buffer\_size = atoi(argv[2]);
69. producer\_count = atoi(argv[3]);
70. consumer\_count = atoi(argv[4]);
72. **if** (total\_message\_number < 0 || buffer\_size < 0 || producer\_count < 0 || consumer\_count < 0){
73. printf("%s\n", "All inputs must be positive integers" );
74. exit(0);
75. }
77. **double** starting\_time;
78. **double** finished\_init\_time;
79. **double** ending\_time;
80. **double** execution\_time;
82. /\* create and open the message queue \*/
84. **struct** mq\_attr attr;
85. mqd\_t qdes;
86. mode\_t mode = S\_IRUSR | S\_IWUSR;
88. attr.mq\_maxmsg  = buffer\_size;
89. attr.mq\_msgsize = **sizeof**(**int**);
90. attr.mq\_flags   = 0;
92. **char** \*qname = "/mailbox\_lab4\_extended";
94. qdes = mq\_open(qname, O\_RDWR | O\_CREAT, mode, &attr);
95. **if**(qdes == -1){
96. perror("mq\_open() failed");
97. exit(1);
98. }
100. /\* create and open a consumption queue
101. consumption queue is a mq of size 1 which keeps track of the number of messages
102. that have been consumed so far
103. \*/
104. **struct** mq\_attr c\_attr; //consumption queue attr
105. mqd\_t consumption\_queue;
106. mode\_t mode\_c = S\_IRUSR | S\_IWUSR;
108. c\_attr.mq\_maxmsg = 1;
109. c\_attr.mq\_msgsize = **sizeof**(**int**);
110. c\_attr.mq\_flags = 0;
112. **char** \* consumption\_queue\_name = "/consumption\_queue\_mailbox";
114. consumption\_queue = mq\_open(consumption\_queue\_name, O\_RDWR | O\_CREAT, mode\_c, &c\_attr);
115. **if** (consumption\_queue == -1){
116. perror("mq\_open() for consumption failed");
117. exit(1);
118. }

121. /\* we start by writing the value of `total\_message\_number` into the consumption\_queue \*/
122. **if**(mq\_send(consumption\_queue, (**char** \*)&total\_message\_number, **sizeof**(**int**), 0) == -1){
123. perror("mq\_send() for consumption failed");
124. }
126. starting\_time = time\_in\_seconds();
128. /\* loop through and create the producers and consumers processes \*/
130. **int** i;
132. **for**(i = 0; i < consumer\_count; i++){
133. spawn("./consume\_p",argv, i);
134. }
136. **for**(i = 0; i < producer\_count; i++){
137. spawn("./produce\_p",argv, i);
138. }

141. finished\_init\_time = time\_in\_seconds();

144. /\* waiting for all the child processes to finsih before continuing
145. reference: stack overflow
146. \*/
147. **int** pid;
148. **while** (pid = waitpid(-1, NULL, 0)) {
149. **if** (errno == ECHILD) {
150. **break**;
151. }
152. }
154. ending\_time = time\_in\_seconds();
156. /\* assumtion: execution time includes the time for initialization \*/
158. execution\_time = ending\_time - starting\_time;
160. printf("System Excution time: %f seconds\n", execution\_time);

163. /\* closing and unlinking the message queue and consumption queue \*/
165. **if** (mq\_close(qdes) == -1){
166. perror("mq\_close() failed");
167. exit(2);
168. }
170. **if**(mq\_close(consumption\_queue) == -1){
171. perror("mq\_close() for cmq failed from main");
172. exit(2);
173. }
175. **if** (mq\_unlink(qname) != 0) {
176. perror("mq\_unlink() failed");
177. exit(3);
178. }
180. **if**(mq\_unlink(consumption\_queue\_name) == -1){
181. perror("mq\_unlink() failed for cmq from main");
182. exit(3);
183. }
185. **return** 0;
186. }

## Producer\_Consumer\_Thread.c (Inter-Thread method)

1. /\*
2. ECE 254- lab 5
3. Note, The structure of this code is based on
4. advancedlinuxprogramming.com/alp-folder/alp-ch04-threads.pdf
5. listing 4.12
6. (Recomended reading from the pre lab)
8. Sadman Khan
9. Ramie Raufdeen
10. \*/


14. #include <stdio.h>
15. #include <stdlib.h>
16. #include <pthread.h>
17. #include <semaphore.h>
18. #include <sys/time.h>
19. #include <stdlib.h>
20. #include <math.h>
22. /\* a queue struct implemented with a circular array
23. it gets allocated as global variable and becomes shared memory
24. that can be accessed by all threads
25. \*/

28. **struct** queue {
29. **int** \*array;
30. **int** head;
31. **int** tail;
32. };
34. **struct** queue message\_queue;
35. pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;
37. sem\_t message\_count;           //makes sure that there is a message in the queue before the consumer tries to consume
38. sem\_t consumed\_messages\_count; //allow the consumer thread to exit once all the messages have been consumed
39. sem\_t empty\_space;             //makes sure that there is enough space in the queue to handle another enqueue
41. **int** producer\_count;
42. **int** total\_number\_of\_messages;
43. **int** message\_queue\_size;
45. /\* a one time init for the message queue and semaphores \*/
47. **void** init\_message\_queue(){
48. message\_queue.array = (**int**\*) malloc (**sizeof**(**int**) \* message\_queue\_size);
49. message\_queue.tail = 0;
50. message\_queue.head = 0;
52. sem\_init(&message\_count, 0,0);
53. sem\_init(&consumed\_messages\_count,0,total\_number\_of\_messages);
54. sem\_init(&empty\_space, 0, message\_queue\_size);
55. }
57. **double** time\_in\_seconds(){
58. **struct** timeval tv;
59. **double** t1;
61. gettimeofday(&tv, NULL);
62. t1 = tv.tv\_sec + tv.tv\_usec/1000000.0;
64. **return** t1;
65. }
67. /\* consumer thread function
68. keep consuming from the message queue until the `consumed\_messsages\_count` hits 0
69. \*/
71. **void**\* dequeue\_message(**void**\* arg){
72. **int** c\_id = (**int**) arg;
73. **while**(1){
75. **if**(sem\_trywait(&consumed\_messages\_count)){
76. **break**;
77. }
79. //wait for a message to be in the queue
80. sem\_wait(&message\_count);
82. // lock the message queue
83. pthread\_mutex\_lock (&lock);
84. **int** value, sqrt\_val;
86. message\_queue.head++;
87. **if** (message\_queue.head == message\_queue\_size){
88. message\_queue.head = 0;
89. }
91. /\* printing the value if it is a perfect square \*/
93. value = message\_queue.array[message\_queue.head];
94. sqrt\_val = sqrt(value);
96. **if** (value == (sqrt\_val \* sqrt\_val)){
97. printf("%d %d %d\n", c\_id, value, sqrt\_val);
98. }
100. pthread\_mutex\_unlock(&lock);
102. //signal producers that there is one less message in the message queue
103. sem\_post(&empty\_space);

106. }
107. **return** NULL;
108. }
110. /\* producer thread function
111. produces using the given algorithm from the manual
113. each producer produces a set number of ints
114. \*/
116. **void**\* enqueue\_message (**void**\* arg){
117. **int** p\_id = (**int**) arg;
118. **int** i;
119. **for**(i = p\_id; i < total\_number\_of\_messages; i += producer\_count){
121. // make sure that the number of ints in the message queue does not exceed the buffer size
122. sem\_wait(&empty\_space);
124. pthread\_mutex\_lock(&lock);
126. message\_queue.tail++;
127. **if** (message\_queue.tail == message\_queue\_size){
128. message\_queue.tail = 0;
129. }
131. message\_queue.array[message\_queue.tail] = i;
132. pthread\_mutex\_unlock(&lock);
134. // let consumers know that there is a message in the queue
135. sem\_post(&message\_count);
137. }
138. **return** NULL;
140. }

143. **int** main(**int** argc, **char** \*argv[0]){
144. **if** (argc < 5){
145. printf("Invalid number of inputs\n");
146. exit(0);
147. }
149. **int** consumer\_count;
151. total\_number\_of\_messages = atoi(argv[1]);
152. message\_queue\_size = atoi(argv[2]);
153. producer\_count = atoi(argv[3]);
154. consumer\_count = atoi(argv[4]);
156. **if** (total\_number\_of\_messages < 0 || message\_queue\_size < 0 || producer\_count < 0 || consumer\_count < 0){
157. printf("all args must be positive integers");
158. exit(1);
159. }
161. init\_message\_queue();
163. **double** starting\_time;
164. **double** ending\_time;
165. **double** total\_time;
167. /\* create an array of producer and consumer thread ids \*/
169. pthread\_t producer\_thread\_id[producer\_count];
170. pthread\_t consumer\_thread\_id[consumer\_count];
172. starting\_time = time\_in\_seconds();
174. /\* create the producer and consumer threads \*/
176. **int** i;
177. **for**(i = 0; i < producer\_count; i++){
178. pthread\_create(&(producer\_thread\_id[i]), NULL, &enqueue\_message, (**void**\*)(i));
179. }
181. **for**(i=0; i < consumer\_count; i++){
182. pthread\_create(&(consumer\_thread\_id[i]), NULL, &dequeue\_message, (**void**\*)(i));
183. }

186. /\* joing the producer and consumer threads to avoid exiting the main thread
187. before the other threads have been completed
188. \*/
190. **for**(i=0; i < producer\_count; i++){
191. pthread\_join(producer\_thread\_id[i], NULL);
192. }
194. **for**(i=0; i < consumer\_count; i++){
195. pthread\_join(consumer\_thread\_id[i], NULL);
196. }
198. ending\_time = time\_in\_seconds();
199. total\_time = ending\_time - starting\_time;
201. printf("System execution time: %f seconds\n", total\_time);
203. /\* destroying the semaphores \*/
205. sem\_destroy(&message\_count);
206. sem\_destroy(&consumed\_messages\_count);
207. sem\_destroy(&empty\_space);
209. **return** 0;
210. }