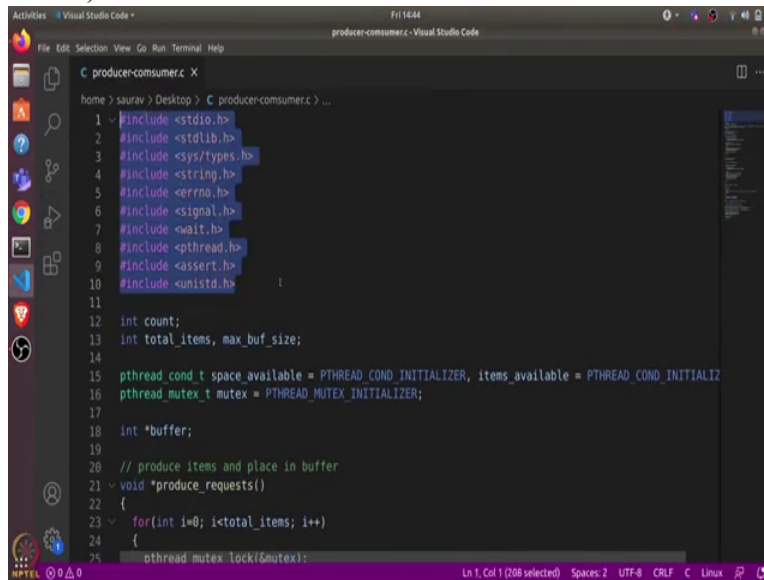
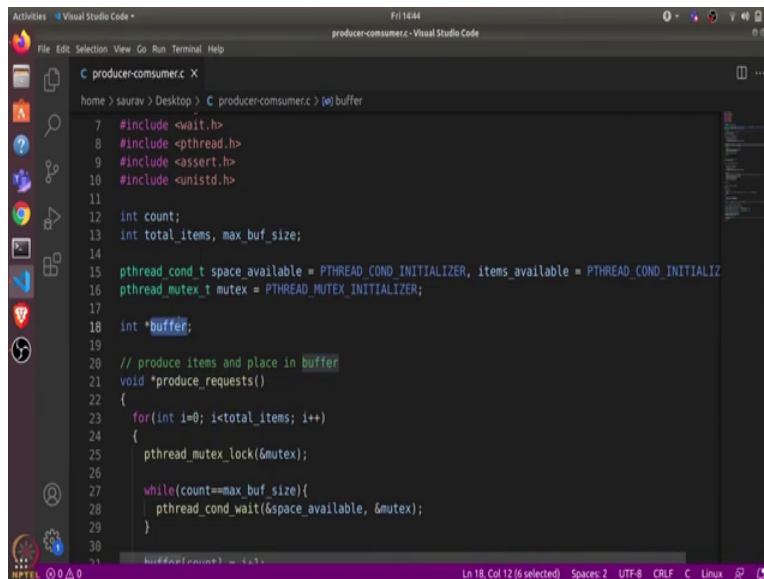


Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 43 (Week 6, Tutorial 1)
Condition variables in C

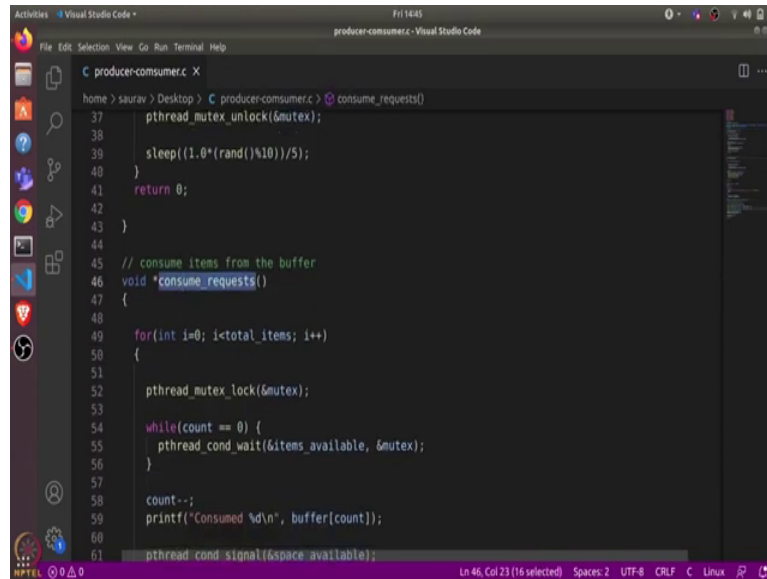
(Refer Slide Time: 00:20)



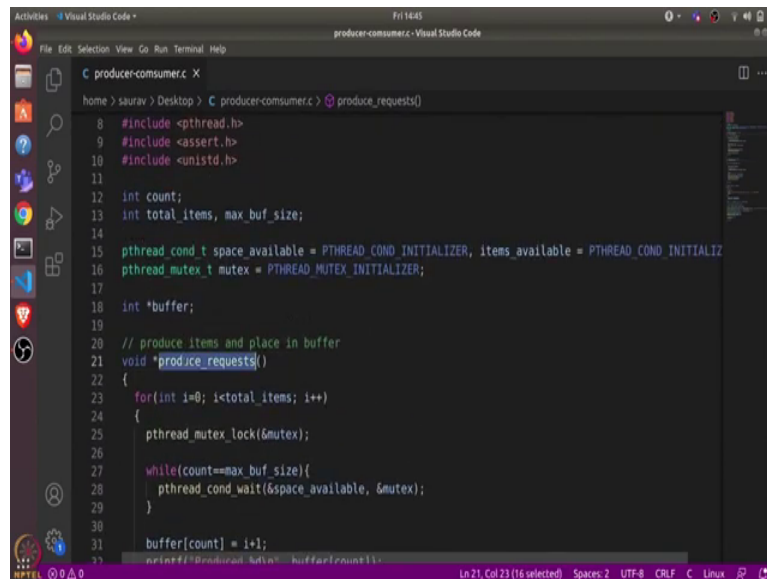
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <string.h>
5 #include <errno.h>
6 #include <signal.h>
7 #include <wait.h>
8 #include <pthread.h>
9 #include <assert.h>
10 #include <unistd.h>
11
12 int count;
13 int total_items, max_buf_size;
14
15 pthread_cond_t space_available = PTHREAD_COND_INITIALIZER, items_available = PTHREAD_COND_INITIALIZER;
16 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
17
18 int *buffer;
19
20 // produce items and place in buffer
21 void *produce_requests()
22 {
23     for(int i=0; i<total_items; i++)
24     {
25         pthread_mutex_lock(&mutex);
```



```
7 #include <wait.h>
8 #include <pthread.h>
9 #include <assert.h>
10 #include <unistd.h>
11
12 int count;
13 int total_items, max_buf_size;
14
15 pthread_cond_t space_available = PTHREAD_COND_INITIALIZER, items_available = PTHREAD_COND_INITIALIZER;
16 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
17
18 int *buffer;
19
20 // produce items and place in buffer
21 void *produce_requests()
22 {
23     for(int i=0; i<total_items; i++)
24     {
25         pthread_mutex_lock(&mutex);
26
27         while(count==max_buf_size){
28             pthread_cond_wait(&space_available, &mutex);
29         }
30
31         // Add item to buffer
```



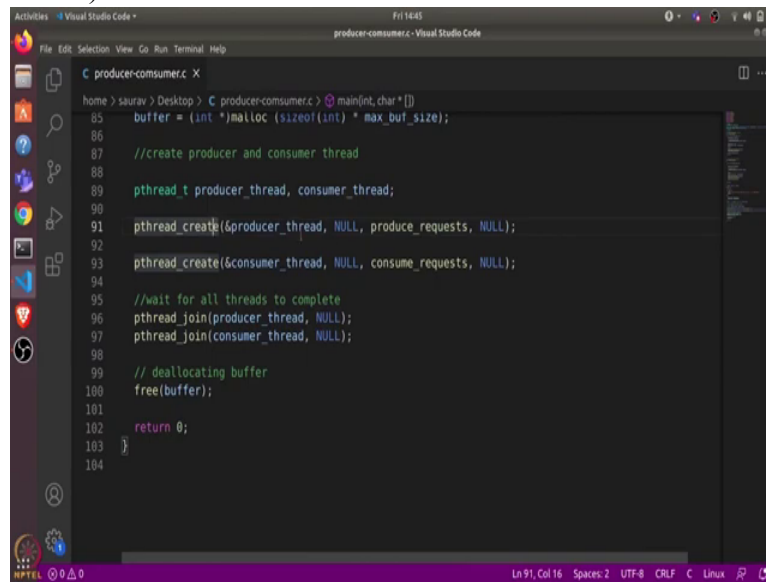
```
home > saurav > Desktop > C producer-consumer.c > consume_request()
37 pthread_mutex_unlock(&mutex);
38
39 sleep((1.0*(rand()%10))/5);
40 }
41 return 0;
42 }
43 }
44
45 // consume items from the buffer
46 void *consume_request()
47 {
48
49 for(int i=0; i<total_items; i++)
50 {
51
52 pthread_mutex_lock(&mutex);
53
54 while(count == 0) {
55 pthread_cond_wait(&items_available, &mutex);
56 }
57
58 count--;
59 printf("Consumed %d\n", buffer[count]);
60
61 pthread_cond_signal(&space_available);
```



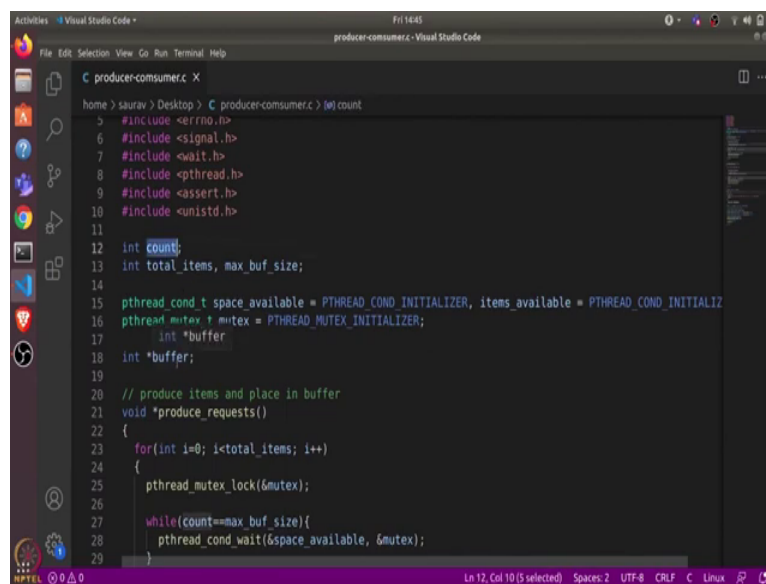
```
home > saurav > Desktop > C producer-consumer.c > produce_request()
8 #include <pthread.h>
9 #include <assert.h>
10 #include <unistd.h>
11
12 int count;
13 int total_items, max_buf_size;
14
15 pthread_cond_t space_available = PTHREAD_COND_INITIALIZER, items_available = PTHREAD_COND_INITIALIZER;
16 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
17
18 int *buffer;
19
20 // produce items and place in buffer
21 void *produce_request()
22 {
23 for(int i=0; i<total_items; i++)
24 {
25 pthread_mutex_lock(&mutex);
26
27 while(count==max_buf_size){
28 pthread_cond_wait(&space_available, &mutex);
29 }
30
31 buffer[count] = i+1;
32 printf("Produced %d\n", buffer[count]);
```

Hi everyone, today we will learn about condition variables. So, I have written the code for the producer consumer problem that you have seen in the lectures. So, let us open this code in this visual code. So, here I have just included some libraries. So, let us first see what all is there in this program. We have a buffer, which will store pending requests that are to be consumed by the consumer. So, we have written this consume_request function that will take out requests from the buffer and consume them one by one. Also, this produce_request function this will add request to the buffer when the buffer is empty.

(Refer Slide Time: 00:55)



```
home > saurav > Desktop > C producer-consumer.c > main(int, char* [])
85  buffer = (int *)malloc (sizeof(int) * max_buf_size);
86
87  //create producer and consumer thread
88
89  pthread_t producer_thread, consumer_thread;
90
91  pthread_create(&producer_thread, NULL, produce_requests, NULL);
92
93  pthread_create(&consumer_thread, NULL, consume_requests, NULL);
94
95  //wait for all threads to complete
96  pthread_join(producer_thread, NULL);
97  pthread_join(consumer_thread, NULL);
98
99  // deallocating buffer
100 free(buffer);
101
102  return 0;
103
104
```



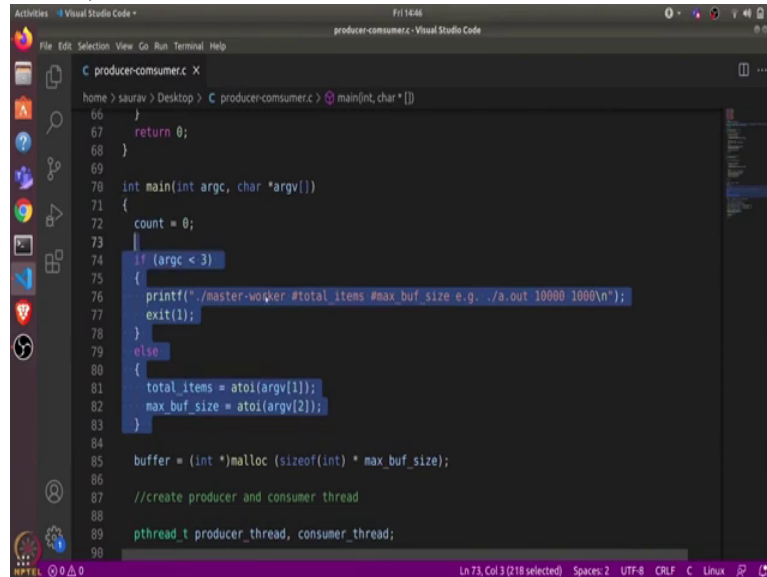
```
home > saurav > Desktop > C producer-consumer.c > | wc -l
1
2  #include <errno.h>
3  #include <signal.h>
4  #include <wait.h>
5  #include <pthread.h>
6  #include <assert.h>
7  #include <unistd.h>
8
9  int count;
10 int total_items, max_buf_size;
11
12 pthread_cond_t space_available = PTHREAD_COND_INITIALIZER, items_available = PTHREAD_COND_INITIALIZER;
13 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
14 int *buffer;
15
16 // produce items and place in buffer
17 void *produce_requests()
18 {
19     for(int i=0; i<total_items; i++)
20     {
21         pthread_mutex_lock(&mutex);
22
23         while(count==max_buf_size){
24             pthread_cond_wait(&space_available, &mutex);
25         }
26     }
27 }
```

So, we have two threads. First thread is the producer thread, which runs this produce_request function. And another is consumer thread, which runs this consume_request function. So, let us see what all variables are there. So, firstly the count variable. This variable stores the number of pending requests that are there in the buffer currently, this denotes the total number of requests that will be added by producer requests function to the buffer.

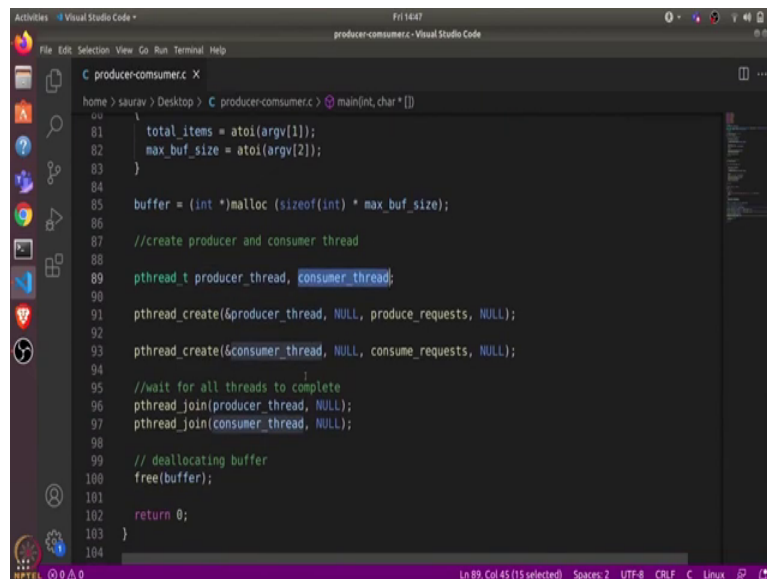
And then we have maximum buffer size. So, the buffer is bounded, which means that we cannot have more than max buff size number of requests in the buffer. So, if let us say max buff size is five, that means that our buffer can only store five pending requests at a time. And if

produce_request want to add more requests to the buffer, it needs to wait for the consumer to consume some requests, and only then it can add more requests.

(Refer Slide Time: 01:45)



```
66 }
67 return 0;
68 }
69
70 int main(int argc, char *argv[])
71 {
72     count = 0;
73
74     if (argc < 3)
75     {
76         printf("master-worker #total_items #max_buf_size e.g. ./a.out 10000 1000\n");
77         exit(1);
78     }
79     else
80     {
81         total_items = atoi(argv[1]);
82         max_buf_size = atoi(argv[2]);
83     }
84
85     buffer = (int *)malloc(sizeof(int) * max_buf_size);
86
87     //create producer and consumer thread
88     pthread_t producer_thread, consumer_thread;
89
90
```



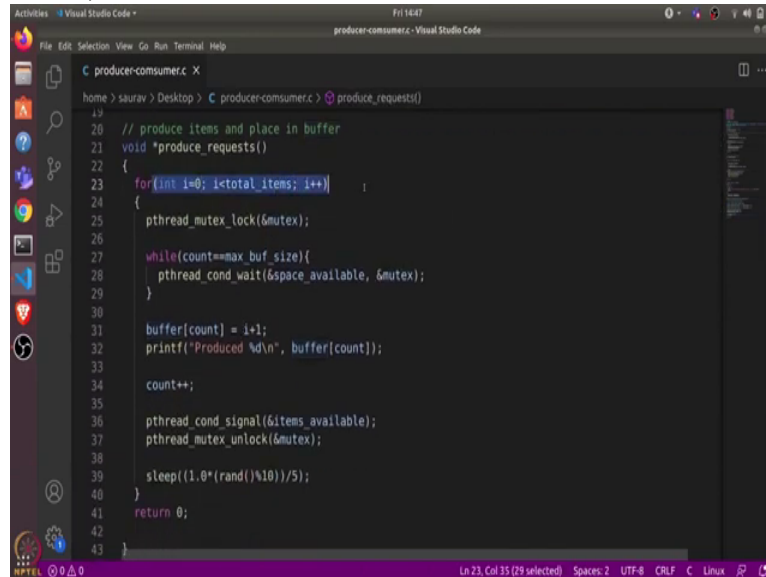
```
81     total_items = atoi(argv[1]);
82     max_buf_size = atoi(argv[2]);
83 }
84
85 buffer = (int *)malloc(sizeof(int) * max_buf_size);
86
87 //create producer and consumer thread
88
89 pthread_t producer_thread, consumer_thread;
90
91 pthread_create(&producer_thread, NULL, produce_requests, NULL);
92
93 pthread_create(&consumer_thread, NULL, consume_requests, NULL);
94
95 //wait for all threads to complete
96 pthread_join(producer_thread, NULL);
97 pthread_join(consumer_thread, NULL);
98
99 // deallocating buffer
100 free(buffer);
101
102 return 0;
103 }
104
```

So, let us go through the code, I will first go through the main function. So, we initialize count with 0. And here we just take two arguments, number one is the total number of items that we want to produce and second is the maximum buffer size. Then we use the malloc function to allocate an integer array of size max buff size.

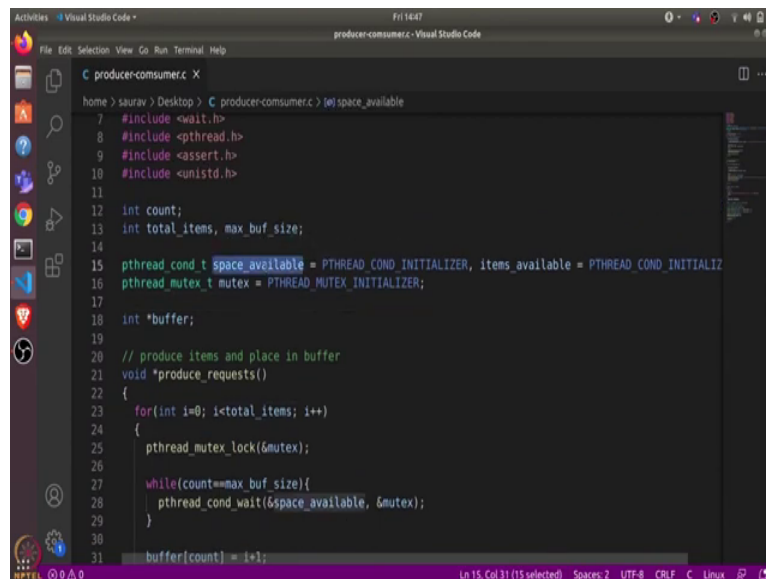
Then we have two threads, producer and consumer thread, as discussed earlier, and we call the pthread join function on both the threads so that the main thread waits for both the threads to

finish. And finally, we free the buffer that was allocated and return 0. So, let us have a look at produced equation consume_request function.

(Refer Slide Time: 02:28)



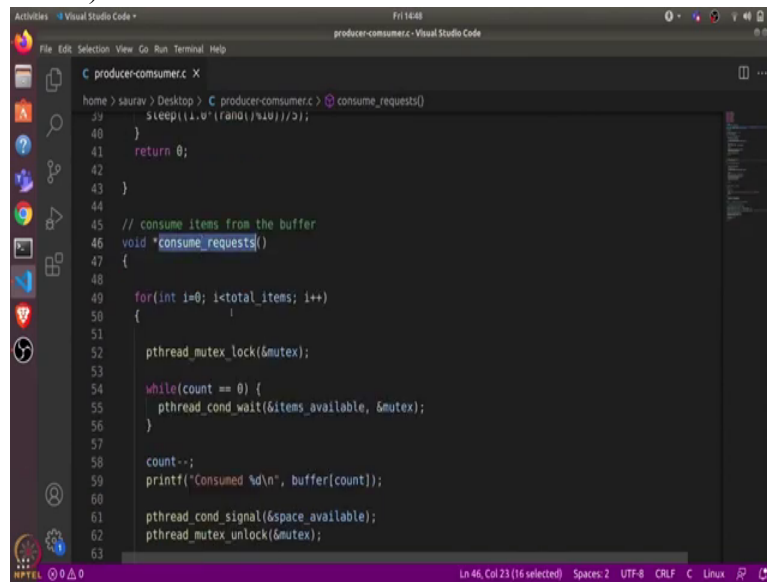
```
17
18 // produce items and place in buffer
19 void *produce_request()
20 {
21     for(int i=0; i<total_items; i++)
22     {
23         pthread_mutex_lock(&mutex);
24
25         while(count==max_buf_size){
26             pthread_cond_wait(&space_available, &mutex);
27         }
28
29         buffer[count] = i+1;
30         printf("Produced %d\n", buffer[count]);
31
32         count++;
33
34         pthread_cond_signal(&items_available);
35         pthread_mutex_unlock(&mutex);
36
37         sleep((1.0*(rand()%10))/5);
38     }
39     return 0;
40 }
```



```
7 #include <wait.h>
8 #include <pthread.h>
9 #include <assert.h>
10 #include <unistd.h>
11
12 int count;
13 int total_items, max_buf_size;
14
15 pthread_cond_t space_available = PTHREAD_COND_INITIALIZER, items_available = PTHREAD_COND_INITIALIZER;
16 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
17
18 int *buffer;
19
20 // produce items and place in buffer
21 void *produce_request()
22 {
23     for(int i=0; i<total_items; i++)
24     {
25         pthread_mutex_lock(&mutex);
26
27         while(count==max_buf_size){
28             pthread_cond_wait(&space_available, &mutex);
29         }
30
31         buffer[count] = i+1;
```

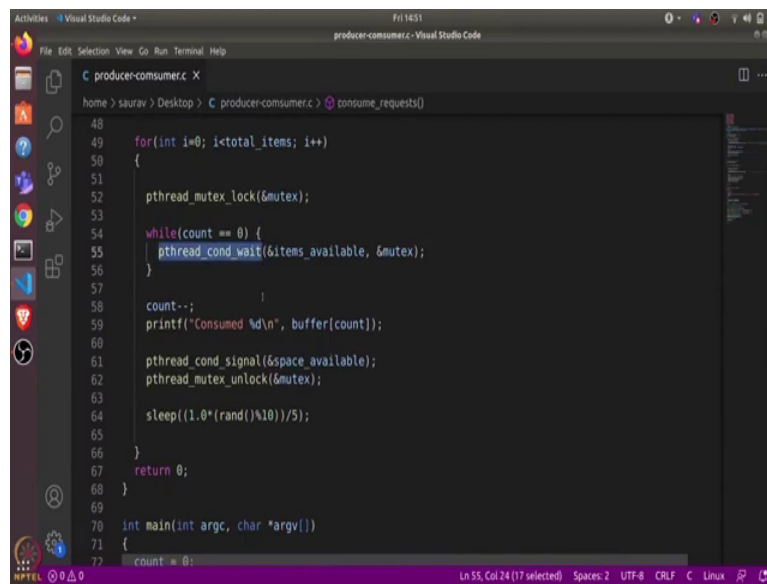
So, this is the produce_request function, here we iterate total item number of times and in every iteration, we will produce one item and put it in the buffer queue. Before putting in the buffer queue, we need to check if the buffer is already full. And if it is, so, then we need to wait for the consumer we have two condition variables here. One is space available and another is items available.

(Refer Slide Time: 02:51)



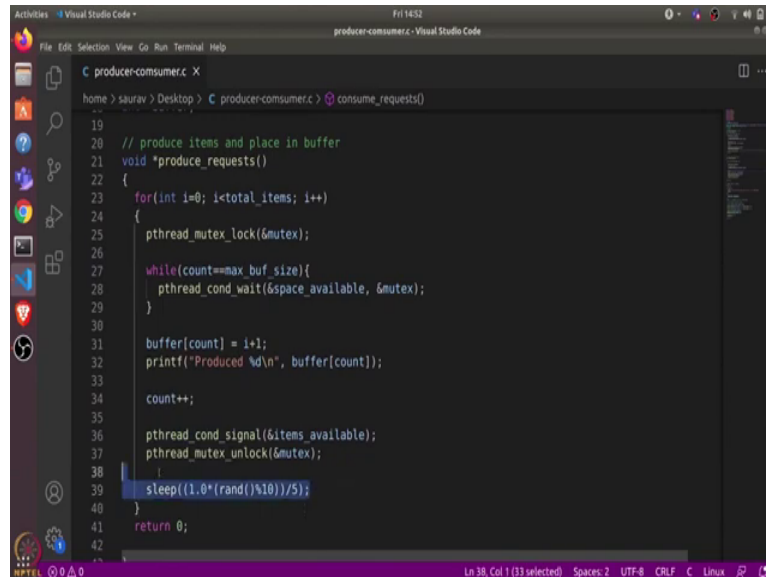
This screenshot shows the Visual Studio Code editor with the file 'producer-consumer.c' open. The 'consume_requests()' function is selected, spanning from line 39 to line 63. The code implements a consumer thread that sleeps for a random duration, then enters a loop to consume items from a buffer. It uses a mutex to ensure mutual exclusion and a condition variable to wait for items to be available. The buffer is represented as an array, and the count of consumed items is tracked.

```
39     sleep((1.0*(rand())%5));
40 }
41 return 0;
42 }
43 }
44
45 // consume items from the buffer
46 void *consume_requests()
47 {
48     for(int i=0; i<total_items; i++)
49     {
50         pthread_mutex_lock(&mutex);
51
52         while(count == 0) {
53             pthread_cond_wait(&items_available, &mutex);
54         }
55
56         count--;
57         printf("Consumed %d\n", buffer[count]);
58
59         pthread_cond_signal(&space_available);
60         pthread_mutex_unlock(&mutex);
61     }
62 }
```



This screenshot shows the Visual Studio Code editor with the file 'producer-consumer.c' open. The 'main()' function is selected, spanning from line 48 to line 72. The code sets up the initial state of the program, including the total number of items, the buffer, and the mutex and condition variables. It then calls the 'consume_requests()' function to start the consumer thread.

```
48     for(int i=0; i<total_items; i++)
49     {
50         pthread_mutex_lock(&mutex);
51
52         while(count == 0) {
53             pthread_cond_wait(&items_available, &mutex);
54         }
55
56         count--;
57         printf("Consumed %d\n", buffer[count]);
58
59         pthread_cond_signal(&space_available);
60         pthread_mutex_unlock(&mutex);
61     }
62     sleep((1.0*(rand())%5));
63     return 0;
64 }
65
66 int main(int argc, char *argv[])
67 {
68     count = 0;
69 }
```



```
19
20 // produce items and place in buffer
21 void *produce_requests()
22 {
23     for(int i=0; i<total_items; i++)
24     {
25         pthread_mutex_lock(&mutex);
26
27         while(count==max_buf_size){
28             pthread_cond_wait(&space_available, &mutex);
29         }
30
31         buffer[count] = i+1;
32         printf("Produced %d\n", buffer[count]);
33
34         count++;
35
36         pthread_cond_signal(&items_available);
37         pthread_mutex_unlock(&mutex);
38         sleep((1.0*(rand())%10)/5);
39     }
40     return 0;
41 }
42
```

So, let us say consumer requests function is executed first, then what will happen here, it will first lock the mutex as discussed in the lectures, it is necessary to hold a lock whenever you are checking for condition, because we do not want the computer to context switch after checking the condition otherwise that there will be a missed signal. So, it will acquire the lock then it will check if count is 0.

Now count is equal to 0 means that there is no item in the buffer. So, consume_request cannot consume request, because there are none in the buffer. So, it needs to wait for the items available condition variable and it will also pass the mutex. So, this function will release the mutex and this will cause this thread to sleep. Now, the produced request function will have to execute and here it will also acquire the mutex and it will check if count is equal maximum buffer size which is not the case because initially count is 0, it will just add one element in the buffer array.

It will set `buffer_count = i + 1` and then it will pin that it has produced this particular request and then when you increase the count by 1. So, now we have one request produced in the buffer queue. So, now we call pthread condition signal. So, we signal this condition variable that items are available now and we unlock the mutex. So, what will happen here? This thread was sleeping for the items available.

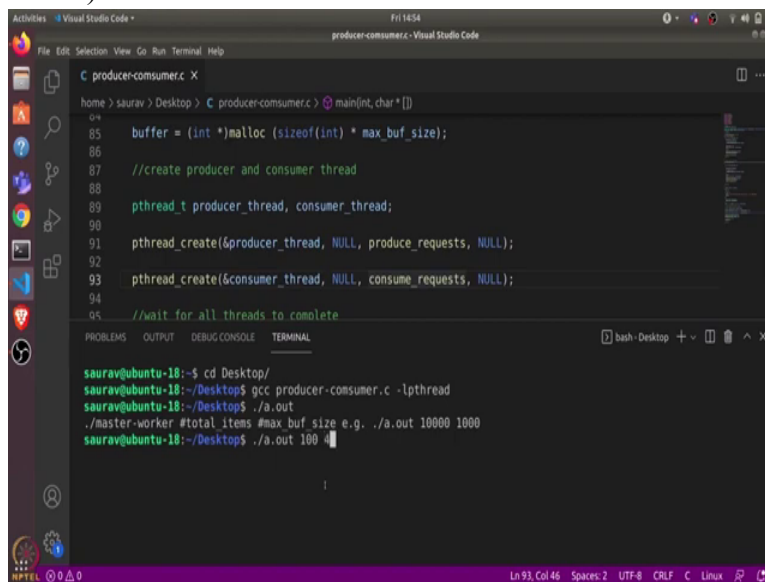
So, once we it gets a signal that items are available, this function will return it will acquire the lock again and return and here this will decrease the count by one and it will consume the item and print that consumed this item. Now, after consuming the item, it will signal that space is

available and it will unlock the mutex. So, why do we need this space available condition variable. So, here in `produce_request`, let us say it is producing request, but consumer is not consuming it.

So, Buffer will get fooled at some point of time. So, it need should check this condition that while count is equal to maximum buffer size, which means that it cannot produce more request. It needs to wait for the space available condition variable again we pass the same mutex to this function as well. So, these two condition variables space available and items available result in coordination of these two functions.

So, `produce_request` will keep producing request and if buffer is full then it will wait for space available and `consume_request` will signal space available after consuming an item. And in both this function we add a random sleep time so that the next request is produced and consumed after some time. So, that is the complete code and let us compile and run this code.

(Refer Slide Time: 05:36)



```
Visual Studio Code
producer-consumer.c - Visual Studio Code
File Edit Selection View Go Run Terminal Help

C producer-consumer.c X
home > saurav > Desktop > C producer-consumer.c > main(int, char*[])
65
66
67 //create producer and consumer thread
68
69 pthread_t producer_thread, consumer_thread;
70
71 pthread_create(&producer_thread, NULL, produce_requests, NULL);
72
73 pthread_create(&consumer_thread, NULL, consume_requests, NULL);
74
75 //wait for all threads to complete

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
bash - Desktop
saurav@ubuntu-18:~$ cd Desktop/
saurav@ubuntu-18:~/Desktop$ gcc producer-consumer.c -lpthread
saurav@ubuntu-18:~/Desktop$ ./a.out
./master-worker #total items #max_buf size e.g. ./a.out 10000 1000
saurav@ubuntu-18:~/Desktop$ ./a.out 100 100
```



```
Activities Visual Studio Code
producer-consumer.c - Visual Studio Code

C producer-consumer.c X
home > saurav > Desktop > C producer-consumer.c > main(int, char* [])
q++
85 buffer = (int *)malloc(sizeof(int) * max_buf_size);
86
87 //create producer and consumer thread
88
89 pthread_t producer_thread, consumer_thread;
90
91 pthread_create(&producer_thread, NULL, produce_requests, NULL);
92
93 pthread_create(&consumer_thread, NULL, consume_requests, NULL);
94
95 //wait for all threads to complete

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
/a.out - Desktop + - - - - - ^ X

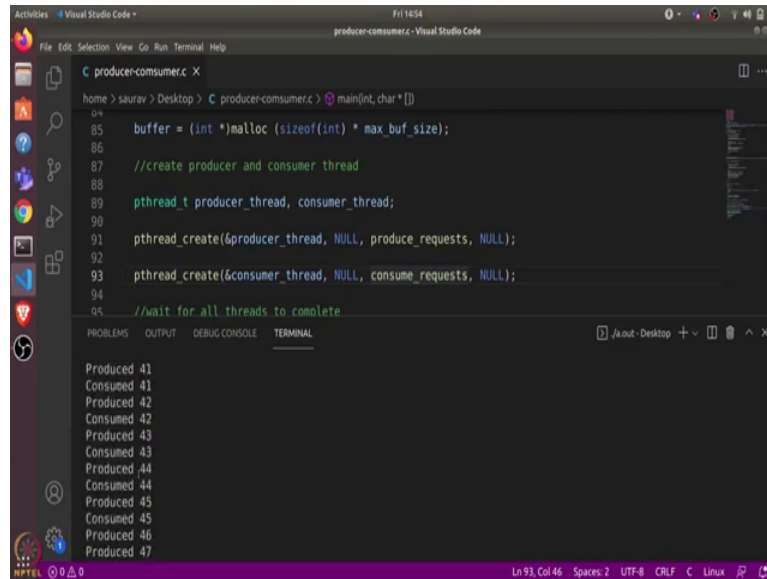
saurav@ubuntu-18:~/Desktop$ ./a.out
./master-worker #total items #max buf size e.g. ./a.out 10000 1000
saurav@ubuntu-18:~/Desktop$ ./a.out 100 4
Produced 1
Produced 2
Consumed 2
Produced 3
Consumed 3
Consumed 1
Produced 4
Consumed 4
```

```
Activities Visual Studio Code
producer-consumer.c - Visual Studio Code

C producer-consumer.c X
home > saurav > Desktop > C producer-consumer.c > main(int, char* [])
q++
85 buffer = (int *)malloc(sizeof(int) * max_buf_size);
86
87 //create producer and consumer thread
88
89 pthread_t producer_thread, consumer_thread;
90
91 pthread_create(&producer_thread, NULL, produce_requests, NULL);
92
93 pthread_create(&consumer_thread, NULL, consume_requests, NULL);
94
95 //wait for all threads to complete

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
/a.out - Desktop + - - - - - ^ X

Produced 2
Consumed 2
Produced 3
Consumed 3
Consumed 1
Produced 4
Consumed 4
Produced 5
Consumed 5
Produced 6
Consumed 6
Produced 7
```



```
home > saurav > Desktop > C producer-consumer.c > main(int, char* [])
85  buffer = (int *)malloc (sizeof(int) * max_buf_size);
86
87  //create producer and consumer thread
88
89  pthread_t producer_thread, consumer_thread;
90
91  pthread_create(&producer_thread, NULL, produce_requests, NULL);
92
93  pthread_create(&consumer_thread, NULL, consume_requests, NULL);
94
95  //wait for all threads to complete
```

Produced 41
Consumed 41
Produced 42
Consumed 42
Produced 43
Consumed 43
Produced 44
Consumed 44
Produced 45
Consumed 45
Produced 46
Produced 47

So, I will first do `cd desktop` and now `gcc producer-consumer.c` with library `pthread`. So, this will produce the a dot out executable let us run that executable. And if we directly run it will say that you need to mention total items and maximum buffer size. So, let us write a dot out total items let us say 100 and maximum buffer size maybe 4. So, here you see that every item will be produced first and only then it will be consumed.

So, here we see it produced item number 1 than it produced item number 2, then we consume item number 2 and there is no consume for some item before it is produced. So, that is a coordination that we get using the condition variables. So, it will keep producing items and adding it in the buffer and consumer will consume them one by one. So, that was it for this video. Thanks, and have a nice day.