DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING

[INQ0091623] AA 2022-23

Lab 9

Synchronization examples

Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

Synchronization

Synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a ***critical section***.

When one thread starts executing the critical section (a serialized segment of the program) the other thread should wait until the first thread finishes.

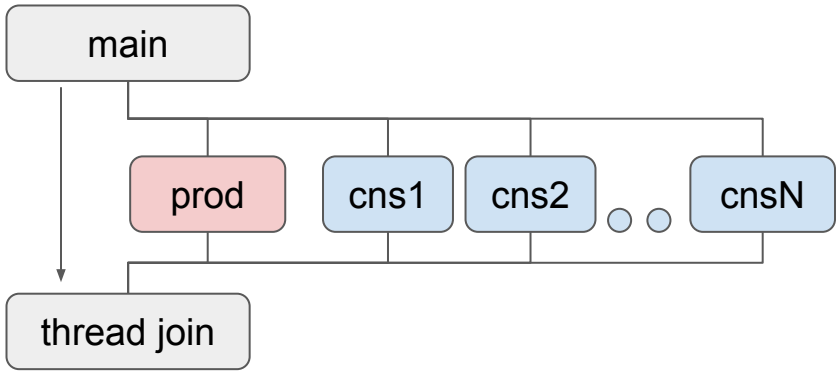
If proper synchronization techniques are not applied, it may cause a ***race condition*** where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

It all depends on shared resources !

but the outcome (race condition) can lead to two issues:

- Loss of data consistency (missing of information)
- DeadLock (missing of synchronization)

REPLAY: Producer and Consumer example



prodcons without synchronization [ERROR]

CODING EXAMPLE

CODING EXAMPLE

PRODUCER

```
static void *producer(void *arg)
{
   int item = 0, i;
   for(i = 0; i < HISTORY_LEN; i++)
   {
       clock_gettime(CLOCK_REALTIME, &sendTimes[item]);
       buffer[writeIdx] = item;
       writeIdx = (writeIdx + 1)% BUFFER_SIZE;
       item++;
   }
   printf("fin.\n");
   return NULL;
}
```

CONSUMER

```
static void *consumer(void *arg)
{
   int item, i;
   for(i = 0; i < HISTORY_LEN; i++)
   {
       item = buffer[readIdx];
       readIdx = (readIdx + 1)% BUFFER_SIZE;

       // simulating a complex operation with 1us
       // average computation time
       wait_us(1);

       printf("%d\n",item);
       clock_gettime(CLOCK_REALTIME, &receiveTimes[item]);
    }
  return NULL;
}
```

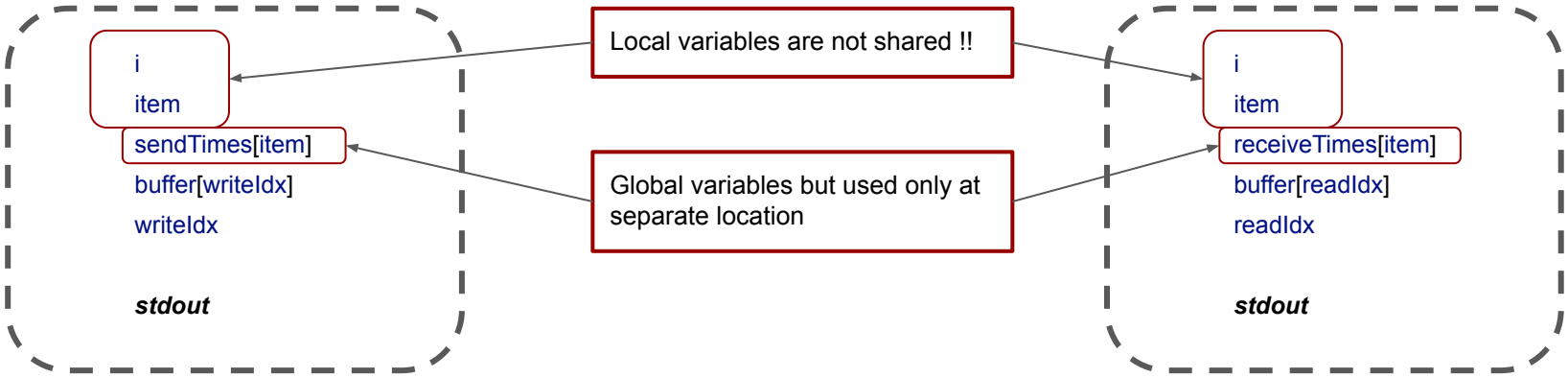
Synchronization (procedure idea)

1. identify the shared resources:

For each thread/process code list any data that is accessed for **write**

```
static void *producer(void *arg)
{
  int item = 0, i;
  for(i = 0; i < HISTORY_LEN; i++)
  {
    clock_gettime(CLOCK_REALTIME, &sendTimes[item]);
    buffer[writeIdx] = item;
    writeIdx = (writeIdx + 1)% BUFFER_SIZE;
    item++;
  }
  printf("fin.\n");
  return NULL;
}
```

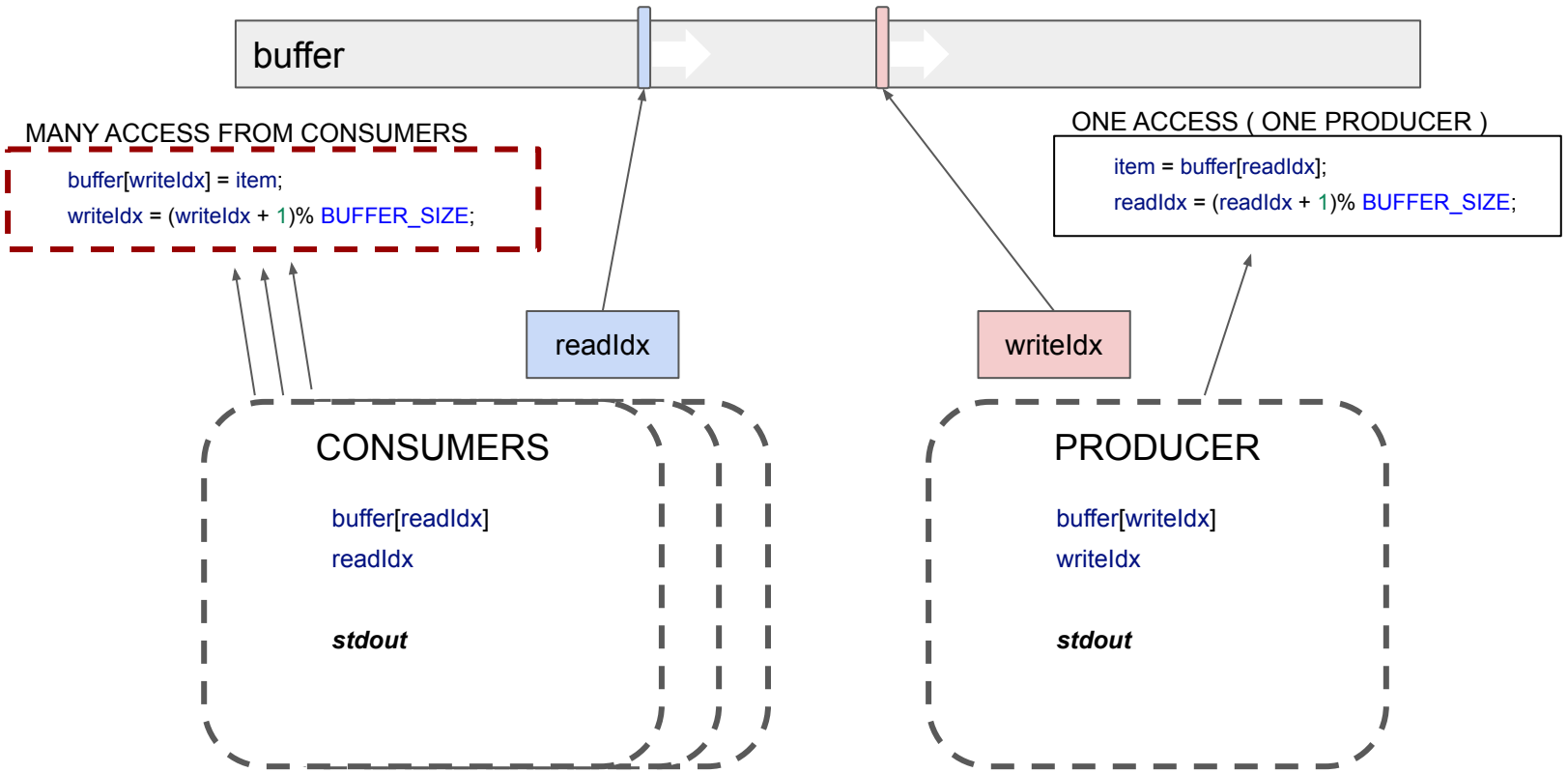
```
static void *consumer(void *arg)
{
  int item, i;
  for(i = 0; i < HISTORY_LEN; i++)
  {
    item = buffer[readIdx];
    readIdx = (readIdx + 1)% BUFFER_SIZE;

    printf("%d\n",item);
    clock_gettime(CLOCK_REALTIME, &receiveTimes[item]);
  }
  return NULL;
}
```



Synchronization (procedure idea)

1. identify the shared resources:

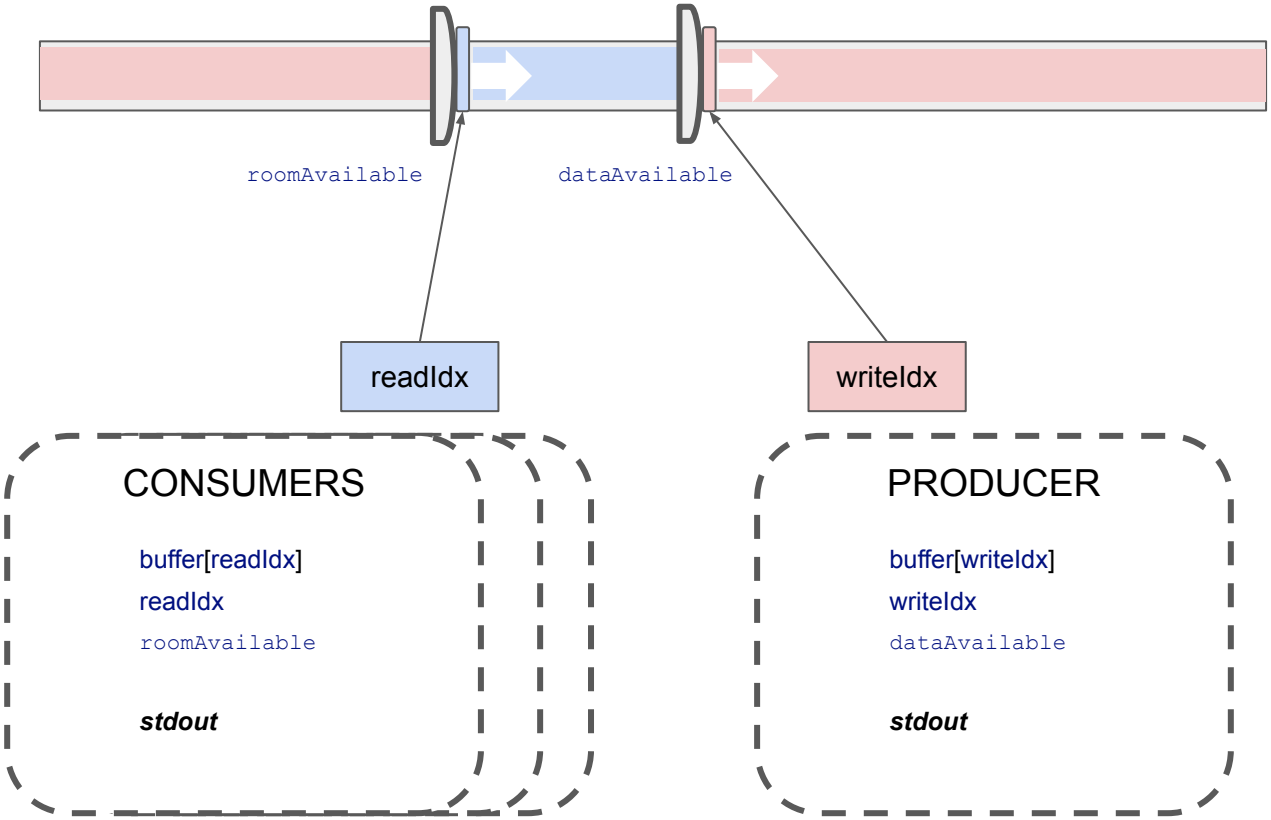
For each thread/process code list any data that is accessed for **write**



Synchronization (procedure idea)

2. create resource boundaries

For each thread/process code design boundaries where data can be accessed



prodcons_0 [ERROR]

CODING EXAMPLE

CODING EXAMPLE

PRODUCER

```
static void *producer(void *arg)
{
   int item = 0, i;
   for(i = 0; i < HISTORY_LEN; i++)
   {
       clock_gettime(CLOCK_REALTIME, &sendTimes[item]);
       while( (writeIdx+1)%BUFFER_SIZE == readIdx ) {
           pthread_cond_wait(&roomAvailable, &mutex);
       }
       buffer[writeIdx] = item;
       writeIdx = (writeIdx + 1)% BUFFER_SIZE;
       pthread_cond_signal(&dataAvailable);
       item++;
   }
   printf("fin.\n");
   return NULL;
}
```

CONSUMER

```
static void *consumer(void *arg)
{
   int item, i;
   for(i = 0; i < HISTORY_LEN; i++)
   {
       while(readIdx == writeIdx) {
           pthread_cond_wait(&dataAvailable, &mutex);
       }
       item = buffer[readIdx];
       readIdx = (readIdx + 1)% BUFFER_SIZE;
       pthread_cond_signal(&roomAvailable);

       // simulating a complex operation with 1us
       // average computation time
       wait_us(1);

       printf("%d\n",item);
       clock_gettime(CLOCK_REALTIME, &receiveTimes[item]);
    }
  return NULL;
}
```

Protecting for critical sections

```
[crtp@osboxes lab9]$ ./prodcons_0_err 1
0
1
2
3
4
fin.
5
6
7
8
9
Average Communication time: 2.71 ms
Overall Communication time: 7.31 ms
```

```
[crtp@osboxes lab9]$ ./prodcons_0_err 1
0
1
2
3
4
5
6
7
fin.
```

DEADLOCK !

a. Any operation to condition variables must be protected !

b. All shared resource operations must be protected

c. We need to add boundaries to the execution

prodcons_1 [ERROR]

CODING EXAMPLE

PRODUCER

```
static void *producer(void *arg)
{
   int item = 0, i;
   for(i = 0; i < HISTORY_LEN; i++)
   {
       clock_gettime(CLOCK_REALTIME, &sendTimes[item]);
       pthread_mutex_lock(& mutex);
       while( (writeIdx+1)%BUFFER_SIZE == readIdx ) {
           pthread_cond_wait(&roomAvailable, &mutex);
       }
       // printf("+");
       buffer[writeIdx] = item;
       writeIdx = (writeIdx + 1)% BUFFER_SIZE;
       pthread_cond_signal(&dataAvailable);
       pthread_mutex_unlock(& mutex);
       item++;
   }
   pthread_mutex_lock(& mutex);
   printf("fin.\n");
   pthread_mutex_unlock(& mutex);
   return NULL;
}
```

CONSUMER

```
static void *consumer(void *arg)
{
   int item, i;
   for(i = 0; i < HISTORY_LEN; i++)
   {
       pthread_mutex_lock(& mutex);
       while(readIdx == writeIdx) {
           pthread_cond_wait(&dataAvailable, &mutex);
       }
       item = buffer[readIdx];
       readIdx = (readIdx + 1)% BUFFER_SIZE;
       pthread_cond_signal(&roomAvailable);
       // simulating a complex operation with 1us
       // average computation time
       wait_us(1);
       // printf("-");
       printf("%d\n",item);
       clock_gettime(CLOCK_REALTIME, &receiveTimes[item]);
       pthread_mutex_unlock(&mutex);
   }
  return NULL;
}
```

matching data size

NOW seems ok .. but if we try to trigger 2 consumer treads …

```
[crtp@osboxes lab9]$ ./prodcons_1_err 2
0
1
2
3
4
5
fin.
6
7
8
9
```

STARVATION !

We need a method to match the number of item produced to the total consumed.

prodcons_2 [ERROR]

CODING EXAMPLE

CODING EXAMPLE

PRODUCER

```
static void *producer(void *arg)
{
   int item = 0, i;
   for(i = 0; i < HISTORY_LEN; i++)
   {
       clock_gettime(CLOCK_REALTIME, &sendTimes[item]);
       pthread_mutex_lock(& mutex);
       while( (writeIdx+1)%BUFFER_SIZE == readIdx ) {
           pthread_cond_wait(&roomAvailable, &mutex);
       }
       // printf("+");
       buffer[writeIdx] = item;
       writeIdx = (writeIdx +1)% BUFFER_SIZE;
       pthread_cond_signal(&dataAvailable);
       pthread_mutex_unlock(& mutex);
       item++;
   }
   pthread_mutex_lock(& mutex);
   finish = 1;
   printf("fin.\n");
   pthread_cond_broadcast(&dataAvailable);
   pthread_mutex_unlock(& mutex);
   return NULL;
}
```

CONSUMER

```
static void *consumer(void *arg)
{
   int item, i;
   for(;;)
   {
       pthread_mutex_lock(& mutex);
       while(readIdx == writeIdx) {
           pthread_cond_wait(&dataAvailable, &mutex);
           if(finish) {
               pthread_mutex_unlock(&mutex);
               return NULL;
           }
       }
       item = buffer[readIdx];
       readIdx = (readIdx +1)% BUFFER_SIZE;
       pthread_cond_signal(&roomAvailable);
       // simulating a complex operation with 1us
       // average computation time
       wait_us(1);
       printf("%d\n",item);
       pthread_mutex_unlock(&mutex);
       clock_gettime(CLOCK_REALTIME, &receiveTimes[item]);
    }
  return NULL;
}
```

Even if valgrind reports no apparent errors, there are some issues. Try to repeat the operation several times, sometimes the execution hangs, and sometimes not all values are correctly evaluated.

```
finish = 1;
printf("fin.\n");
pthread_cond_broadcast(&dataAvailable);
pthread_mutex_unlock(& mutex);
```

The consumer is not waiting and misses the broadcast to finish the run.

ctx sw

```
while(readIdx == writeIdx) {
        pthread_cond_wait(&dataAvailable, &mutex);

        DEADLOCK !

        if(finish) {
            pthread_mutex_unlock(&mutex);
            return NULL;
        }
    }
```

Even if valgrind reports no apparent errors, there are some issues. Try to repeat the operation several times, sometimes the execution hangs, and sometimes not all values are correctly evaluated.

```
        pthread_mutex_lock(& mutex);
        while( (writeIdx+1)%BUFFER_SIZE == readIdx ) {
            pthread_cond_wait(&roomAvailable, &mutex);
        }
        buffer[writeIdx] = item;
        writeIdx = (writeIdx +1)% BUFFER_SIZE;
        pthread_cond_signal(&dataAvailable);
        pthread_mutex_unlock(& mutex);
        item++;
    }
    pthread_mutex_lock(& mutex);
    finish = 1;
    printf("fin.\n");
    pthread_cond_broadcast(&dataAvailable);
    pthread_mutex_unlock(& mutex);
```

The producer trigger the dataAvailable signal but does not switch context…

then all data is produced and it sets the finish variable to true.

ctx sw

```
while(readIdx == writeIdx) {
            pthread_cond_wait(&dataAvailable, &mutex);

            if(finish) {
                pthread_mutex_unlock(&mutex);
                return NULL;
            }
        }
```

The consume wakes and terminates the run before all data have been processed

prodcons_3 [ERROR]

CODING EXAMPLE

PRODUCER

```
static void *producer(void *arg)
{
   int item = 0, i;
   for(i = 0; i < HISTORY_LEN; i++)
   {
       clock_gettime(CLOCK_REALTIME, &sendTimes[item]);
       pthread_mutex_lock(& mutex);
       while( (writeIdx+1)%BUFFER_SIZE == readIdx ) {
           pthread_cond_wait(&roomAvailable, &mutex);
       }
       // printf("+");
       buffer[writeIdx] = item;
       writeIdx = (writeIdx + 1)% BUFFER_SIZE;
       pthread_cond_signal(&dataAvailable);
       pthread_mutex_unlock(& mutex);
       item++;
   }
   pthread_mutex_lock(& mutex);
   finish = 1;
   printf("fin.\n");
   pthread_cond_broadcast(&dataAvailable);
   pthread_mutex_unlock(& mutex);
   return NULL;
}
```

CONSUMER

```
static void *consumer(void *arg)
{
   int item, i;
   for(;;)
   {
       pthread_mutex_lock(& mutex);
       if(finish) {
           pthread_mutex_unlock(&mutex);
           return NULL;
       }
       while(readIdx == writeIdx) {
           pthread_cond_wait(&dataAvailable, &mutex);
           if(finish) {
               pthread_mutex_unlock(&mutex);
               return NULL;
           }
       }
       item = buffer[readIdx];
       readIdx = (readIdx + 1)% BUFFER_SIZE;
       pthread_cond_signal(&roomAvailable);
       // simulating a complex operation with 1us
       // average computation time
       wait_us(1);
       printf("%d\n",item);
       pthread_mutex_unlock(&mutex);
       clock_gettime(CLOCK_REALTIME, &receiveTimes[item]);
    }
  return NULL;
}
```

prodcons_4 [OK !]

CODING EXAMPLE

PRODUCER

```
static void *producer(void *arg)
{
   int item = 0, i;
   for(i = 0; i < HISTORY_LEN; i++)
   {
       clock_gettime(CLOCK_REALTIME, &sendTimes[item]);
       pthread_mutex_lock(& mutex);
       while( (writeIdx+1)%BUFFER_SIZE == readIdx ) {
           pthread_cond_wait(&roomAvailable, &mutex);
       }
       // printf("+");
       buffer[writeIdx] = item;
       writeIdx = (writeIdx + 1)% BUFFER_SIZE;
       pthread_cond_signal(&dataAvailable);
       pthread_mutex_unlock(& mutex);
       item++;
   }
   pthread_mutex_lock(& mutex);
   finished = 1;
   printf("fin.\n");
   pthread_cond_broadcast(&dataAvailable);
   pthread_mutex_unlock(& mutex);
   return NULL;
}
```

CONSUMER

```
static void *consumer(void *arg)
{
   int item;
   for(;;)
   {
       pthread_mutex_lock(& mutex);
       while(readIdx == writeIdx) {
           if (!finished)
               pthread_cond_wait(&dataAvailable, &mutex);
           else {
               pthread_mutex_unlock(& mutex);
               return NULL;
           }
       }
       item = buffer[readIdx];
       readIdx = (readIdx + 1)% BUFFER_SIZE;
       pthread_cond_signal(&roomAvailable);
       // simulating a complex operation with 1us
       // average computation time
       wait_us(1);
       printf("%d\n",item);
       pthread_mutex_unlock(&mutex);
       clock_gettime(CLOCK_REALTIME, &receiveTimes[item]);
    }
  return NULL;
}
```

Parallel processing

There is still a huge issue affecting the parallel processing ….

… did you see it ??

Timing communication from producer to consumer

Now that we have a working solution we can just play with the numbers in prodcons.c

```
#define BUFFER_SIZE 10000
#define HISTORY_LEN 10000

andrea@HP:~/devel/unipd/crtp/src/lab9$ ./prodcons 1
Average Communication time: 315.16 ms (Std. Dev: 362.38 ms)
Overall Communication time: 612.78 ms
```

If we decrease the size of the buffer we implicitly ask the system to perform context switching because of all the firing of condition variables that set and unset mutex.

```
#define BUFFER_SIZE 10
#define HISTORY_LEN 10000

andrea@HP:~/devel/unipd/crtp/src/lab9$ ./prodcons 1
Average Communication time: 0.63 ms (Std. Dev: 0.64 ms)
Overall Communication time: 585.22 ms
```

threads	avg	all	avg_1K	all_1K	avg_10	all_10
1	308.18	604.69	57.84	607.41	0.64	590.73
2	150.74	308.15	27.88	292.25	0.34	287.88
3	93.46	187.33	18.51	194.43	0.26	208.71
4	68.95	138.03	13.48	141.05	0.19	141.2
8	28.14	56.48	5.79	60.62	0.1	61.53
12	20.25	39.81	4.02	41.91	0.08	42.58
16	13.66	27.55	3.42	35.51	0.08	37.03